Unedited Draft

EJB3 IN ACTION

Debu Panda
Reza Rahman
Derek Lane

MANNING

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

MEAP Edition
Manning Early Access Program

Copyright 2006 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Contents

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 1 What's what in EJB 3.0

One day, when God was looking over his creatures, he noticed a boy named Sadhu whose humor and cleverness pleased him. God felt generous that day and granted Sadhu three wishes. Sadhu asked for three reincarnations--one as a ladybug, one as an elephant, and the last as a cow. Surprised by these wishes, God asked Sadhu to explain himself. The boy replied, "I want to be a ladybug so that everyone in the world will admire me for my beauty and forgive the fact that I do no work. Being an elephant will be fun because I can gobble down enormous amounts of food without being ridiculed. I will like being a cow the best because I will be loved by all and useful to mankind." God was charmed by these answers and allowed Sadhu to live through the three incarnations. He then made Sadhu a morning star for his service to mankind as a cow.

EJB too has lived through three incarnations. When it was first released, the industry was dazzled by its innovations. But like the ladybug, EJB 1 had limited functionality. The second EJB incarnation was just about as heavy as the largest of our beloved pachyderms. The brave souls who could not do without its elephant-power had to tame the awesome complexity of EJB 2. And finally, in its third incarnation, EJB has become much more useful to the huddled masses, just like the gentle bovine that is sacred for Hindus and respected as a mother whose milk feeds us all..

A lot of people have put in a lot of hard work to make EJB 3 as simple and lightweight as possible without sacrificing enterprise-ready power. EJB components are now little more than Plain Old Java Objects (POJOs) that look a lot like code in a *Hello World* program. We hope you will agree with us as you read through the next Chapters that it has all the makings of a star.

We've strived to keep this book as earthy as possible without skimping on content. The book is designed to help you learn EJB 3.0 as quickly and as easily as possible. At the same time, we will not neglect to cover the basics where needed. We will also dive into deep waters with you where we can, share with you all the amazing sights we've discovered and warn you about any lurking dangers.

This book is about the radical transformation of a very important and uniquely influential technology in the Java World. We suspect you are not picking this book up to learn too much about EJB 2. You probably either already know EJB 2.x or are completely new to the world of EJB. In either case, spending too much time on previous versions is a waste of your time --you won't be surprised to learn that EJB 3.0 and EJB 2.x have very little in common. . If you are really curious about the journey that brought us to the current point, we encourage you to pick up one of the many good books on the previous versions of EJB.

Our goal in this Chapter is to tell you what's what in EJB 3.0, why you should consider using it, and, for EJB 2.x veterans, outline the significant improvements the newest version offers. We will then jump right into code in the next Chapter to build on the momentum of this one. With this goal in mind, we now start with a broad overview of EJB.

1.1 EJB Overview

In very straightforward terms, Enterprise Java Beans (EJB) is a platform for building portable, reusable and scalable business applications using Java programming language. Since its initial

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

incarnation, EJB has been touted a component model or framework to build enterprise Java applications without having to reinvent a lot of services such as transactions, security, automated, persistence, and so on that you may need for building an application. EJB lets application developers focus on building business logic without having to spend time on building infrastructure code.

From a developer's point of view an EJB is a piece of Java code that executes in a specialized runtime environment called the *EJB container* that provides a bunch of component services. The persistence services are provided by a specialized framework called the *persistence provider*. We'll talk more about the EJB container, persistence provider and services in Section 1.3.

In this section, you will learn how EJB functions as both a component and a framework. We'll also examine how EJB lends itself to building layered applications. We'll round off this section by listing a few reasons why EJB might be right for you.

1.1.1 EJB as a Component

In this book, when we talk about EJBs, we are talking about the server-side *components* that you can use to build parts of your application like the business logic or persistence code. A lot of us tend to associate the term *component* with developing complex and heavyweight CORBA, Microsoft COM+ code. In the brave new world of EJB 3.0, a component is really what it ought to be—nothing more than a POJO with some special powers. More importantly, these powers stay invisible until they are needed and don't distract from the real purpose of the component. You will see this firsthand throughout this book, especially starting with Chapter 2.

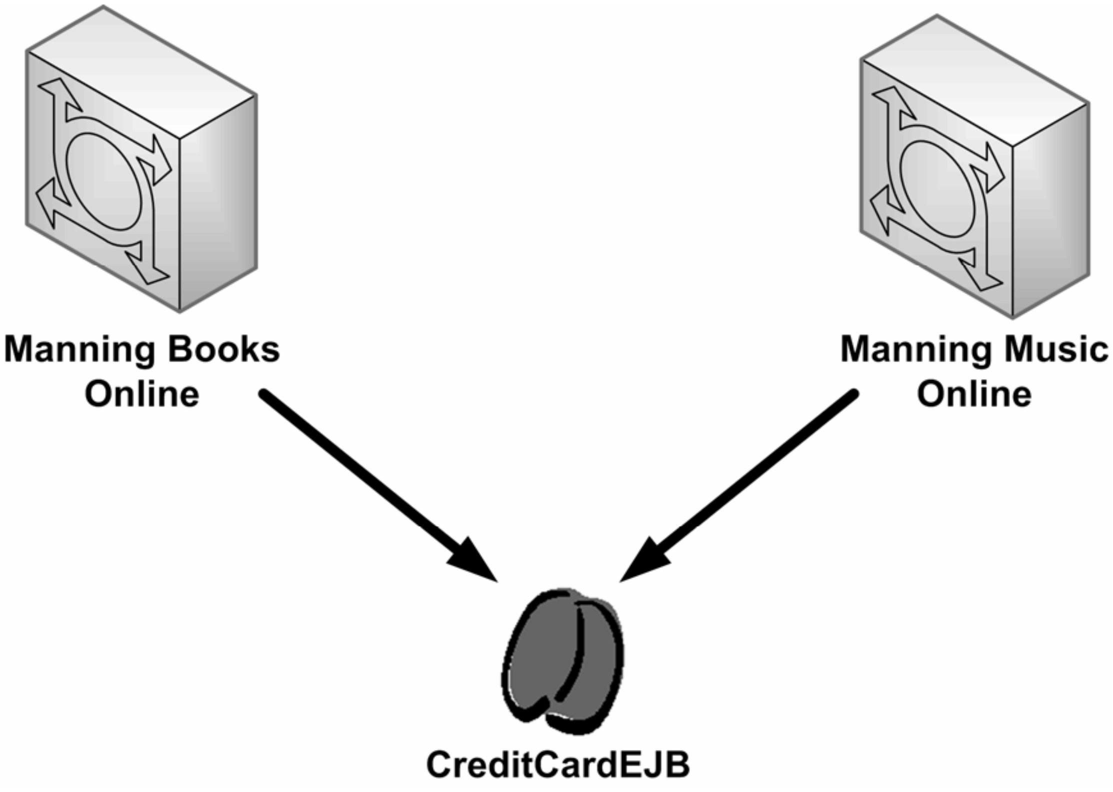
The real idea behind a component is that it should effectively encapsulate application behavior. The users of a component are not required to know its inner workings. All they need to know is what to pass in and what to expect back.

There are three types of EJB components: Session beans, Message Driven beans, and Entities. Session beans and Message driven beans are used to implement business logic in an EJB application and Entities are used for persistence.

Components can be reusable. For instance, suppose you were in charge of building a web site for an online merchant that sells technology books. You implemented a module to charge the Credit Card as part of a regular Java object. Your company did fairly well and you moved on to greener pastures. The company then decided to diversify and began developing a web site for selling CDs and DVDs. Since the deployment environment for the new site was different, it could not be located on the same server as your module. The person building the new site was forced to duplicate your Credit card module in the new website because there was no easy way to access your module. If you had implemented the Credit Card charging module as an EJB component as depicted in Figure 1.1 (or as a web service) it would have been much easier for the new person to access it by simply making a call to it when she needed that functionality. She could have reused it without having to understand its implementation.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



1 **Figure 1.1: EJB allows development of reusable components. For example you can build charging of CreditCard as an EJB that may be accessed by multiple applications.**

Having said that, building a reusable component requires very careful planning because, across enterprise applications within an organization, very little of the business logic may be reusable. Hence you may not care about the reusability of EJB components, but EJB still has much to offer as a framework as we will discover next section.

1.1.2 EJB As a Framework

As we mentioned, EJB components live in a container. Together, the components, or EJBs, and the container can be viewed as a framework that provides valuable services for enterprise application development.

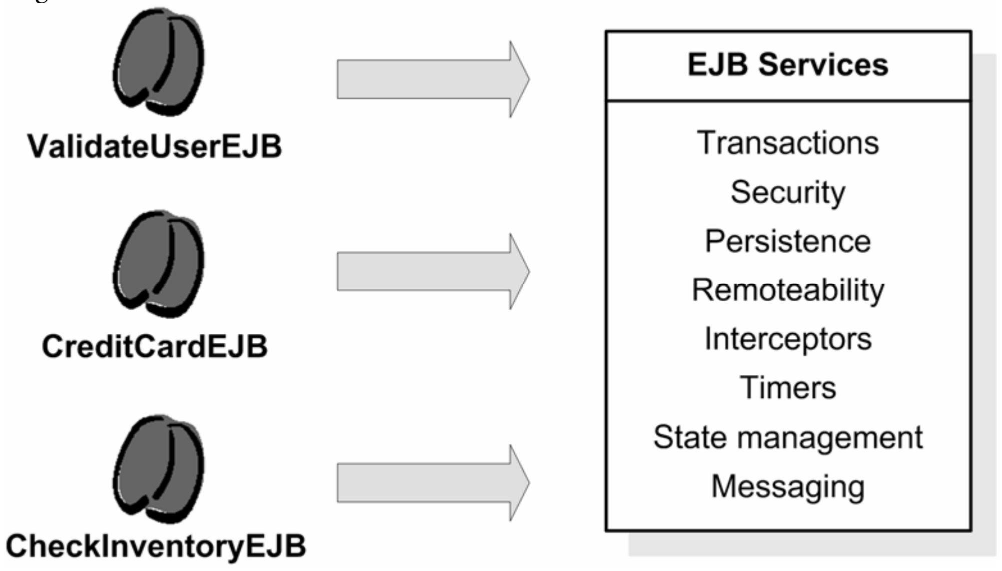
Although many people think EJBs are overkill for developing relatively simple web applications of moderate size, nothing could be farther from the truth. When you build a house you don't build everything from scratch. Instead, you buy materials or even the services of a contractor as you need it. It isn't too practical to build an enterprise application from scratch either. Most server-side applications have a lot in common, including churning business logic, managing application state, storing and retrieving information from a relational database, transaction management, security, asynchronous processing, system integration and so on.

As a framework, the EJB container provides these kinds of common functionality as out-of-the-box *services*, so that your EJB components can use them in your applications without reinventing the wheel. For instance, let's say that when you built the credit card module in your web application, you

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

wrote a lot of complex and error-prone code to manage transactions and security access control. You could have avoided that by using the declarative transaction and security services provided by the EJB container. These services, as well as many others you will learn about in section 1.3, are available to the EJB components when they are deployed in the EJB container, as you can see in figure 1.2. At the end of the day, this means writing high-quality, feature-rich applications much faster than you might think.



2 **Figure 1.2: EJB as a framework provides services to EJB components**

The container provides the services to the EJB components in a rather elegant new way; metadata annotations are used to pre-configure the EJBs by specifying the type of services to add when the container deploys the EJBs. Java 5 introduced metadata annotations, which are property settings that mark a piece of code, such as a class or method, as having particular attributes. This is a declarative style of programming, where the developer specifies what should be done and the system adds the code to do it.

In EJB, metadata annotations dramatically simplify development and testing of applications, without having to depend on an external XML configuration file. It allows developers to declaratively add services to EJB components as and when they need. As Figure 1.3 depicts, an annotation transforms a simple POJO into an EJB.

POJO
Annotation
EJB

3 **Figure 1.3: EJBs are regular Java objects configured using metadata annotations.**

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

As you will learn, annotations are used extensively throughout EJB, and not only to specify services. For example, an annotation is used to specify the type of the EJB component.

Although it's sometimes easy to forget it, enterprise applications have one more thing in common with a house. Both are meant to last, often much longer than anyone expects. Being able to support high-performance, fault-tolerant, scalable applications is an up-front concern for the EJB platform instead of being an afterthought. This means that you won't just be writing good server-side applications faster, but can expect your platform to grow with the success of your application. When the need to support a larger number of users becomes a reality, you won't have to rewrite your code. Thankfully these concerns are take care of by EJB container vendors. You'll be able to count on moving your application to a distributed, clustered server farm by doing nothing more than a little bit of configuration.

Last but certainly not least, with a world that's crazy about Service-Oriented Architecture (SOA) and interoperability EJB lets you turn your application into a Web Services powerhouse with ease when you need it to.

The EJB framework is a standard Java technology with an open specification. If it catches your fancy, you can check out the real deal on the Java Community Process (JCP) website at http://www.jcp.org/en/jsr/detail?id=220. What this means for you is that EJB is supported by a large group of people with competing but compatible implementations. On one hand, this means that the large group of people will work their best to keep EJB competitive. On the other hand, the ease of portability means that you get to pick and choose what implementation suits you best making your application portable across EJB containers from different vendors.

Having given you a high level introduction to EJB, let's turn our attention next to how EJB can be used to build layered applications.

1.1.3 Layered architectures and EJB

Most enterprise applications contain a pretty large number of components. Enterprise applications are designed to solve a unique type of customer problem, , but they share many common characteristics. For example, most enterprise applications have some kind of user interface and they implement business processes, model a problem domain, and save data into a database. Because of these commonalities, you can a follow a common architecture or design principles for building enterprise applications known as *patterns*.

For server-side development, the dominant pattern is *layered architectures*. In a layered architecture, components are grouped into tiers. Each tier in the application has a well-defined purpose, kind of like a profession, but more like a section of a factory assembly line. Each section of the assembly line does its designated part and passes on the remaining work on to the next one in line. In layered architectures, each layer delegates work to a layer underneath it.

EJB loosely allows you to build applications using two different layered architectures – the traditional four-tier architecture and Domain Driven Design (DDD). We will take a brief look at each of these architectures next.

Traditional Four Tier Layered Architecture

Figure 1.1 shows the traditional four-tier server architecture. This architecture is pretty intuitive and enjoys a good amount of popularity. In this architecture, the *presentation layer* is responsible for

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

rendering the graphical user interface and handling user input. The presentation layer passes down each request for application functionality to the business logic layer.

The *business logic layer* is the heart of the application and contains workflow and processing logic. In other words, business logic layer components model distinct actions or processes the application can perform, such as billing, search, ordering, user account maintenance and so on.

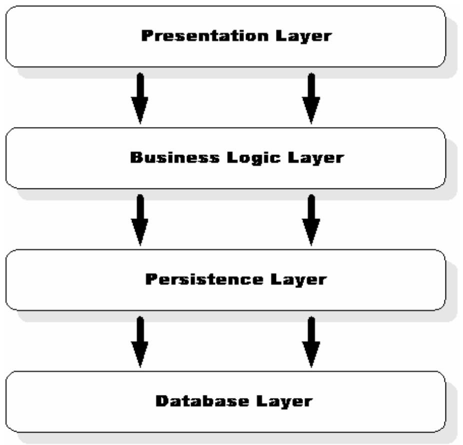


Figure 1.1: Most traditional enterprise applications have at least four layers. 1) The presentation layer is the actual user interface and can either be a browser or a desktop application. 2) The business logic layer defines the business rules. 3) The persistence layer deals with interactions with database. 4) The database layer consists up of a relational database such as Oracle that stores the persistent objects.

The business logic layer retrieves data from and saves data into the database by utilizing the persistence tier.

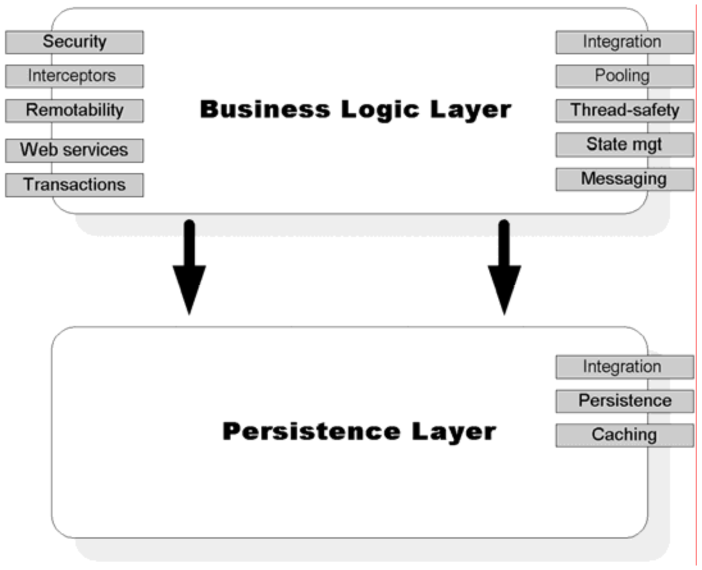
The *persistence layer* provides a high level Object-Oriented (OO) abstraction over the database layer.

The *database layer* typically consists of a relational database management system (RDBMS) like Oracle, DB2 or SQL Server.

EJB is obviously not a presentation layer technology. EJB is all about robust support for implementing the business logic and persistence layers. Figure 1.2 shows how EJB supports these layers via its services.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



5 **Figure 1.2: The component services offered by EJB 3.0 at each supported application layer. Note each service is independent of each other, so you are for the most part free to pick and choose the features important for your application. You'll learn more about services in section 1.3.**

In section 1.3, we will get into more detail about EJB services. And in section 1.2, we will explore the EJB bean types. For now, just note that the bean types called Session Beans and Message Driven Beans (MDBs) reside in and use the services in the business logic tier, and the bean types called Entities reside in and use the services in the persistence tier.

As groundbreaking as the traditional four-tier layered architecture is, it is not perfect. One of the most common criticisms of the traditional layered architecture is that it undermines the OO ideal of modeling the business domain as objects that encapsulate both data and behavior. Because the traditional architecture focuses on modeling business processes instead of the domain, the business logic tier tends to look more like a database driven procedural application rather than an Object Oriented one. Since persistence tier components are simple data holders, they look a lot like database record definitions rather than first class citizens of the OO World. Domain Driven Design (DDD) proposes an alternative architecture that attempts to solve these perceived problems.

Domain Driven Design (DDD)

The term Domain Driven Design[1] may be relatively new but the concept is not. Domain Driven Design emphasizes that *domain objects* should contain business logic and should not just be a dumb replica of database records. Domain objects are known as Entities in EJB 3.0 and we will discuss them in section 1.2. With DDD, the Catalog and Customer objects in a trading application are typical examples of entities and they may contain business logic.

In his excellent book "POJOs in Action," author Chris Richardson points out the problem with using domain objects just as a data holder.

"Some developers still view persistent objects simply as a means to get data in and out of the database and write procedural business logic. They develop what Fowler calls an "anemic domain model" [Fowler Anemic]. Just as anemic blood lacks vitality, anemic object models only superficially model the problem and consist of classes that implement little or no behavior"

[1] Domain Driven Design by Eric Evans Addison Wesley

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Yet, even though its value is clear, until this release of EJB, it was very difficult to implement DDD. Chris goes on to explain how EJB2 encouraged procedural code.

"… J2EE developers write procedural-style code [because] it is encouraged by the EJB architecture, literature, and culture, which place great emphasis on EJB components. EJB 2 components are not suitable for implementing an object model."

Admittedly, implementing a real domain model was almost impossible with EJB 2.x because beans were not POJOs and did not support many OO features such as inheritance and polymorphism. Chris specifically targets Entitity beans as the problem..

".EJB 2 entity beans, which are intended to represent business objects, have numerous limitations that make it extremely difficult to use them to implement a persistent object model."

The good news is that EJB 3.0 enables you to easily follow good object-oriented design or DDD. JPA Entities being POJOs support OO features such as inheritance or polymorphism. It's very easy to implement a persistence object model with the EJB 3.0 JPA. More importantly, you can easily add business logic to your Entities and, hence, implementing a rich domain model with EJB 3.0 is a trivial task.

Note though, many people don't like adding complex business-logic in the domain object itself and prefer creating a layer for procedural logic named *service layer* [2] or *application layer*. The application layer is really a much thinner version of the business logic layer similar to the traditional four-tier architecture. Not surprisingly, you can use Session Beans to build the service layer. Whether you use the traditional four-tier architecture or a layered architecture with Domain-Driven Design you can use Entities to model domain objects, including modeling state and behavior. We will discuss domain modeling with JPA Entities in Chapter 7.

Despite its impressive services and vision, EJB 3.0 is not the only act in town. You can combine various technologies to more or less match EJB services and infrastructure. For example you could use Spring with other open source technologies such as Hibernate and AspectJ to build your application, so why choose EJB 3.0? Glad that you asked. Next, we will unravel the reasons behind that.

1.1.5 Why Choose EJB 3.0?

At the beginning of the Chapter, we hinted at EJB's status as a pioneering technology. EJB is a groundbreaking technology that has raised the standards of server-side development. Just like Java itself, EJB has changed things in ways that are here to stay and inspired many innovations beyond itself. In fact, up until a few years ago, the only serious competition to EJB came from the .NET framework.

In this section, we'd like to point out a few of the compelling features of EJB 3.0 that we feel certain will have this latest version at the top of your short list.

Ease of Use

Thanks to the unwavering focus on ease of use, EJB 3.0 is probably the simplest server-side development platform around. The features that shine the brightest are POJO programming, annotations in favor of verbose XML, heavy use of sensible defaults, and JPA, all of which you will be learning about in this book. Although significant in number, you might find that EJB services are

[2] Patterns of Enterprise Application Architecture by Martin Fowler

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

very intuitive and only as complicated as its most common usage pattern. For the most part, EJB 3.0 has a practical outlook on things and doesn't demand that you be a genius computer scientist that understands the theoretical intricacies of the services EJB offers. In fact, most EJB services are designed to give you a break from this mode of thinking so you can focus on getting the job done and go home at the end of the day knowing you accomplished something.

Integrated Solution Stack

EJB 3.0 attempts to offer you a complete stack of server solutions including persistence, messaging, lightweight scheduling, remoting, Web Services, dependency injection (DI) and interceptors. This means that you won't have to spend a lot of time looking for third-party tools to integrate into your application.. In addition, EJB 3.0 provides seamless integration with other Java EE technologies like JDBC, JavaMail, JTA (Java Transaction API), JMS (Java Messaging Service), JAAS (Java Authentication and Authorization Service), JNDI (Java Naming and Directory Interface), Java RMI (Remote Method Invocation), and so on. EJB is also guaranteed to seamlessly integrate with presentation tier technologies like JSP, Servlets, JSF and Swing.

Open Java EE Standard

EJB is a critical part of the Java EE standard. This is an extremely important concept to grasp if you are to adopt it. EJB 3.0 has an open, public API specification against which any company is encouraged to create a container or persistence provider implementation. The EJB 3.0 standard is developed by the Java Community Process (JCP), which consists of a non-exclusive group of individuals driving the Java standard. The open standard leads to broader vendor support for EJB 3.0 and that means better choice for you instead of depending upon a proprietary solution.

Broad Vendor Support

EJB is supported by a large and diverse variety of independent organizations. This includes the technology world's largest, most respected and most financially strong names like Oracle and IBM as well as passionate and energetic open source groups like JBoss and Geronimo.

Wide vendor support means three important facts. Firstly, you are not at the mercy of the ups and downs of a particular company or group of people. Secondly, a lot of people have concrete long-term interests to keep the technology as competitive as possible. You can essentially count on being able to take advantage of the best of breed technologies both in and outside the Java world in a competitive timeframe. Lastly, vendors have historically competed against each other by providing value added non-standard features. All of these factors help keep EJB on the track of continuous healthy evolution.

Stable High Quality Code Base

Although EJB 3.0 is a radical evolutionary step, most application server implementations will still benefit from a relatively stable code base that has lived through some of the most demanding enterprise environments over a prolonged period of time. Most persistence provider solutions like JDO, Hibernate and TopLink are also stable products that are being used in many mission critical production environments. This means that although EJB 3.0 is very new, you can expect pretty stable implementations relatively quickly. Also, because of the very nature of standards based development, EJB 3.0 container implementations are generally not taken lightly. To some degree this helps ensure a healthy level of inherent implementation quality.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Clustering, Load-balancing and failover

Although the EJB specification does not make it a requirement, a feature historically added by almost every application server vendor is robust support for clustering, load balancing and failover. EJB application servers have a proven track record of supporting some of the largest high-performance computing (HPC) enabled server farm environments. More importantly, you can leverage such support with no changes to code, no third-party tool integration and relatively simple configuration beyond the inherent work in setting up a hardware cluster. This means that you can rely on hardware clustering to scale up your application with EJB 3.0 if you need to.

EJB 3.0 is a compelling option for building enterprise applications. In the following sections, we're going to tell you more about EJB types and how to use them. We will help you discover what are containers and persistence providers and what type of services they provide. By the time you have finished reading sections 1.2 and 1.3 you will have a pretty good idea what EJBs are, where they run and what services they provide! So let's get started by learning more about EJBs, their types and how they work.

1.2 Discovering Types of EJBs

From our own experience, when we work on any software products we are always on fire because we always have a tight deadline to meet. Most of us try to beg, borrow or steal reusable code to make our life easier. Gone are those days when developers had the luxury to build their own infrastructure when building a commercial application. While there are several commercial and open source frameworks available that can simplify application development, EJB is a very compelling framework that has a lot to offer.

If you're like us, you must be getting excited about EJB by now, and eager to learn more. So let's jump right in and see how you can use EJB as a framework to build your business logic and persistence tier of your applications, starting with the beans.

In EJB-speak, a component is a "bean". If your manager doesn't find the Java-"Coffee bean" play on words cute either, blame Sun's marketing department. Hey, at least we get to hear people in suits use the words "enterprise" and "bean" in close sequence as if it were perfectly normal…

As we mentioned, EJB classifies beans into three types based on what they are used for:

Session Beans

Message Driven Beans

Entities

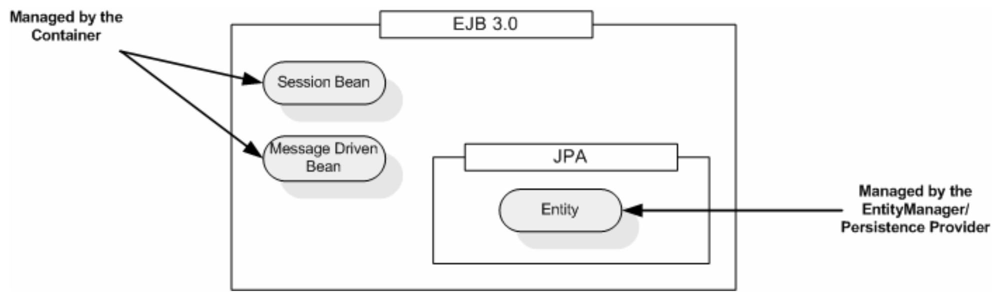
Each bean type serves a specific purpose and can use a specific subset of EJB services. The real purpose of bean types is to safeguard against overloading them with services that cross wires. This is kind of like making sure the accountant in the horn-rimmed glasses doesn't get too curious about what happens when you touch both ends of a car battery terminal at the same time. Bean classification also helps to understand and organize an application in a sensible way, for example, bean types help you develop applications based on a layered architecture.

As we've briefly mentioned, Session Beans and Message Driven Beans are used to build business logic, and they live in the *container, which* manages these beans and provides services to them. Entities are used to model the persistence part of an application. Like the container, it is the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

persistence provider that manages Entities. A persistence provider is pluggable within the container and is abstracted behind the *Java Persistence API* (JPA). This organization of the EJB 3.0 API is shown in Figure 1.3.



6

7 **Figure 1.3: Overall organization of the EJB 3.0 API. The persistence part of the API is completely separable, the JPA. The business logic processing is done through two component types, Session Beans and Message Driven Beans. Both of these components are managed by the container. Persistence components are called Entities, which are managed by the persistent provider through the EntityManager interface.**

We'll discuss the container and the persistence provider in section 1.3. For the time being, all you need to know is that these are separable parts of an EJB implementation, each of which provide support for different EJB component types.

Let's start digging a little deeper into the different EJB component types, starting with Session Beans.

1.2.1. Session Beans

A Session Bean is invoked by a client to perform a specific business operation such as checking the credit history for a customer. The name "session" implies that a bean instance is available for the duration of a "unit of work" and does not survive a server crash or shutdown. A Session Bean can model any application logic functionality. There are two types of Session Beans: *stateful* and *Stateless* Session Beans.

A Stateful Session Bean automatically saves bean state between client invocations without your having to write any additional code. The typical example of a state-aware process is the shopping cart for a web merchant like Amazon. In contrast, Stateless Session Beans do not maintain any state and model application services that can be completed in a single client invocation. You could build Stateless Session Beans for implementing business processes like charging a credit card or checking customer credit history.

A Session Bean can be invoked either locally or remotely using Java Remote Method Invocation (RMI) or a stateless session bean can be exposed as a Web Service.

1.2.2 Message Driven Bean (MDB)

Like Session Beans, Message Driven Beans process business logic. However, Message Driven Beans are different in a very important way. Clients never invoke Message Driven Bean methods directly. Instead, Message Driven Beans are triggered by messages sent to a messaging server, which enables sending asynchronous messages between system components . Some typical examples of messaging servers include IBM WebSphere MQ, SonicMQ, Oracle Advanced Queueing, Tibco and many

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

more. Message Driven Beans are typically used for robust system integration or asynchronous processing. An example of messaging could be sending an inventory-restocking request from an automated retail system to a supply chain management system. Do not worry too much about messaging too much we are going to discuss in greater details when time is ripe.

Next we will learn about entities and Java Persistence API. Before that we will explain what is persistence and describe how Object-Relational Frameworks help use automated persistence.

1.2.3 Entities and the Java Persistence API (JPA)

One of the exciting new features of EJB3 is in the way it handles persistence. We've briefly mentioned persistence providers and the JPA before, but now let's dig down into the details.

As you probably know, *persistence* is the ability to have data contained in Java objects automatically stored into a relational database like Oracle, SQL Server and DB2. Persistence in EJB 3.0 is managed by the Java Persistence API (JPA). It automatically persists the Java objects using a technique called Object Relational Mapping (ORM). ORM is essentially the process of mapping data held in Java objects to database tables using configuration. It relieves you of the task of writing low-level, boring, and complex JDBC code to persist objects into database.

The frameworks that provide ORM capability to perform automated persistence are known as Object-Relational Mapping frameworks. As the name implies an O-R framework perform transparent persistence by making use of O-R mapping metadata that defines how objects are mapped to database tables. O-R Mapping is not a new concept and has been around for a while. Oracle TopLink is probably the oldest ORM farmework in the market whereas open source framework JBoss Hibernate popularized ORM concepts among the mainstream developer community.

In EJB3 terms, a persistence provider is essentially an O-R framework that supports the EJB3 Java Persistence API.. The JPA defines standard ORM configuration metadata, a standard API to perform CRUD (Create, Read, Update, Delete) persistence operations named the *EntityManager* and the *Java Persistence Query Language* (JPQL) to search and retrieve persisted application data. Since JPA standardizes O-R frameworks for the Java platform, you can plug in ORM products like JBoss Hibernate, Oracle TopLink or BEA Kodo (JDO) as the underlying JPA "persistence provider" for your application.

Something that might occur to you is the fact that automated persistence is something that is useful for all kinds of applications, not just server-side applications like those built with EJB. After all, JDBC, the grandfather of JPA, is used in everything from large-scale real-time systems to desktop-based hacked-up prototypes. This is exactly why JPA is completely separable from the rest of EJB 3.0 and usable in plain Java SE environments.

Entities are the Session Bean and MDB equivalent in the JPA world. Let's take a quick glance at it next, as well as the `EntityManager` API and the Java Persistence Query Language (JPQL).

Entities

If you are using JPA to build persistence logic of your applications then you have to use entities. Entities are the Java objects that are persisted into the database. Just as Session Beans model processes, Entities model lower-level application concepts that high-level business processes manipulate. While Session Beans are the "verbs" of a system, Entities are the "nouns". Examples include an *Employee* Entity, a *User* Entity, an *Item* Entity and so on. Another perfectly valid (and often simpler to understand) way of looking at Entities is that they are the OO representations of the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

application data stored in the database. In this sense, Entities survive container crashes and shutdown. You must be wondering how the persistence provider knows where the entity will be stored. The real magic lies in the O-R mapping metadata – basically an Entity contains the data that specifies how it is mapped to the database. You will see an example of this in the next chapter. JPA Entities support a full range of relational and OO capabilities including relationships between Entities, inheritance and polymorphism.

The EntityManager

The JPA `EntityManager` interface manages Entities in terms of actually providing persistence services. While Entities tell a JPA provider how they map to the database, they do not persist themselves. The `EntityManager` interface reads the O-R mapping metadata for an Entity and actually performs persistence operations. Namely, the `EntityManager` knows how to add Entities to the database, update stored Entities, delete Entities and retrieve Entities from the database. In addition, the JPA also provides ability to handle life-cycle management, performance tuning, caching and transaction management.

The Java Persistence Query Language (JPQL)

JPA provides a specialized SQL-like query language called the Java Persistence Query Language (JPQL) to search for Entities saved into the database. The existence of a robust and flexible API like JPQL means that you don't lose anything by choosing automated persistence instead of hand-written JDBC. In addition, JPA supports native, database specific SQL, in the rare cases where they are worth using.

At this point, you should have a decent high-level view of the different parts of EJB. you also know that EJB 3.0 components (session beans and MDBs) and entities need to be deployed in an EJB 3.0 container and persistence providers respectively so that they can make use of the services EJB 3.0 provides. The container, the persistence provider and the services are central concepts in EJB 3.0 and we'll address them next.

1.3 Getting Inside EJB

- When you build a simple Java class you need a Java Virtual Machine (JVM) to execute it. Similary you need an EJB container to execute Session beans and MDBs and a Persistence provider to run your Entities. In this section you will give a birds-eye-view of containers and persistence providers and how they are related.
- In the Java world, containers aren't just limited to the realm of EJB 3.0. You're probably familiar with a web container, which allows you to run web based applications using Java technologies such as Servlet, JSP or JSF. A *Java EE container* is an application server solution that supports EJB 3.0, a web container, and other Java EE APIs and services. BEA WebLogic Server, GlassFish, IBM Websphere, JBoss Application Server and Oracle Application Server 10g are examples of Java EE containers.
- The relationship between the Java EE container, web container, EJB container and JPA persistence provider is shown in Figure 1.4.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

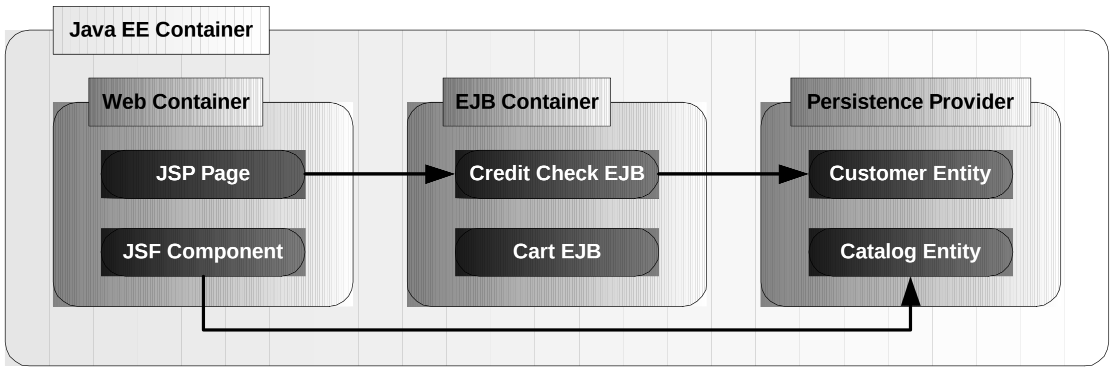


Figure 1.4: A Java EE container typically contains web and EJB containers and a persistence provider. The Stateless session bean (Credit Check EJB) and Stateful Session bean (Cart EJB) are deployed and run in the EJB container. Entities (Customer and Catalog) are deployed and run within an EJB persistence provider and can be accessed by either web or EJB container components.

If you install a Java EE compliant application server such as Sun's Glassfish, it will contain a preconfigured web container, EJB container and a JPA provider. However some vendors and open source projects may provide only a web container such as Tomcat or an EJB 3.0 compliant persistence provider such as Hibernate. You have to realize that these containers provided limited functionalities compared to what you get with a complete Java EE 5.0 container.

- In this section, we'll focus on how the EJB container and the Persistence Provider work and we'll finish with a fuller discussion about the EJB services. First, let's tackle the EJB container.

1.3.2 Accessing EJB services: the EJB Container

The best way to think of the container is simply an extension of the basic idea of a Java Virtual Machine (JVM). Just as the JVM transparently manages memory on our behalf, the container transparently provides EJB component services like transactions, security management, remoting, web services support and so on. As a matter of fact, you might even think of the container as a JVM on steroids, whose specific purpose in life is to execute EJBs. In EJB 3.0, the container only provides services applicable to Session Beans and Message Driven Beans. The task of putting an EJB 3.0 component inside a container is called "deployment". Once an EJB is successfully deployed in a container it can be used in your applications.

The persistence provider is the container counterpart in JPA. Let's briefly talk about it next.

1.3.3 Accessing JPA services: the Persistence Provider

- In section 1.2.3, we mentioned that the persistence provider's job is to provide standardized JPA services. Let's explore how it does that. Instead of following the JVM-like container model, JPA follows a model similar to APIs like JDBC . This means that while the container provides its services at runtime without your explicitly asking for it. JPA provides persistence services like retrieving, adding, modifying and deleting JPA Entities when you explicitly ask for them by invoking `EntityManager` API methods.

The "provider" terminology comes from APIs like JDBC and JNDI too. If you've worked with JDBC, you know that a "provider" is essentially the vendor implementation the JDBC API uses

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

under the cover. Products that provide JPA implementation are *persistence providers* or *persistence engines*. JBoss Hibernate and Oracle TopLink are two popular JPA providers.

Since JPA is completely pluggable and separable, the persistence provider and container in an EJB 3.0 solution need not come from the same vendor. For example, you could use Hibernate inside a BEA WebLogic container if it suits you better instead of the Kodo implementation WebLogic ships with.

But without services, what good are containers, you may ask. In the next section, let's explore the services concept critical to EJB.

1.3.3 Gaining functionality with EJB Services

The first thing that should cross your mind while evaluating any technology is what it really gives you. What's so special about EJB? Beyond a presentation layer technology like JSP, JSF or Struts, couldn't you create your web application using just the Java language and maybe some APIs like JDBC for database access? The plain answer is that you could, if deadlines and cutthroat competition were not realities. Indeed, before anyone dreamed up EJB, this is exactly what people did. What the resulting long hours proved is that you tend to spend a lot of time solving very common system level problems instead of focusing on the real business solution. These bitter experiences flushed out the fact that there are common solutions that can be reused to solve common development problems. This is exactly what EJB brings to the table.

EJB is really a collection of "canned" solutions to common server application development problems as well as a roadmap to common server component patterns. These "canned" solutions, or *services*, are provided by either the EJB container or the persistence provider. To access those services, you build the application components and deploy them into the container. Most of this book will be spent explaining how you can exploit EJB services.

In this Section, we will briefly introduce some of the services EJB offers. Obviously, we can't explain the implementation details of each service in this section. Neither is it really necessary to cover every service EJB offers right now. Instead we will briefly list the major EJB 3.0 services in Table 1.1 and mention what they mean to you from a practical perspective.

1.1 Table 1.1: Major EJB 3.0 Component Services and why they are important to you. The persistence services are provided by the JPA provider.

1.2

Service	**Applies to**	**What does it mean for you**
Integration	Session Beans and MDB	Helps glue together components, ideally through simple configuration instead of code. In EJB 3.0, this is done through dependency injection (DI) as well as lookup.
Pooling	Stateless Session Beans, MDB	For each EJB component, the EJB platform creates a pool of component instances that are shared by clients. At any given point in time, each pooled instance is only allowed to be used by a single client. As soon as an instance is done servicing a client, it is returned to the pool for reuse instead of being frivolously discarded for the garbage collector to reclaim.
Thread-safety	Session Beans and MDB	EJB makes all components thread-safe and highly peformant in ways that are completely invisible. This means that you can write your server components as if you were developing a single threaded desktop application. It doesn't matter how complex the component itself is, EJB will make sure it is thread-safe.
State management	StatefulSession beans	The EJB container manages state completely transparently for Stateful components instead of having you to write verbose and error-prone code for state management. This means that you can maintain state in instance variables as if you were developing a desktop application. EJB takes care of all the details of session maintenance behind the scenes.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Messaging	MDB	EJB 3.0 allows you to write messaging aware components without having to deal with a lot of the mechanical details of the Java Messaging Service (JMS) API.
Transactions	Session beans and MDB	EJB supports declarative transaction management that helps us add transactional behavior to components using simple configuration instead of code. In affect, you can designate any component method to be transactional. If the method completes normally, EJB commits the transaction and makes the data changes made by the method permanent. Otherwise the transaction is rolled back.
Security	Session Beans	EJB supports integration with the Java Authentication and Authorization Service (JAAS) API, so it is very easy to completely externalize security and secure an application using simple configuration instead of cluttering up your application with security code.
Interceptors	Session Beans and MDB	EJB 3.0 introduces AOP in a very lightweight, accessible manner using *Interceptors*. This allows you to easily separate out crosscutting concerns such as logging, auditing, and so on in a configurable way.
Remote Access	Session Beans	EJB 3.0, you can make components remotely accessible without writing any code. In addition, EJB 3.0 enables client code to access remote components as if they were local components using dependency injection.
Web services	Stateless Session Beans	EJB 3.0 can transparently turn business components into robust Web Services with minimal code change.
Persistence	Entities	Providing standards-based, 100% configurable, automated persistence as an alternative to verbose and error-prone JDBC/SQL code is a principal goal of the EJB 3.0 platform.
Caching and Performance	Entities	In addition to automating persistence, JPA also transparently provides a number of services geared toward data caching, performance optimization and application tuning. These services are invaluable in supporting medium to large-scale systems.

You will learn how to use each of these services in your application throughout in this book.

Despite its robust features, one of the biggest beefs people had with EJB 2.x was that it was too complex. It was clear that EJB 3.0 had to make development as simple as possible instead of simply continuing to add additional features or services. If you have worked with EJB 2.x or have simply heard or read that it is complex, you should be curious as to what makes EJB 3.0 different. Let's tackle this question next.

1.4 Renaissance of EJB

[[JC: can you think of a better word to use? "Renaissance" feels like EJB2 was stuck in the dark ages and no one used it

DP: This word was suggested by Lianna]]

Software is organic. Much like carbon-based life forms, software grows and evolves. Features die. New features are born. Release numbers keep adding up like the rings of a healthy tree. EJB is no exception to the rule of software evolution. In fact, as far as technologies go, the saga of EJB is more about change than it is about stagnation. Only a handful of other technologies can boast the robust metamorphosis and continuous improvements EJB has pulled off.

It's time to get a glimpse of the new incarnation of EJB starting with an example of a simple stateless session bean and then revealing the features changes that make EJB and easy–to-use development tool.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

In order to explore the new features of EJB3, we will be pointing out some of the problems associated with EJB2. If you are not familiar with EJB2, don't worry – the important information is how the problems have been resolved in EJB3.

The problems associated with EJB2 have been widely discussed. In fact, there have been whole books[3] written about it. Chris Richardson in POJOs in Action rightfully identified amount of sheer code you had to write to build an EJB:

> *You must write a lot of code to implement an EJB*—You must write a home interface, a component interface, the bean class, and a deployment descriptor, which for an entity bean can be quite complex. In addition, you must write a number of boilerplate bean class methods that are never actually called but that are required by the interface the bean class implements. This code isn't conceptually difficult, but it is busywork that you must endure."

In this section, we'd like to walk through some of those pain points and show you how they have been resolved in EJB 3.0. As you will see, EJB 3.0 specifically targets the thorniest issues in EJB 2.x and solves them primarily through bold adoption and clever adaptation of the techniques widely available in popular open source solutions such as Hibernate and Spring. Both of which have passed the "market incubation test" without getting too battered. In many ways, this release primes EJB for even further innovations by solving the most immediate problems and creating a buffer zone for the next metamorphosis.

But first, let's look at a bit of code. You will probably never use EJB2 for building simple applications such as HelloWorld. However we want to show you a simple EJB implementation of the ubiquitous 'Hello World" developed using EJB3. We want to you to see this code for a couple reasons: first, to demonstrate how simple developing with EJB 3.0 really is, and second, because this will provide context for the discussions in the following sections and make them more concrete.

1.4.1 HelloUser Example

HelloWorld examples have ruled the world since they first appeared in "The C Programming Language" by Kernighan and Ritchie. Hello World' caught on and held ground for good reason. It is very well suited to focus on introducing a technology as simply and plainly as possible. While almost every technology book starts with a HelloWorld example, to keep things lively and relevant we were planning to deviate from that rule and provide a slightly different example.

In 2004, I wrote an article for the TheServerSide.com in which I stated that when EJB3 was released, it would be so simple you could write a Hello World in it using only a few lines of code. Any experienced EJB2 developer knows that this couldn't be done easily in EJB2. You would need to write a home interface, a component interface, a bean class and a deployment descriptor.. Well, now that EJB3 has been finalized, let's see if I was right in my prediction. Listing 1.1 shows

1 Listing 1.1: HelloUser Session bean

```
package ejb3inaction.example;
public interface HelloUser {                                          |#1
    public void sayHello(String name);
}

package ejb3inaction.example;
```

[3] Bitter EJB: Manning Publications,2003

Licensed to John Sweitzer <admin@saolailaem.info>

```
import javax.ejb.Stateless;
@Stateless                                                           |#3
public class HelloUserBean implements HelloUser {                    |#2
    public void sayHello(String name) {
        System.out.println("Hello " + name + " welcome to EJB 3.0!");
    }
}
```
(annotation) <#1 HelloUser POJI>
(annotation) <#2 HelloUserBean POJO>
(annotation) <#3 Stateless annotation>

Believe it or not, this is a complete and self-contained example of a working EJB! Note that for simplicity we have kept both the interface and class as part of same listing. As you can see, the EJB does not look much more complex than your first Java program. The interface is a regular Java interface (POJI) and the bean class is a regular Java class (POJO). The funny `@Stateless` symbol in Listing 1.1 is a metadata annotation#3 that converts the POJO to a full-powered Stateless EJB. If you are not familiar with metadata annotations, we will explore them in Chapter 2. In affect, they are "comment-like" configuration information that can be added to Java code.

To execute this EJB you have to deploy it to the EJB container. If you really want to execute this sample download the chapter1.zip from http://ejb3inaction.com and follow the online instructions to deploy and run it in your favorite EJB container.

However don't worry too much about the details of this code right now; it's just a simple illustration. We'll dive into coding details in the next Chapter. Our real intent for the HelloWorld example was to use this as a basis for discussions how EJB3 addresses the thorniest issues that branded EJB2 as an elephantine.

Let's move on now and take a look at what has transformed the EJB-elephant into the EJB-cow.

1.4.2 Simplified Programming Model

We heartily agree with Chris Richardson's quote at the beginning of this section - one of the biggest problems with EJB 2.x was the sheer amount of code to generate for a given component

If we had attempted to produce listing 1.1 as an EJB2 example, we would have had to work with several classes and interfaces just to produce the simple one-line output. All of these class and interfaces had to either implement or extend EJB API interfaces with rigid and unintuitive constraints like throwing `java.rmi.RemoteException` for all methods. Implementing interfaces like `javax.ejb.SessionBean` for the bean implementation class was particularly bad since you had to provide an implementation for life-cycle callback methods like `ejbCreate`, `ejbRemove`, `ejbActivate`, `ejbPassivate` and `setSessionContext`, whether you actually used them or not. In effect, you were forced to deal with a lot of mechanical steps to accomplish very little. IDE tools like JBuilder, JDeveloper and WebSphere Studio helped matters a little bit by automating some of the mechanical steps. However, in general, decent tools with robust support were extremely expensive and clunky.

As you saw in Listing 1.1, EJB 3.0 enables you to develop an EJB component using Plain Old Java Objects (POJOs) and Plain Old Java Interfaces (POJIs) that know nothing about platform services.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

You can then apply configuration metadata, using annotations, to these POJO and POJI to add platform services such as remoteability, web services support and life cycle callbacks only as needed.

The largely redundant step of creating home interfaces has been done away with altogether. In short, EJB service definitions have been moved out of the type-safe world of interfaces into deploy and runtime configuration where they are suited best. A lot of mechanical steps that were hardly ever used have now been automated by the platform itself. Hence *you do not* have to write a lot of code to implement an EJB!

1.4.3 Annotations instead of Deployment Descriptors

In addition to having to write a lot of boilerplate code, a significant hurdle in managing EJB 2.x was the fact that you still had to do a lot of XML configuration for each component. Although XML is a great mechanism, the truth is that not everyone is a big fan of its verbosity, poor readability and fragility.

Before the arrival of Java 5 metadata annotations, there was no other viable alternative but to use XML for configuration. EJB 3.0 allows us to use metadata annotations to configure a component instead of using XML deployment descriptors.. As you might be able to guess from Listing 1.1, besides getting rid of verbosity, annotations help avoid the monolithic nature of XML configuration files and localizes configuration to the code that is being affected by it. Note though, you can still use XML deployment descriptors if they suit you better or simply to supplement annotations. We'll talk more about this in Chapter 2.

Other than making the task of configuration easier, EJB 3.0 also reduces the total amount of configuration altogether by using sensible defaults wherever possible. This is especially important while dealing with automated persistence using ORM, as we will see in Chapters 7, 8, 9, and 10.

1.4.4 Dependency Injection Instead of JNDI Lookup

One of the most tedious parts of EJB 2.x development was writing the same few lines of boilerplate code many times to do a JNDI lookup whenever you needed to access an EJB or a container-managed resource such as a pooled database connection handle. Chris Richardson sums it up well:

A traditional J2EE application uses JNDI as the mechanism that one component uses to access another. For example, the presentation tier uses a JNDI lookup to obtain a reference to a session bean home interface. Similarly, an EJB uses JNDI to access the resources that it needs, such as a JDBC `DataSource`. The trouble with JNDI is that it couples application code to the application server, which makes development and testing more difficult. (POJOs In Action)

In EJB 3.0, JNDI lookups have been turned into simple configuration using metadata based dependency injection (DI). For example, if you want to access the `HelloUser` EJB that we saw in Listing 1.1 from another EJB or Servlet we could use code that looks like the following:

```
...
@EJB
private HelloUser helloUser;

void hello(){
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    helloUser.sayHello("Curious George");
}
...
```

Isn't that great? The `@EJB` annotation transparently "injects" the `HelloUser` EJB into the annotated variable. EJB 3.0 dependency injection essentially gives you a simple abstraction over a full-scale enterprise JNDI tree. Note you can still use JNDI lookups where they are absolutely unavoidable.

1.4.5 Simplified Persistence API

A lot of the problems with the EJB 2.x persistence model were due to the fact that it was applying the container paradigm to a problem for which it was ill suited. This made the EJB 2.x Entity Bean programming model extremely complex and unintuitive, even on top of the previously mentioned problems of EJB 2.x, such as excessive code volume and "XML hell". One of the prime motivators behind making Entity Beans container-managed was enabling remote access. In reality, very few clients made use of this feature because of performance issues, opting to use Session Beans as the remote access point.

Undoubtedly Entity Beans were easily the worst part of EJB 2.x.. EJB 3.0 solves these problems by using a more natural API paradigm centered on manipulating metadata based POJOs through the `EntityManager` interface. Moreover, EJB 3.0 Entities do not carry the unnecessary burden of remote access.

Another limitations with EJB2 was that you cannot send an EJB2 entity bean across the wire in different tiers. .. EJB developers discovered an anti-pattern for this problem and it was to add another layer of objects: the DTOs. Chris sums it up nicely:

You have to write data transfer objects—A data transfer object (DTO) is a dumb data object that is returned by the EJB to its caller and contains the data the presentation tier will display to the user. It is often just a copy of the data from one or more entity beans, which cannot be passed to the presentation tier because they are permanently attached to the database. Implementing the DTOs and the code that creates them is one of the most tedious aspects of implementing an EJB. (- POJOs in Action)

Entities being POJOs can be transferred between different tiers without having to resort to anti-patterns like data transfer objects.

The simplification of the persistence API leads to several other benefits such as standardization of persistence frameworks, a separable persistence API that can be used outside EJB container and better support of object-oriented features such as inheritance and polymorphism. We'll see EJB 3.0 persistence in action in Chapter 2, but now let's take a close look at some of the main features of the persistence API.

Standardized Persistence

One of the major problems with EJB 2.x Entity Beans was that OR Mapping was never standardized. EJB 2.x Entity Beans left the details of database mapping configuration to the provider. This resulted in Entity Beans that were not portable across container implementations. The EJB 2.x query mechanism, EJB-QL, had a similar unfinished feel to it. These standardization gaps have in

Licensed to John Sweitzer <admin@saolailaem.info>

effect given rise to highly divergent alternative ORM paradigms like Hibernate, Oracle TopLink and JDO.

A major goal of JPA is to close the standardization gaps left by EJB 2.x. EJB 3.0 solidifies automated persistence with JPA in three distinct ways. The first is providing a robust Object-Relational Mapping configuration set capable of handling most automated persistence complexities. Secondly, the Java Persistence Query Language (JPQL) significantly improves upon EJB-QL, standardizing divergent Object-Relational query technologies. Finally, the `EntityManager` API standardizes ORM CRUD (Create, Read, Update and Delete) operations. But standardization wasn't the only benefit of the simplified API: another great feature is that it can run outside the container.

Separable Java Persistence API

As we touched on in section 1.2.3, API. Persistence isn't just a solution for server-side applications. Persistence is problem that even a standalone Swing based desktop application has to solve. This is the realization that drove the decision to make JPA a cleanly separated API on its own right that can be run outside an EJB 3.0 container. Much like JDBC, JPA is intended to be a general-purpose persistence solution for any Java application. This is a remarkably positive step in expanding the scope of EJB 3.0 outside the traditional realm of server applications.

Better Persistence Tier OO Support

Because EJB 2.x Entity Beans were record-oriented they did not support rich OO features like inheritance and polymorphism, as well as not allowing the mixing of persistent state and domain logic. As we saw in section 1.1.3, this made it impossible to model the domain layer in Domain Driven Design architecture.

EJB 3.0 Entities have robust OO support, not just because they are POJOs but also because the JPA ORM mapping scheme is designed with OO in mind. JPQL has robust support for OO too. Getting impatient to learn more about JPA? Hang on with us and we have many discussions on JPA throughout the book and we have Part-3 of the book is reserved with discussions on JPA.

Test Driven Development has become quite popular because it can dramatically improve performance of software applications and next we will see how EJB 3.0 improves testability of applications.

1.4.6 Unit Testable POJO Components

Being able to unit test component state or logic in response to simulated input is a critical technique in increasing code quality. In EJB 2.x, only functional testing of components was possible since components had to be deployed to the container to be executed. While functional testing simulating user interactions with the system is invaluable, it is not a good substitute for lower level unit testing.

Because all EJB 3.0 components are POJOs, they can easily be executed outside the container. This means that it is possible to unit test all component business logic using testing frameworks such as JUnit or TestNG.

These are just the primary changes to EJB 3.0. We mentioned some of the others in the services section like interceptors and transparent Web Services support. There is many more that we will cover throughout the book.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Just in case you thought you had to choose between Spring and EJB 3.0, we thought we'd mention why they don't necessarily need to be regarded as competing technologies.

1.4.7 EJB 3.0 and Spring

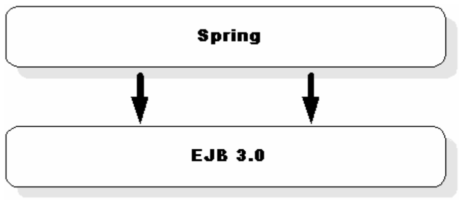
As we mentioned before Ejb3 and Spring are often seen as competitors; however, it we look a little closer, we can see that they can also be complementary Spring has some particularly strong points: support for Inversion of Control (IoC) for components with simple life cycles such as singletons, feature-heavy (but slightly more complex) Aspect Oriented Programming (AOP) support, a number of simple interfaces such as `JDBCTemplate` and `JMSTemplate` utilizing common usage patterns of low-level Java EE APIs and so on.

EJB 3.0, on the other hand, provides better support for transparent state management with Stateful Session Beans, pooling, thread-safety, robust messaging support with Message Driven Beans, integrated support for distributed transaction management, standardized automated persistence through JPA, and so on.

In fact, from a levelheaded, neutral point of view, EJB 3.0 and Spring can be complementary technologies. The good news is that parts of both the Spring and Java EE communities are working hard to make Spring/EJB 3.0 integration a reality. This is particularly good news if you have a significant investment in Spring but want to utilize the benefits of EJB 3.0. We will talk about Spring/EJB 3.0 integration in much more detail in Chapter 16. However, we will give you a "teaser" of the possibilities now.

Treat EJB 3.0 business tier components as Spring beans

It is possible to treat EJB 3.0 business tier components as Spring beans. This translates into an architecture depicted in Figure 1.5. In this architecture, Spring is used for gluing together the application that contains EJB 3.0 business tier components.



11 **Figure 1.5: A Spring/EJB 3.0 integration strategy. It is possible to use EJB 3.0 business tier components as if they were Spring beans. This allows you to use the complementary strengths of both technologies in a "hybrid" fashion.**

The Spring Pitchfork project, part of Spring 2.0, is meant to make such an integration scenario completely transparent. Interface21, the company that [[JC: Debu, what is their relation to Spring?]], plans to support EJB 3.0 annotation metadata specifying Stateless Session Beans, Interceptors, Resource injection, and so on.

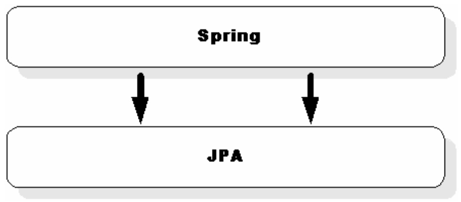
Integrate the JPA into Spring

Alternatively, it is possible that Spring is a good fit for your business tier needs and you simply want to standardize your persistence layer. In such a case, it is very easy to integrate JPA directly into

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Spring, much like Spring/Hibernate or Spring/JDO integration. This scheme is depicted in Figure 1.6.



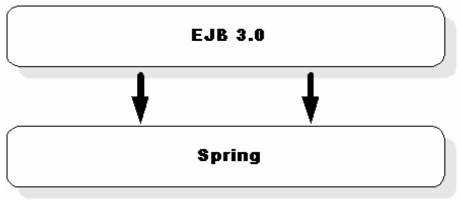
12 **Figure 1.6: Spring/JPA integration. Because JPA is a cleanly separable API, you can integrate Spring with JPA just as you would integrate Hibernate.**

13

Besides using Spring with JPA, you may find in a situation where you would like to use both Spring and EJB3 Session beans together and next we will examine the possibilities such integration.

Use Spring interfaces inside EJB 3.0 components

Yet another very interesting idea is to use some of the Spring interfaces like `JDBCTemplate` and `JMSTemplate` or even Spring beans inside EJB 3.0 components. You can do this today either through direct instantiation or access through the Spring application context. Container vendors like JBoss, Oracle and BEA are working to provide seamless support for integrating Spring beans into Session Beans and Message Driven Beans. This kind of integration is visualized in Figure 1.7.



14 **Figure 1.7: In certain cases, it might be a very good idea to use Spring from EJB 3.0. Although it is possible to do so today, such support is very likely to be much better in the future.**

We will discuss combining the power of EJB 3.0 and Spring in Chapter 16.

In this section we outlined what makes EJB 3.0 different than its predecessor and how EJB3 makes development and testing of applications simpler. EJB3 components are simple POJOs and metadata annotations are used instead of deployment descriptor. Dependency Injection instead of complex JNDI tree makes use of EJB components simpler. EJB3 JPA simplifies persistence aspect of EJB replacing EJB2 entity beans that was criticized for its limitations. EJB3 being POJOs are simple for following test driven development approach. EJB3 being POJOs have been embraced by lightweight containers and provide interesting integration possibilities with frameworks such as Spring. You will see all these in action in rest of the book starting with next chapter.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

1.5 Summary

You should now have a pretty good idea of what EJB 3.0 is, what it brings to the table, and why you should consider using it to build server-side applications. We gave you an overview of the new features in EJB 3.0, including:

EJB 3.0 components are POJOs configurable through simplified metadata annotations.

Accessing EJBs from client applications has become very simple using dependency injection.

EJB 3.0 standardizes the persistence with the Java Persistence API, which defines POJO Entities that can be used both inside and outside the container.

We also provided a taste of code to show how EJB3 addresses development pain points that was inherent with EJB 2.x and took a brief look at how EJB 3.0 can be used with Spring.
Armed with this essential background, at this point you are probably eager to look at some code. We aim to satisfy this desire, at least in part, in the next Chapter. In Chapter 2, we are going to take a whirlwind tour of the entire EJB 3.0 API to show you how easy the code really is.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 2 A First taste of EJB

In this age of hyper-competitiveness, learning a new technology by balancing a book on your lap while hacking away at a business problem on the keyboard has become norm. Let's face it—somewhere deep down, you probably prefer this "baptism by fire" to trudging the same old roads over and over again. This Chapter is for the brave pioneer in all of us, eager to get a peek over the horizon into the new world of EJB 3.0.

The first Chapter gave you a 20,000-foot view of the EJB 3.0 landscape on board a hypersonic jet. We told you what EJB is, what services it offers, what the EJB 3.0 architectural vision is and what the different parts of EJB 3.0 are. This Chapter is a low-altitude fly-over with a reconnaissance airplane. In this Chapter, we will take a quick and dirty look at the code for solving a realistic problem using EJB 3.0. The example solution will use all of the EJB 3.0 components types, a layered architecture and some of the services we talked about in Chapter 1.

EJB 3.0 offers a pretty wide range of features and services. To keep things sane, the examples in this chapter are designed to show you the very high level features and services of EJB 3.0, and to introduce you to the major players in EJB: the beans and clients. Thanks to the almost invisible way most EJB 3.0 services are delivered, this is pretty easy to do. You will see exactly how easy and useful EJB 3.0 is and how quickly you could pick it up.

We start by covering some basic concepts necessary for understanding the examples, and then we introduce the application that runs throughout the book, the Action Bazaar. In the rest of the chapter, we illustrate each EJB type with an example from the ActionBazaar application. We will implement business logic with session beans and then we will add the power of asynchronous messaging by adding an MDB. Finally we will discover the most powerful innovation of EJB 3.0 by looking at a simple example of JPA Entity.

If you aren't really a big fan of views from heights, don't worry too much. Think of this Chapter as that first day at a new workplace, shaking hands with the stranger from the next cubicle. In the Chapters that follow, you will get to know more about your new co-workers' likes, dislikes, eccentricities and how to work around these foibles. All you are expected to do right now is put names to the faces.

Note

In the examples in this chapter, we won't explore the solutions beyond what is necessary for discussing the EJB 3.0 component types but will leave some of it for you as a brainteaser. If you want to, you can peek at the entire solution by downloading the Chapter2.zip file from http://ejb3inaction.com. In fact, we highly recommend that you follow the tutorial on the site to set up your development environment using the code. This way, you can follow along with us and even tinker with the code on your own, including running it inside a container.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

EJB 3.0 is a fundamental paradigm shift from previous versions. There are a number of innovations, some familiar and some unfamiliar, that make this paradigm shift possible. A very good point to start this Chapter is by exploring three of the most important of these innovations.

2.1 New features: simplifying EJB

There are three primary sources of complexities in EJB 2.x: the heavyweight programming model, directly using the Java Naming Directory Interface (JNDI) and a verbose XML deployment descriptor. Three primary techniques in EJB 3.0 eliminate these sources of complexity: metadata annotations, minimal deployment descriptors and dependency injection. In the following Sections, we will introduce all three of these major innovations that make developing EJB 3.0 as quick and as easy as possible. Let's begin by looking at how annotations and deployment descriptors work.

2.1.1 Replacing deployment descriptors with annotations

Service configuration using Java metadata annotations is easily the most important change in EJB 3.0. As you will see throughout the book, annotations make the EJB programming model simple, remove the need for detailed deployment descriptors and act as an effective delivery mechanism for dependency injection.

In the next few years, it is very likely that annotations will play a greater and greater role in improving Java SE and Java EE usability by leaps and bounds. In case you are not familiar with the metadata annotation facility added in Java SE 5.0, let's review it first.

Java Metadata Annotations: a brief primer

Annotations essentially allow us to "attach" additional information (officially called "attributes") to a Java class, interface, method or variable. The additional information conveyed by annotations can be used by a development environment like Eclipse, the Java compiler, a deployment tool, a persistence provider like Hibernate or a runtime environment like the Java EE container. Another way to think about annotations is that they are "custom" Java modifiers (in addition to `private`, `public`, `static`, `final` and so on) that can be used by anything handling Java source or byte code. This is how annotations look like:

```
import mypackage.Author;

@Author("Debu Panda, Reza Rahman and Derek Lane")
public class EJB3InAction implements ManningBook
```

The `@Author` symbol is the annotation. It essentially tells whoever is using the `EJB3InAction` Java class that the authors are Debu Panda, Reza Rahman and Derek Lane. More interestingly, it adds this bit of extra information about the class without forcing us to implement an interface, extend a class or add a member variable or method. Since annotations are really a special kind of interface, it must be imported from where it is defined. In our case, the `@Author` annotation is defined in the `mypackage.Author.class` file. This is all there is to making the complier happy. It is up to the runtime environment how the `@Author` annotation should be used. As an example, it could be used by the Manning website engine to display the author names for this book.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Like many of the Java EE 5.0 innovations, annotations have humble beginnings. The '@' character is a dead giveaway to the grandparent of annotations–Javadoc tags. The next step in the evolution of the annotation from the lumbering caveman Javadoc tag was the XDoclet tool. If you have done a significant amount of work with EJB 2.x, it is very likely you are already familiar with XDoclet. XDoclet acted as a source code preprocessor that allowed to you to process custom Javadoc tags and do whatever you needed to do with the tagged source code, such as generate PDF documentation, additional source code or even EJB 2.x deployment descriptors. XDoclet named this paradigm "attribute-oriented programming". In case you are curious, you can find out more about XDoclet at http://xdoclet.sourceforge.net/xdoclet/index.html.

The sleek new annotation facility essentially makes attribute oriented programming a core part of the Java language. Although this is entirely possible, it is probably unlikely you will be creating your own annotations. If your inner geek just won't leave you alone, feel free to explore *Making the Most of Java's Metadata* (http://www.oracle.com/technology/pub/articles/hunter_meta.html) by Jason Hunter. You can find out more about annotations in general at: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

Note, just like anything else, annotations and attribute oriented programming have a few weaknesses. Specifically, it isn't always a good idea to mix and match configuration with source code such as annotations. This means that you would have to change source code each time you made a configuration change to something like a database connection resource or deployment environment entry. EJB 3.0 solves this problem by allowing you to override annotations with XML deployment descriptors where appropriate.

Know Your Deployment Descriptor

A deployment descriptor is simply an XML file that contains application configuration information. Every deployment unit in Java EE can have a deployment descriptor that describes its contents and environment. Some typical examples of deployment units are the Enterprise Archive (EAR), Web Application Archive (WAR) and the EJB (ejb-jar) module. If you have ever used EJB 2.x, you know how verbose the XML (`ejb-jar.xml`) descriptor was. Most elements were required even if they were trivial. This added to the complexity of using EJB. For example, you could have had the following deployment descriptor for the `HelloUserBean` that we saw in Chapter 1:

```
<enterprise-beans>
    <session>
        <ejb-name>HelloUserBean</ejb-name>
        <local>ejb3inaction.example.Hello</local>
        <ejb-class>ejb3inaction.example.HelloUserBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>
```

We will discuss deployment descriptors in greater detail when we talk about EJB packaging in Chapter 11. The good news is that EJB 3.0 makes deployment descriptors completely optional. You can now use metadata annotations instead of descriptor entries, making the development experience much simpler. Note that we will primarily use annotations throughout this book. This is not because we think deployment descriptors are unimportant or outdated, but because concepts are more easily

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

explained using annotations. As matter of fact, although deployment descriptors involve dealing with often confusing and verbose XML, we think they can be an excellent mechanism for separating coding concerns from deployment and configuration concerns. With this fact in mind, we present the deployment descriptor counterparts for each of the annotations described in the chapter (and more) in the appendix.

You can use deployment descriptor entries only for corner cases where you need them.

Mixing annotations and deployment descriptors

Annotations and descriptors are not mutually exclusive. In fact, in EJB 3.0 they are designed for harmonious co-existence. Deployment descriptor entries override configuration values hard-coded into EJB components. As an example, we could override the `@Author` annotation we just introduced with the following imaginary deployment descriptor:

```
<ManningBooks>
    <ManningBook>
       <BookClass>EJB3InAction</BookClass>
       <Author>Larry, Moe and Curly</Author>
    </ManningBook>
</ManningBooks>
```

At runtime, the Manning website engine would detect that the authors of the `EJB3InAction` book really are Larry, Moe and Curly and not Debu Panda, Reza Rahman and Derek Lane.

This is an invaluable feature if you develop enterprise applications that can be deployed to a variety of environments. In the simplest case, the differing environments could be a test and a production server. In the most complex case, you could be selling shrink-wrapped enterprise applications deployed to an unknown customer environment. The most obvious way of mixing and matching annotation and XML metadata is to use XML for deployment environment specific configuration while using annotations for everything else. If you really don't like annotations, that's fine too. You can avoid using them completely in favor of XML deployment descriptors. We will primarily focus on annotations rather than deployment descriptors in this book simply because they are so much more intuitive to look at and explain.

Common metadata annotations

Obviously, EJB defines its own set of standard annotations. We will be discussing these annotations throughout this book.

During the course of developing Java EE 5.0, it became apparent that the Java EE container as a whole could use some of the annotations geared toward EJB 3.0. In particular, these annotations are extremely useful in integrating EJB with the web/Servlet tier. Some of these annotations were separated out of the EJB 3.0 spec and christened "Common Metadata Annotations". These annotations are a core part of what makes EJB 3.0 development easy, including dependency injection. Table 2.1, lists some of the major common metadata annotations. We will discuss them throughout this part of the book, starting with some of the most fundamental ones in this Chapter.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

1.3 Table 2.1: Major Metadata annotations introduced in Java EE. Although primarily geared toward EJB, these annotations apply to Java EE components such as Servlets and JSFs managed beans as well as application clients.

Annotations	**Usage**	**Components that can use**
javax.annotation.Resource	Dependency Injection of resources such as DataSource, JMS objects, etc.	EJB, Web, Application Client
javax.ejb.EJB	Dependency Injection of Session beans	EJB, Web, Application Client
javax.jws.WebServiceRef	Dependency Injection of Web services	EJB, Web, Application Client
javax.persistence.PersistenceContext	Dependency Injection of container-managed EntityManager	EJB, Web
javax.persistence.PersistenceUnit	Dependency Injection of EntityManagerFactory	EJB, Web
javax.annotation.PostConstruct	Lifecycle method	EJB, Web
javax.annotation.PreDestroy	Lifecycle method	EJB, Web
javax.annotation.security.RunAs	Security	EJB, Web
javax.annotation.security.RolesAllowed	Security	EJB
javax.annotation.security.PermitAll	Security	EJB
javax.annotation.security.DenyAll	Security	EJB
javax.annotation.security.DeclareRoles	Security	EJB, Web

As you can see, dependency injection is front and center of the common metadata annotations, including the `@Resource`, `@EJB`, `@WebServiceRef`, `@PersistenceContext` and `@PersistenceUnit` annotations. Just as metadata annotations take away the ugliness of descriptors away from the developer's view, dependency injection solves the complexities surrounding manual JNDI lookups. Let's take a look at this concept next.

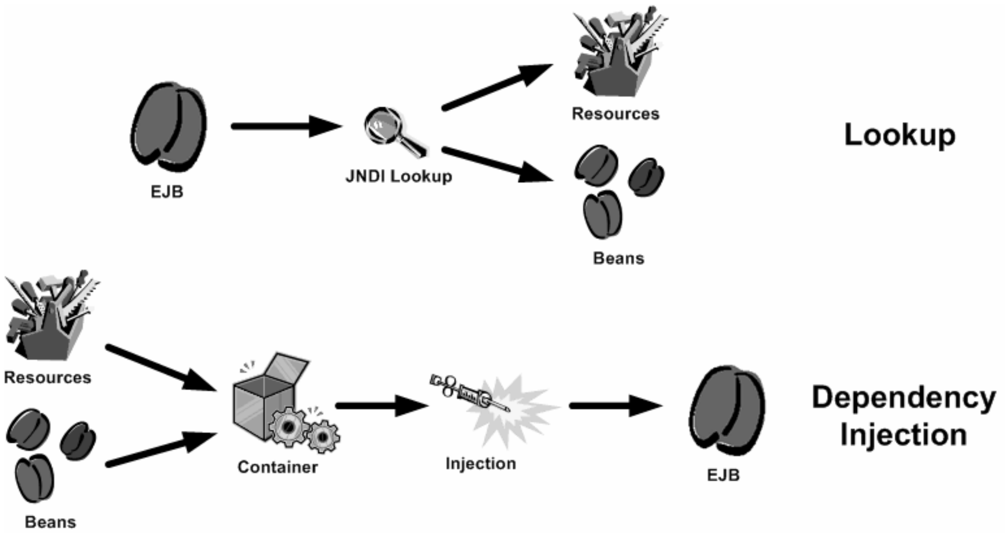
2.1.2 Introducing Dependency Injection

Almost every component uses another component or resource to implement functionality. The primary goal of Dependency Injection (DI) is to make component interdependencies as loosely coupled as possible. In real terms, this means that one component should only call another component or resource through an interface and components and resources should be glued together using configuration instead of code. As a result, component implementations can easily be swapped out as necessary simply by reconfiguring the application.

If you have used JNDI extensively in EJB 2.x, you will appreciate how much this means. We won't talk about JNDI very much here since in most cases you can get away without knowing anything about it. If you don't know about JNDI and are curious to learn more, we discuss it in some length in Appendix A. Figure 2.1 shows the difference between manual JNDI lookups and DI.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



15 **Figure 2.1: When using JNDI it's the responsibility to do a lookup and obtain a reference to the object. In EJB 3.0, you may think Dependency Injection is inverse of JNDI. It is responsibility of container to inject an object based on the dependency declaration**

In a sense, injection is lookup *reversed*. As you can see, in the manual JNDI lookup model, the bean explicitly retrieves resources and components it needs. As a result, component and resource names are hard-coded in the bean. With DI on the other hand, the container reads target bean configuration, figures out what beans and resources the target bean needs and injects them into the bean at runtime. In the end, you write no lookup code and can easily change configuration to swap out beans and resources as needed.

In essence, DI allows us to declare component dependencies, and lets the container deal with the complexities of service or resource instantiation, initialization, sequencing and supplies the service or resource references to the clients as required. As we work our way through the examples in this chapter, you will see several places where we use DI, including `@EJB` to inject EJBs in section 2.3. `@Resource` to inject JMS resources in section 2.4, and `@PersistenceContext` to inject container managed persistence in section 2.5,.

To learn more about DI:

Lightweight application containers like the Spring Framework and PicoContainer popularized the idea of DI. To learn more about the roots of DI itself, visit http://www.martinfowler.com/articles/injection.html. The article by Martin Fowler faithfully examines the pros and cons of DI over JNDI-style manual lookups. Since the article was written before EJB 3.0 was conceived, you might find the discussion of EJB 2.x cool too!

Now that we've covered some of the most fundamental concepts of EJB 3.0, it is time to warm up to code. The problem we will solve in this Chapter utilizes an essential element of this book– *ActionBazaar*. ActionBazaar is an imaginary enterprise system that we will weave most of the material in this book around. In a certain sense, this book is a case study of developing the ActionBazaar application using EJB 3.0.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Let's take a quick stroll around the bazaar to see what it is all about.

2.2 Introducing the ActionBazaar application

ActionBazaar is a simple online auctioning system like eBay. Sellers dust off the treasures hidden away in basement corners, take a few out-of-focus pictures, and post their item listings on ActionBazaar. Eager buyers get in the good-old competitive spirit and put exorbitant bids against each other on the hidden treasures with the blurry pictures and misspelled descriptions. Winning bidders pay for the items. Sellers ship sold items. Everyone is happy, or so the story goes.

As much as we would like to take credit for it, the idea of ActionBazaar was first introduced in *Hibernate in Action* by Christian Bauer and Gavin King as the *CaveatEmptor* application. The *Hibernate in Action* book primary dealt with developing the persistence layer using the Hibernate ORM framework. The idea was later used by *Webworks in Action* to discuss the Open Source presentation-tier framework. We thought this was a pretty good idea to adopt for EJB 3.0.

The next two Parts of the book roughly follow the course of developing each layer of the ActionBazaar application as it relates to EJB 3.0. We will use EJB 3.0 to develop the business logic tier in Part 2 of this book, and then the persistence tier in Part 3. We will deal with the presentation layer as necessary as well.

This section will introduce you to the ActionBazaar application. We start with a subset of the architecture of ActionBazaar and then we will design a solution based on EJB 3.0 and JPA. After this section, the rest of the chapter will explore some of the important features of EJB 3.0, using examples from the ActionBazaar application to introduce you to the different bean types and show how they are used.

Let's begin by taking a look at the requirements and design of our example.

2.2.1 Starting with the architecture

For the purposes of introducing all three EJB 3.0 component types across the business logic and persistence layers, we will focus on a small subset of ActionBazaar functionality in this Chapter—starting from bidding on an item and ending in ordering the item won. This set of application functionality is shown in Figure 2.2.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

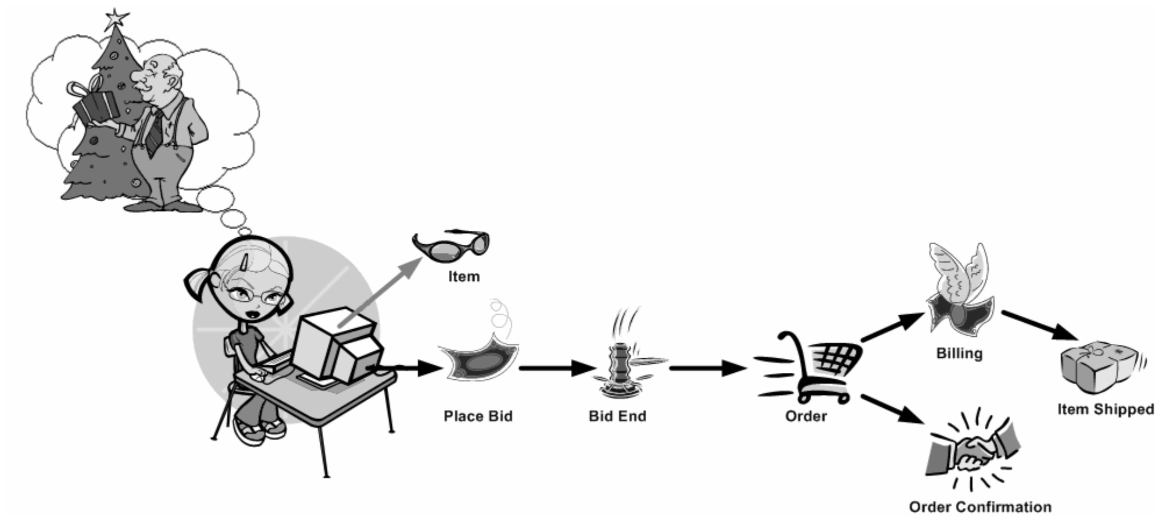


Figure 2.2: A chain of representative ActionBazaar functionality used to quickly examine a cross-section of EJB 3.0. The bidder bids on a desired item, wins the item, orders it, and instantaneously receives confirmation. In parallel to order confirmation, the user is billed for the item. Upon successful receipt of payment, the seller ships the item.

In a sense, the functionality represented in Figure 2.2 is the "essentials" of ActionBazaar. The major functionalities not covered are posting an item for sale, browsing items and searching for items. We will save these pieces of functionality for the Chapters in Part 2 and 3. This includes presenting the entire domain model, which we will save for Chapter 7, when we start talking about domain modeling and persistence using JPA.

The chain of actions in Figure 2.2 starts with the user deciding to place a bid on an item. Our user, Jenny, spots the perfect Christmas gift for grandpa and quickly puts down a starting bid of $5.00. After the timed auction ends, the highest bidder wins the item. Jenny gets lucky and no one else bids on the item, so she wins it for the grand sum of $5.00. As the winning bidder, Jenny is allowed to order the item from the seller, Joe. An order has all the things we've come to expect from online merchants—shipping information, billing details, a total bill with calculated shipping and handling costs an so on. Persuasive Jenny gets mom to foot the bill with her credit card and gets the order to be shipped directly to grandpa's address. Not unlike many e-Businesses such as Amazon.com and eBay, ActionBazaar does not make the user wait for the billing process to finish before confirming an order. Instead, the order is confirmed as soon as it is reasonably validated and the billing process is started in parallel in the background. Jenny gets an order confirmation number back as soon as she clicks the 'order' button. Although Jenny doesn't realize it, the process to charge mom's credit card starts in the background as she gets the confirmation back. After the billing process is finished, both the Jenny and the seller, Joe, are sent email notifications. Having been notified of the receipt of the money for the order, Joe ships the item, just in time for grandpa to get it before Christmas!

In the next Section, we will see how the business logic and persistence components for this set of actions can be implemented using EJB 3.0. Before peeking at the solution diagram in the next Section, you should try to visualize how the components might look like with respect to an EJB-based layered architecture. How do you think Session Beans, Message Driven Beans, Entities and the JPA API fit into the picture, given our discussion? Chances are, with the probable exception of the messaging components, your design will closely match ours.

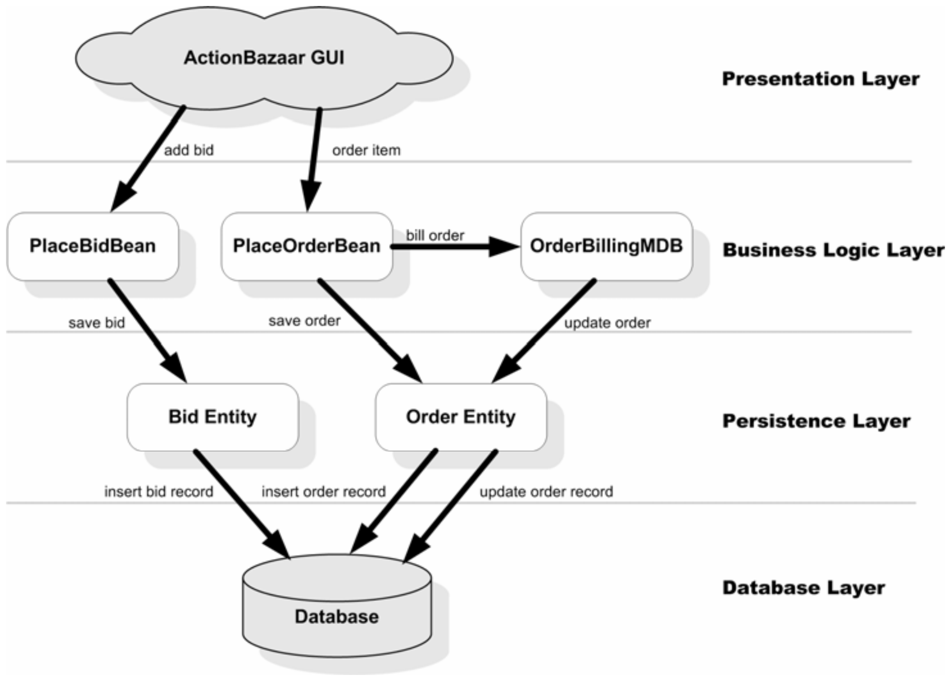
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

2.2.2 An EJB 3.0 based solution

Figure 2.2 shows how the ActionBazaar scenario in the previous Section can be implemented using EJB 3.0 in a traditional four-tier layering scheme. For our purposes, the presentation tier is essentially an amorphous blob that generates business tier requests in response to user actions. If you examine the scenario in Figure 2.2, there are really only two processes that are triggered by the user—adding a bid to an item and ordering items won. Thinking a little harder, one more process might be apparent, the background billing process to charge the order, triggered by order confirmation. If you guessed that the billing process is triggered through a message, you guessed right. As you can see in Figure 2.3, the bidding and ordering processes are implemented as session beans (`PlaceBidBean` and `PlaceOrderBean`) in the business logic tier. On the other hand, the billing process is implemented as a Message Driven Bean (`OrderBillingMDB`) since it is triggered by a message sent from the `PlaceOrderBean` instead of a direct user request.

All three of the processes persist data. The `PlaceBidBean` needs to add a bid record to the database. Similarly, the `PlaceOrderBean` must add an order record. Alternatively, the `OrderBillingMDB` updates the order record to reflect the results of the billing process. These database changes are performed through two Entities in the JPA-managed persistence tier—the `Bid` and `Order` Entities. While the `PlaceBidBean` uses the `Bid` Entity, the `PlaceOrderBean` and `OrderBillingMDB` use the `Order` Entity.



17 **Figure 2.3: The ActionBazaar scenario implemented using EJB 3.0. From the EJB 3.0 perspective, the presentation layer is an amorphous blob that generates business tier requests. The business logic tier components match up wit h the distinct processes in the scenario—putting a bid on an item, ordering the item won and billing the user. The billing MDB is triggered by a message sent by the order confirmation process. The business tier components use JPA Entities to persist application state into the database.**

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Recall that although JPA Entities contain ORM configuration, they do not persist themselves. As we will see in the actual code solutions, the business tier components have to use the JPA `EntityManager` API to add, delete, update and retrieve Entities as needed.

If your mental picture matches up with Figure 2.3 pretty closely, it is likely the code we are going to present next will seem pretty intuitive too, even though you don't know EJB 3.0.

In the next Sections, we will explore each of the EJB 3.0 component types using our scenario as a crutch. Without further ado, we can now start our whirlwind tour of EJB 3.0 component types, starting with the Session Beans in the business logic tier.

2.3 Building business logic with Session Beans

Session Beans are meant to model business processes or actions, especially as perceived by the system user. This is why are ideal for modeling the bidding and ordering processes in our scenario. In a sense, Session Beans are the easiest but most versatile part of EJB.

Recall that Session Beans come in two flavors – Stateful and Stateless. We will take on Stateless Session Beans first mostly because they are simpler. We will then discover how you can add statefulness to ActionBazaar application by using a stateful session bean. Along the way, we'll introduce you to an example of a Session Bean client in a web tier, and then build a standalone Java client for a session bean.

2.3.1 Using Stateless Beans

Stateless Session Beans are used to model actions or processes that can be done "in-one-shot", like placing a bid on an item in our ActionBazaar scenario. The `addBid` Bean method in Listing 2.1 is called from the ActionBazaar web tier when a user decides to place a bid. The parameter to the method, the `Bid` Object, represents the bid to be placed. The `Bid` Object contains the ID of the bidder placing the bid, the ID of the item being bid on, and the bid amount. As we know, all the method really needs to do is save the passed-in `Bid` data to the database. In a real application, you would see more validation and error handling code in the `addBid` method. Since the point is to show you how a Session Bean looks like and not to demonstrate the Über geek principles of right and proper enterprise development, we've conveniently decided to be slackers. Also, as you will see towards the end of the Chapter, the `Bid` Object is really a JPA Entity.

2 Listing 2.1: PlaceBid Stateless Session Bean code

```
package ejb3inaction.example.buslogic;

import javax.ejb.Stateless;
import ejb3inaction.example.persistence.Bid;

@Stateless                                              |#1
public class PlaceBidBean implements PlaceBid {
    ...
    public PlaceBidBean() {}

    public Bid addBid(Bid bid) {
        System.out.println("Adding bid, bidder ID=" + bid.getBidderID()
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
                + ", item ID=" + bid.getItemID() + ", bid amount="
                    + bid.getBidAmount() + ".");

           return save(bid);
       }
       ...
   }
   ...
package ejb3inaction.example.buslogic;

import javax.ejb.Local;
import ejb3inaction.example.persistence.Bid;

@Local                                                        |#2
public interface PlaceBid {
    Bid addBid(Bid bid);
}
```

(annotation) <#1 Mark POJO as Stateless Session Bean>
(annotation) <#2 Mark EJB interface as local>

The first thing that you have probably noticed is how plain this code looks like. The `PlaceBidBean` class is just a Plain Old Java Object (POJO) and the `PlaceBid` interface is a Plain Old Java Interface (POJI). There is no cryptic EJB interface to implement, class to extend, or confusing naming convention to follow. In fact, the only notable features in Listing 2.1 are the two EJB 3.0 annotations, `@Stateless` and `@Local`.

@Stateless

The `@Stateless` annotation tells the EJB container that `PlaceBidBean` is a Stateless Session Bean. This means that the container automatically provides a bunch of services to the bean like automatic concurrency control, thread-safety, pooling, and transaction management.

In addition, you can add other services that Stateless Beans are eligible for such as transparent security and interceptors.

@Local

The `@Local` annotation on the `PlaceBid` interface tells the container that the `PlaceBid` EJB can be accessed locally through the interface. Since EJB and Servlets are typically collocated in the same application this is probably perfect. Alternatively, we could have marked the interface with the `@Remote` annotation. Remote access through the `@Remote` annotation is provided under the hood by Java RMI, so this is the ideal means of remote access from Java clients.

If the EJB needs to be accessed by non-Java clients like .NET applications, web services based remote access can be enabled using the `@WebService` annotation applied either on the interface or the bean class.

That's pretty much all we are going to say about Stateless Session Beans for now. Let's now turn our attention to the client code for using the `PlaceBid` EJB.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

2.3.2 The Stateless Bean Client

Virtually any client can use the `PlaceBid` EJB in Listing 2.1. However, the most likely scenario for EJB usage is from a Java based web tier. In the ActionBazaar scenario, the `PlaceBid` EJB is probably called from a JSP or Servlet. For simplicity, let's assume that the `PlaceBid` EJB is used by a Servlet named `PlaceBidServlet`. Listing 2.2 shows how the code might look like. The Servlet's `service` method is invoked when the user wants to place a bid. The bidder's ID, item ID and the bid amount are passed in as HTTP request parameters. The Servlet creates a new `Bid` Object, sets it up and passes it to the EJB `addBid` method.

3 Listing 2.2: A simple Servlet client for the PlaceBid EJB

```
package ejb3inaction.example.buslogic;

import javax.ejb.EJB;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import ejb3inaction.example.persistence.Bid;

public class PlaceBidServlet extends HttpServlet {
    @EJB                                         |#1
    private PlaceBid placeBid;

    public void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
            IOException {
        int bidderID = Integer.parseInt(
            request.getParameter("bidder_id"));
        int itemID = Integer.parseInt(
            request.getParameter("item_id"));
        double bidAmount = Double.parseDouble(
            request.getParameter("bid_amount"));

        Bid bid = new Bid();
        bid.setBidderID(bidderID);
        bid.setItemID(itemID);
        bid.setBidAmount(bidAmount);

        placeBid.addBid(bid);
        ...
    }
    ...
}
```

(annotation) <#1 Inject instance of PlaceBid EJB>

As you can see from Listing 2.2, EJB from the client-side looks even simpler than developing the component code. Other than the `@EJB` annotation on the `placeBid` private variable, the code is no different than using a local POJO.

@EJB

When the Servlet container sees the `@EJB` annotation as the Servlet is first loaded, it looks up the `PlaceBid` EJB behind the scenes and sets the `placeBid` variable to the retrieved EJB reference. If necessary, the container will lookup the EJB remotely over RMI.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The @EJB annotation works in any component that is registered with the Java EE container such as a Servlet or JSF backing bean. As long as you are using the standard Java EE stack, this is probably more than sufficient..

There are a couple other interesting things to look at in this code, illustrating concepts we introduced earlier. Let's take a closer look at them.

EJB 3.0 Dependency Injection

Although we mentioned DI in the beginning of the Chapter, if you are not familiar with DI, what the @EJB annotation is doing might seems a little unusual, in a nifty "black-magic" kind of way. In fact, if we didn't tell you anything about the code, you might have been wondering if the placeBid private variable is even usable in the Servlet's service method since it is never set! If fact, if the container didn't intervene, we would get the infamous java.lang.NullPointerException when we try to call the addBid method in Listing 2.2 since the placeBid variable would still be null. One interesting way to remember and understand DI is to think of it as "custom" Java variable instantiation. The @EJB annotation in Listing 2.2 makes the container "instantiate" the placeBid variable with the EJB named 'PlaceBid' before the variable is available for use.

Recall our discussion in Section 2.1.3 that DI can be viewed as inverse of JNDI lookup. Recall also that JNDI is the container registry that holds references to all container-managed resources such as EJBs. Clients get access to Session Beans like our PlaceBid EJB directly or indirectly through JNDI. In EJB 2.x, you would have to manually populate the placeBid variable using JNDI lookup code that looks like the following:

```
Context initialContext = new InitialContext();
Object ejbHome = initialContext.lookup("java:comp/env/PlaceBid");
PlaceBidHome placeBidHome = (PlaceBidHome)
    PortableRemoteObject.narrow(ejbHome, PlaceBidHome.class);
PlaceBid placeBid = placeBidHome.create();
```

It isn't easy to fully appreciate DI until you see the code above. EJB 3.0 DI using the @EJB annotation reduces all this mechanical JNDI lookup code to a single statement! In a non-trivial application, this can easily translate to getting rid of hundreds of lines of redundant, boring, error-prone code. In a sense, EJB 3.0 DI is really a high level abstraction over JNDI lookups.

Understanding Statelessness

An interesting thing about the PlaceBid Stateless Bean is that as long as calling the addBid method results in a new bid record being created each time, the client does not care about the internal state of the bean. There is absolutely no need for the Stateless Bean to guarantee that the value of any of its instance variables will be the same across any two invocations. This property is what statelessness means in terms of server-side programming.

The PlaceBid Session Bean can afford to be stateless because the action of placing a bid is simple enough to be accomplished in a single step. The problem is that not all business processes are that simple. Breaking a process down into multiple steps and maintaining internal state to "glue together" the steps is a common technique to present complex processes to the user in a simple way. Statefulness is particularly useful if what the user does in a given step in a process determines what the next step is. Think of a questionnaire based setup wizard. The user's input for each step of the wizard

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

is stored behind the scenes and is used to determine what to ask the user next. Stateful Session Beans make maintaining server-side application state as easy as possible.

2.3.3 Using Stateful Beans

Unlike Stateless Session Beans, Stateful Session Beans guarantee that a client can expect to set the internal state of a bean and count on the state being maintained between any numbers of method calls. The container makes sure this happens by doing two important things behind the scenes.

Maintaining the session

The first is to make sure that a client can reach a bean dedicated to it across more than one method invocation. Think of this as a phone switchboard that makes sure it routes you to the same customer service agent if you call a technical support line more than once in a given period of time (the period of time is the "session").

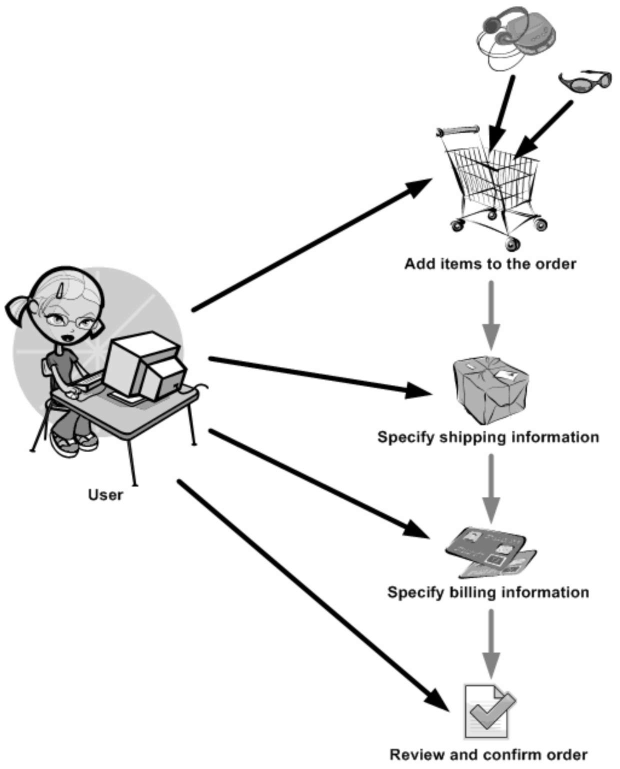
The second thing that the container does is make sure that bean instance variable values are maintained for the duration of a session without our having to write any session maintenance code. In terms of the customer service example, this makes sure that your account information and call history in a given period of time is automatically popped up on your agent's screen when you call technical support. This "automagic" maintenance of session state is a huge leap from having to fiddle with the HTTP session, browser cookies or hidden HTML form variables to try to accomplish the same thing. As we will see in the coming code samples, you can develop Stateful Beans as if you are developing a in a 'Hello World' application, not a web application with verbose code to maintain session state. The ActionBazaar ordering process is a great example for Stateful Session Beans since it is broken up into four steps, each of which correspond to a screen presented to the user. These are the steps:

1. Adding items to the order. If the user started the ordering process by clicking the "order item" button on the page displaying an item won, the item is automatically added to the order. The user can still add additional items in this step.
2. Specifying shipping information including shipping method, shipping address, insurance, and so on.
3. Adding billing information such as credit card data, billing address and the like.
4. Confirming the order after reviewing the complete order, including total cost.

Figure 2.4 depicts these ordering steps. Using a Stateful Bean, the data the user enters at each step could be cached into bean variables until the ordering workflow completes when the use confirms the order.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



18 **Figure 2.4: To make an otherwise overwhelming process manageable, the ActionBazaar ordering process is broken down into several steps. The first of these steps is to add one or more item to the order. The second step is to specify shipping information for the order. The third is to specify the billing information. Reviewing and confirming the order finishes the ordering process.**

Now they we know what we want, let's see how we can implement it.

Implementing the solution

Listing 2.3 shows a possible implementation of the ActionBazaar ordering workflow using a bean named `PlaceOrderBean`. As you might be able to see, each of the ordering steps maps to a method in the `PlaceOrderBean` implementation. The `addItem`, `setShippingInfo`, `setBillingInfo` and `confirmOrder` methods are called in sequence from the web-tier in response to user actions in each step. The `setBidderID` method essentially represents an implicit workflow setup step. It is called at the beginning of the workflow behind the scenes by the web application to identify the currently logged-in user as the bidder placing the order. Other than the `confirmOrder` method, the rest of the methods do little else than simple save off user input into stateful instance variables. In a real application, of course, these methods would be doing a lot more like error handling, validation, figuring out the user's options for a given step, calculating costs, and so on. The `confirmOrder` method does a bunch of things using the data accumulated throughout the session – the complete order is saved into the database, the billing process is started in parallel, and an order ID is returned back to the user as confirmation.

4 Listing 2.3: PlaceOrderBean Stateful Session Bean

```
package ejb3inaction.example.buslogic;

import javax.ejb.*;
import java.util.ArrayList;
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
import java.util.List;

@Stateful                                                              |#1
public class PlaceOrderBean implements PlaceOrder {
    private Long bidderID;                                             |#2
    private List<Long> items;                                          |#2
    private ShippingInfo shippingInfo;                                 |#2
    private BillingInfo billingInfo;                                   |#2

    public PlaceOrderBean () {
        items = new ArrayList<Long>();
    }

    public void setBidderID(Long bidderId) {
        this.bidderId = bidderId;
    }

    public void addItem(Long itemId) {
        items.add(itemId);
    }

    public void setShippingInfo(ShippingInfo shippingInfo) {
        this.shippingInfo = shippingInfo;
    }

    public void setBillingInfo(BillingInfo billingInfo) {
        this.billingInfo = billingInfo;
    }

    @Remove                                                            |#3
    public Long confirmOrder() {
        Order order = new Order();
        order.setBidderId(bidderId);
        order.setItems(items);
        order.setShippingInfo(shippingInfo);
        order.setBillingInfo(billingInfo);

        saveOrder(order);
        billOrder(order);

        return order.getOrderId();
    }
    ...
}
...
package ejb3inaction.example.buslogic;
import javax.ejb.Remote;
@Remote                                                                |#4
public interface PlaceOrder {
    void setBidderId(Long bidderId);
    void addItem(Long itemId);
    void setShippingInfo(ShippingInfo shippingInfo);
    void setBillingInfo(BillingInfo billingInfo);
    Long confirmOrder();
}
```
(annotation) <#1 Mark bean as Stateful>
(annotation) <#2 Stateful instance variables>
(annotation) <#3 Remove method>
(annotation) <#4 Remote business interface>

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

As you can see, overall, there is not really a big difference between developing a Stateless and a Stateful Bean. In fact, from a developer's perspective, the only difference is that the `PlaceOrderBean` Class is marked with the `@Stateful` annotation instead of the `@Stateless` annotation#1. As we know though, under the hood this makes a very large difference as to how the container handles the bean's relationship to a client and the values stored in the bean instance variables#2. The `@Stateful` annotation also serves to tell the client-side developer what to expect from the bean if behavior is not obvious from the bean's API and documentation.

It is also important to note the `@Remove` annotation placed on the `confirmOrder` method. Although this annotation is optional, it is very critical for a server performance standpoint.

@Remove

The `@Remove` annotation marks the end of the workflow modeled by a Stateful Bean. In our case, we are telling the container that there is no longer a need to maintain the bean's session with the client after the `confirmOrder` method is invoked. If we didn't tell the container what method invocation marks the end of the workflow, the container could wait for a long time until it can safely time out the session. Since Stateful Beans are guaranteed to be dedicated to a client for the duration of a session, this could mean a lot of "orphaned" state data consuming precious server resources for long periods of time!

There is virtually no difference between the bean interfaces for our Stateless and Stateful Bean examples. Both are POJI marked with the `@Remote` annotation to enable remote client access#4.

Let's now take a quick look at Stateful Beans from the client perspective. As you might expect, there really are not major semantic differences from Stateless Beans.

2.3.4 A Stateful Bean Client

It is clear that the `PlaceOrder` EJB is called from the ActionBazaar web-tier. However, to give a slightly more colorful perspective on things, we'll deliberately stay out of web-tier client examples this time. We will use a thick Java application that functions as a test script to run through the entire workflow of the `PlaceOrder` EJB using some dummy data. This test script could just as easily been part of a very high-level regression test suit using a framework like JUnit or NUnit.

A note on unit testing

If you actually have management buy-in to invest in extensive unit testing, you might also note the fact that because of the POJO centric nature of EJB 3.0, the application below could be easily be modified to be a full-scale unit test using dummy data sources and the like. We will leave this for you as an exercise in case you are interested in exploring further by tweaking the source code available for download from http://ejb3inaction.com. If unit testing and code coverage are just not viable topics to bring up in your work environment, don't worry. We understand where you are coming from and don't assume you do a ton of unit testing.

Listing 2.4 shows the code for the stateful session bean client.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

5 Listing 2.4: Stateful Session Bean Client

```
package ejb3inaction.example.buslogic;

import javax.ejb.EJB;

public class PlaceOrderTestClient {
    @EJB                                                              |#1
    private static PlaceOrder placeOrder;

    public static void main(String [] args) throws Exception {
         System.out.println("Exercising PlaceOrder EJB...");
        placeOrder.setBidderId(new Long(100));
        placeOrder.addItem(new Long(200));
        placeOrder.addItem(new Long(201));
        placeOrder.setShippingInfo(
             new ShippingInfo("123 My Sweet Home",
                        "MyCity","MyState"));
        placeOrder.setBillingInfo(
              new BillingInfo("123456789","VISA","0708"));
        Long orderId = placeOrder.confirmOrder();
         System.out.println("Order confirmation number: " + orderId);

    }
}
```

(annotation) <#1 Inject an instance of EJB>

As you can see, there is nothing special you need to do from the client-side to use Stateful Beans. As a matter of fact, there is virtually no difference in the client code between using a Stateless and a Stateful Bean, other than the fact that the client can safely assume that the EJB is maintaining state even if it is sitting on a remote application server. The other remarkable thing to note about Listing 2.4 is the fact that the `@EJB` annotation is injecting a remote EJB into a standalone client. This is accomplished by running the client in the Application Client Container (ACC).

The Application Client Container

The application client container is a mini Java EE container that can be run from the command line. Think of it as a souped-up JVM with some Java EE juice added. You can run any Java SE client like a Swing application inside the application client container as if you were using a regular old JVM. The beauty of it is that the application client container will recognize and process most Java EE annotations like the `@EJB` DI annotation. Among other things, the client container can lookup and inject EJBs on remote servers, communicate with remote EJBs using RMI, provide authentication, authorization and so on. The application client really shines if you need to use EJBs in an SE application or would like to inject real resources into your POJO during unit testing.

Any Java class with a `main` method can be run inside the ACC. Typically, thought, an application client is packaged in a jar file that must contain a Main-class in the Manifest file. Optionally the jar may contain a deployment descriptor (`application-client.xml`) and a `jndi.properties` file that contains the environment properties to connect to a remote EJB container. Let's assume you packaged up your application client classes in jar file named chapter2-client.jar file. Using Sun's Glassfish application server you could launch your application client inside ACC as follows:

```
appclient -client  chapter2-client.jar
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

This finishes our brief introduction of Session Beans using our ActionBazaar scenario. We are now ready to move on to the next business tier EJB component, Message Driven Beans (MDB).

2.4 Messaging with Message Driven Beans

Just as Session Beans process direct business requests from the client, Message Driven Beans process indirect *messages*. Messaging has numerous uses in enterprise systems, such as system integration, asynchronous processing, distributed system communication and so on. If you have been doing enterprise development for a little bit of time, you are probably familiar with at least the basic idea of messaging. In the most basic terms, messaging means communicating between two separate processes, usually across different machines. Java EE messaging follows this basic idea, just put on steroids. Most significantly, Java EE makes messaging robust by adding a reliable middleman between the message sender and receiver. This idea is illustrated in Figure 2.5.

Sender
Message
Reliable Middleman
Message
Receiver

19 **Figure 2.5: The Java EE "pony express" messaging model. Java EE adds reliability to messaging by adding a middleman that guarantees the delivery of messages despite network outages, even if the receiver is not present on the other end when the message is sent. In this sense, Java EE messaging has much more in common with the postal service than it does with common RPC protocols like RMI. We will discuss this model in much greater detail in Chapter 5.**

In Java EE terms, the reliable middleman is called a messaging *destination*, powered by Message Oriented Middleware (MOM) servers like IBM MQSeries or SonicMQ. Java EE standardizes messaging through a well-known API, JMS (Java Messaging Service), which Message Driven Beans rely heavily on.

We will discuss messaging, JMS and Message Driven Beans in much greater detail in Chapter 4. For now, this is all you really need.

In this section we will build a simple example of message producer and a message driven bean.

In our ActionBazaar example, we enable asynchronous order billing through messaging. To see how this is done, let's revisit the parts of the `PlaceOrderBean` introduced in Listing 2.3 that we deliberately left hidden, namely the implementation of the `billOrder` method.

2.4.1 Producing a Billing Message

As we discussed in our high-level solution schematic in Section 2.2, The `PlaceOrderBean` accomplishes asynchronous or "out-of-process" order billing by generating a message in the `confirmOrder` method to request that the order billing be started in parallel. A soon as this billing request message is sent to the messaging middleman, the `confirmOrder` method returns with the order confirmation to the user. We will now take a look at exactly how this piece is implemented. As we see in Listing 2.5, the billing request message is sent to a messaging destination named

Licensed to John Sweitzer <admin@saolailaem.info>

‘jms/OrderBillingQueue’. Since you have already seen most of the implementation of the `PlaceOrder` bean, we won’t repeat a lot of the code shown in Listing 2.3 here.

6 Listing 2.5: PlaceOrderBean Producing JMS Message

```
package ejb3inaction.example.buslogic;

...
import javax.annotation.Resource;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.jms.*;
...

@Stateful
public class PlaceOrderBean implements PlaceOrder {
    @Resource(name="jms/QueueConnectionFactory")                  |#1
    private ConnectionFactory connectionFactory;

    @Resource(name="jms/OrderBillingQueue")                       |#1
    private Destination billingQueue;
    ...
    @Remove
    public Long confirmOrder() {
        Order order = new Order();
        order.setBidderId(bidderId);
        order.setItems(items);
        order.setShippingInfo(shippingInfo);
        order.setBillingInfo(billingInfo);

        saveOrder(order);
        billOrder(order);

        return order.getOrderId();
    }
    ...
    private billOrder(Order order) {
        try {
            Connection connection =                                |#2
                connectionFactory.createConnection();              |#2
            Session session =                                      |#2
                connection.createSession(false,                    |#2
                    Session.AUTO_ACKNOWLEDGE);                     |#2
            MessageProducer producer =                             |#2
                session.createProducer(billingQueue);              |#2
            ObjectMessage message =                                |#3
                session.createObjectMessage();                     |#3
            message.setObject(order);                              |#3
            producer.send(message);                                |#3
            producer.close();                                      |#4
            session.close();                                       |#4
            connection.close();                                    |#4
        } catch(Exception e)  {
            e.printStackTrace();
        }
    }
}
```

(annotation) <#1 Inject JMS resources>
(annotation) <#2 JMS setup code>
(annotation) <#3 Creating and sending the message>
(annotation) <#4 Release JMS resources>

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Not surprisingly, the code to send the message in Listing 2.5 is heavily dependent on the JMS API. In fact, that is all that the code in the `billOrder` consists of. If you are familiar with JDBC, the flavor of the code in the method might seem familiar. The end result of the code is that the newly created `Order` Object is sent as a message to a JMS destination named 'jms/OrderBillingQueue'. We will not deal with the intricacies of JMS now, but will save a detailed discussion of this essential messaging API for Chapter 4. It is important to note a few things right now, though.

The first thing you should note is that two JMS resources, including the message destination, are *injected* using the `@Resource` annotation#1,instead of being looked up.

@Resource

As we stated earlier, in addition to the `@EJB` annotation, the `@Resource` annotation provides DI functionality in EJB 3.0. While the `@EJB` annotation is limited to injecting EJBs, the `@Resource` annotation is much more general purpose and can be used to inject anything that the container knows about.

In Listing 2.5 it looks up the JMS resources specified through the `name` parameter and injects it into the `connectionFactory` and `billingQueue` instance variables. The name parameter values are what JNDI knows the resources by. The second thing that is important to realize is that the `MessageProducer.send` method#3 does not wait for a receiver to receive the message on the other end. Because the messaging server guarantees that the message will be delivered to anyone interested in the message, this is just fine. In fact, this is exactly what enables the billing process to start in parallel to the ordering process, which continues on it's merry way as soon as the message is sent. You should also note how loosely coupled the ordering and billing processes are. The ordering bean doesn't even know who picks up and processes its message; it simply knows the message destination!

As we know from our solution schematic in Section 2.3, the `OrderBilllingMDB` processes the request to bill the order. It continuously listens for messages sent to the 'jms/OrderBillingQueue' messaging destination, picks up the messages from the queue, inspects the `Order` Object embedded in the message and attempts to the bill the user. We will depict this scheme in Figure 2.6 to reinforce the concept:

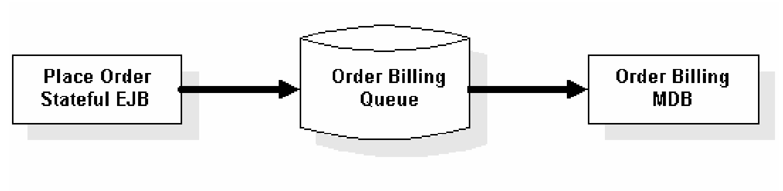


Figure 2.6: Asynchronously billing orders using MDBs. The Stateful Session Bean processing the order sends a message to the order-billing queue. The billing MDB picks up this message and processes it asynchronously.

Let's take a look now at how the `OrderBilllingMDB` is implemented.

2.4.2 Using the Order Billing Message Processor MDB

The `OrderBilllingMDB's` sole job in life is to attempt to bill the bidder for the total cost of an order, including the price of the items in the order, shipping, handling, insurance costs and the like. Listing 2.6 shows the abbreviated code for the order billing MDB. Recall that the `Order` Object passed inside the message sent by the `PlaceOrder` EJB contains a `BillingInfo` Object.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The BillingInfo Object tells the OrderBilllingMDB how to bill the customer – for example, perhaps by charging a credit card or crediting against an online bank account. However the user is supposed to be charged, after attempting to bill the user, the MDB notifies both the bidder and seller of the results of the billing attempt. If billing is successful, the seller ships to the address specified in the order. If the billing attempt fails, the bidder must correct and resubmit the billing information attached to the order. Last but not least, the MDB must also update the order record to reflect what happened during the billing attempt. Feel free to explore the complete code sample and deployment descriptor entries containing the JMS resource configuration in Chapter2.zip available for download from http://ejb3inaction.com.

7 Listing 2.6: The OrderBillingMDB

```
package ejb3inaction.example.buslogic;

import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import ejb3inaction.example.persistence.Order;
import ejb3inaction.example.persistence.OrderStatus;

@MessageDriven(                                                   |#1
    activationConfig = {                                          |#2
        @ActivationConfigProperty(                                |#2
            propertyName="destinationName",                       |#2
            propertyValue="jms/OrderBillingQueue")                |#2
    }
)
public class OrderBillingMDB implements MessageListener {         |#3
    ...
    public void onMessage(Message message) {
        try {
            ObjectMessage objectMessage = (ObjectMessage) message;
            Order order = (Order) objectMessage.getObject();

            try {
                bill(order);
                notifyBillingSuccess(order);
                order.setStatus(OrderStatus.COMPLETE);
            } catch (BillingException be) {
                notifyBillingFailure(be, order);
                order.setStatus(OrderStatus.BILLING_FAILED);
            } finally {
                update(order);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ...
}
```

(annotation) <#1 Mark as Message Driven Bean>
(annotation) <#2 Specify JMS destination to get messages from>
(annotation) <#3 Implements javax.jms.MessageListener interface>

As you might be able to notice from the code, Message Driven Beans are really Session Beans in JMS disguise. Like Stateless Beans, Message Driven Beans are not guaranteed to maintain state. The

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

`@MessageDriven` annotation is the MDB counterpart of the `@Stateless` and `@Stateful` annotations—it makes the container transparently provide messaging and other EJB services into a POJO#1. The activation configuration properties nested inside the `@MessageDriven` annotation tells the container what JMS destination the MDB wants to receive messages from.

The container and MDBs

Behind the scenes, the container takes care of a bunch of mechanical details to start listening for messages sent to the destination specified by the activation configuration properties. As soon as a message arrives at the destination, the container forwards it to an instance of the MDB.

Instead of implementing a remote or local business interface, MDBs implement the `javax.jms.MessageListener` interface. The container uses this well-known JMS interface to invoke an MDB. The `onMessage` method defined by the interface has a single `javax.jms.Message` parameter that the container uses to pass a received message to the MDB. Believe it or not, this is more or less all you need to know to get by using Message Driven Beans, as long as you have a decent understanding of messaging and JMS.

This pretty much wraps up all we need to say right now about EJB 3.0 business tier components too. As we mentioned earlier, we will devote the entirety of the next part of the book to this vital part of the EJB platform. Now, we will move on to the other major part of EJB, the Persistence API.

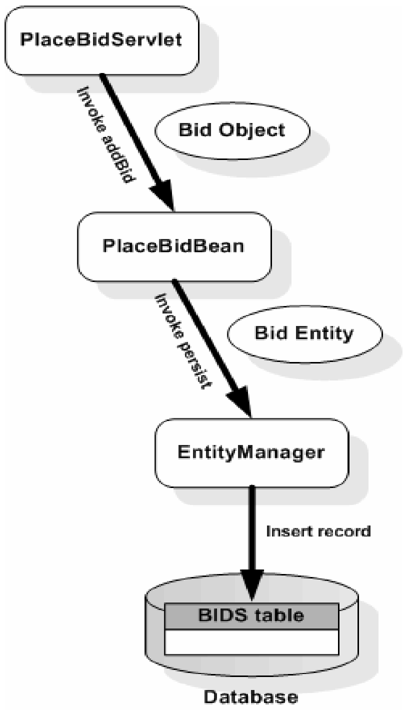
2.5 Persisting data with EJB 3.0 JPA

The Java Persistence API (JPA) is the persistence tier solution for the Java EE platform. Although a lot has changed in EJB 3.0 for Session Beans and Message Driven Beans, the changes in the persistence tier have truly been phenomenal. In fact, other than some naming patterns and concepts, JPA has very little in common with the EJB 2.x Entity Bean model. This is particularly true in the fact that JPA does not follow the container model that is just not very well suited to the problem of persistence. Instead it follows an API paradigm similar to JDBC, JavaMail or JMS. We will soon see, the JPA `EntityManager` interface defines the API for persistence while JPA Entities specify how application data is mapped to a relational database. Although JPA takes a serious bite out of the complexity in saving enterprise data, ORM based persistence is still a non-trivial topic. We will devote the entire third part of this book to JPA, namely Chapters 7, 8, 9 and 10..

As we know, data is saved into the database using JPA in almost every step of our ActionBazaar scenario. We won't bore you to death by going over all of the persistence code for the scenario. Instead, we'll introduce JPA using a representative example and leave you to explore the complete code on you own. We will see how EJB 3.0 Persistence looks like by revisiting the `PlaceBid` Stateless Session Bean. As a reminder to how the bidding process is implemented, Figure 2.7 depicts the different components that interact with each other when a bidder creates a bid in ActionBazaar.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



21
22 **Figure 2.7: PlaceBidServlet invokes the addBid method of PlaceBid EJB and passes a Bid object. The PlaceBidEJB invokes persist method of EntityManager to save the Bid entity into the database. When the transaction commits you will see that a corresponding database record in the BIDS table will be stored**

Recall that the `PlaceBidServlet` calls the `PlaceBidBean`'s `addBid` method to add a `Bid` Entity into the database. As we will see, the `PlaceBidBean` uses the JPA `EntityManager`'s `persist` method to save the bid. Let's first take a look at the JPA, and then we'll see the EntityManager in action.

2.5.1 Working with the Java Persistence API

You might have noticed that we kept the actual code to save a bid into the database conveniently out of sight in Listing 2.1. The `PlaceBid` EJB's `addBid` method references the hidden `save` method to persist the `Bid` Object to the database. Listing 2.8 will fill in this gap by showing you what the `save` method actually does. The `save` method uses the JPA `EntityManager` to save the `Bid` Object. But first let's take a quick look at the fourth and final kind of EJB—the JPA Entity. Listing 2.7 shows us how the `Bid` Entity actually looks like:

8 Listing 2.7: Bid Entity
```
package ejb3inaction.example.persistence;

import java.io.Serializable;
import java.sql.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.GenerationType;
import javax.persistence.GeneratedValue;
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
@Entity                                                            |#1
@Table(name="BIDS")                                                |#2
public class Bid implements Serializable {
    private Long bidID;
    private Long itemID;
    private Long bidderID;
    private Double bidAmount;
    private Date bidDate;


    @Id                                                             |#3
    @GeneratedValue(strategy=GenerationType.AUTO)                  |#4
    @Column(name="BID_ID")                                          |#5
    public Long getBidID() {
        return bidID;
    }

    public void setBidID(Long bidID) {
        this.bidID = bidID;
    }

    @Column(name="ITEM_ID")                                        |#5
    public Long getItemID() {
        return itemID;
    }

    public void setItemID(Long itemID) {
        this.itemID = itemID;
    }

    @Column(name="BIDDER_ID")                                      |#5
    public Long getBidderID() {
        return bidderID;
    }

    public void setBidderID(Long bidderID) {
        this.bidderID = bidderID;
    }

    @Column(name="BID_AMOUNT")                                     |#5
    public Double getBidAmount() {
        return bidAmount;
    }

    public void setBidAmount(Double bidAmount) {
        this.bidAmount = bidAmount;
    }

    @Column(name="BID_DATE")                                       |#5
    public Date getBidDate() {
        return bidDate;
    }

    public void setBidDate(Date bidDate) {
        this.bidDate = bidDate;
    }
}
```

(annotation) <#1 Marking POJO as Entity>
(annotation) <#2 Table mapping>
(annotation) <#3 Entity ID>

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

(annotation) <#4 ID value generation>
(annotation) <#5 Column mappings>

You can probably gather a pretty good idea of exactly how O-R mapping in JPA works just by glancing at Listing 2.7, even if you have no familiarly with ORM tools like Hibernate. Think about the annotations that mirror relational concepts like tables, columns and ids…

`@Entity` annotation signifies the fact that the `Bid` class is a JPA Entity#1. Note that `Bid` is a POJO that does not require an interface, unlike Session and Message Driven Beans. The `@Table` annotation tells JPA that the `Bid` Entity is mapped to the `BIDS` table#2. Similarly, the `@Column` annotations indicate what `Bid` properties map to what `BIDS` table fields. Note, Entities need not use getter and setter based properties. Instead, the field mappings could have been placed directly onto member variables exposed through non-private access modifiers. We'll learn more about access by Entity property and field in Chapter 7. `@Id` annotation is somewhat special. It marks the `bidID` property to be the primary key for the `Bid` Entity#3. Just like a database record, a primary key uniquely identifies an Entity instance. We have used the `@GeneratedValue` annotation with strategy set to `GenerationType.AUTO#4` to indicate that the persistence provider should automatically generate the primary key when the Entity is saved into the database.

The primary key generation

If you have used EJB 2.x you may remember that it was almost rocket science to generate primary key values with CMP entity beans. With EJB 3.0 JPA, the generation of primary keys is a snap and you have several options such as table, sequence, identity key and so on. We will discuss more about primary key generation in Chapter 8.

Although we didn't do things this way in Listing 2.7, the `Bid` Entity could have been related to a number of other JPA Entities by holding direct object references (such the `Bidder` and `Item` Entities). EJB 3.0 ORM allows such object reference based implicit relationships to be elegantly mapped to the database. We've decided to keep things simple for now and not dive into this quite this early. Instead we'll discuss Entity relationship mapping in Chapter 8.

Having looked at the `Bid` Entity, let's now turn our attention to how the Entity actually winds up in the database through the `PlaceBid` bean.

2.5.2 Using the EntityManager

You've probably noticed that the `Bid` Entity doesn't have a method to save itself into the database. The JPA `EntityManager` does this bit of heavy lifting in terms of actually reading ORM configuration and providing Entity persistence services through an API based interface.

The EntityManager

The `EntityManager` knows how to store a POJO Entity into the database as a relational record, read relational data from a database and turn in into an Entity, update Entity data stored in the database, and delete data mapped to an Entity instance from the database. As we will see in Chapters 9 and 10, the `EntityManager` has methods corresponding to each of these CRUD operations, in addition to support for the robust Java Persistence Query Language (JPQL).

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

As promised earlier, Listing 2.8 shows how the `PlaceBid` EJB uses `EntityManager` API to persist the Bid Entity.

9 Listing 2.8: Saving a bid record using the EJB 3.0 Java Persistence API

```
package ejb3inaction.example.buslogic;

...
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
...

@Stateless
public class PlaceBidBean implements PlaceBid {
    @PersistenceContext(unitName="actionBazaar")                       |#1
    private EntityManager entityManager;
    ...
    public Bid addBid(Bid bid) {
        System.out.println("Adding bid, bidder ID=" + bid.getBidderID()
             + ", item ID=" + bid.getItemID() + ", bid amount="
                 + bid.getBidAmount() + ".");

        return save(bid);
    }

    private Bid save(Bid bid) {
        entityManager.persist(bid);                                |#2
        return bid;
    }
}
```

(annotation) <#1 Injects instance of EntityManager>
(annotation) <#2 Persist Entity instance>

The true magic of the code in Listing 2.8 really lies in the `EntityManager` interface. One interesting way to think about the `EntityManager` interface is as an "interpreter" between the OO and relational worlds. The manager reads the ORM mapping annotations like `@Table` and `@Column` on the `Bid` Entity and figures out how to save the Entity into the database. The `EntityManager` is injected into the `PlaceBid` Bean through the `@PersistenceContext` annotation#1. Similar to the `name` parameter of the `@Resource` annotation in Listing 2.5, the `unitName` parameter of the `@Resource` annotation points to persistence unit specific to Act ionBazaar. A persistence unit is a group of entities packaged together in an application module. You will learn more about persistence units in Chapter 9 and 10.

In the `save` method, the `EntityManager persist` method is called to actually save the `Bid` data into the database#2. After the `persist` method returns, an SQL statement very much like the following is issued against the database to insert a record corresponding to the bid:

```
INSERT INTO BIDS (BID_ID, BID_DATE, BIDDER_ID, BID_AMOUNT, ITEM_ID)
VALUES (52, NULL, 60, 20000.50, 100)
```

It might be instructive to look back at Listing 2.7 now to see how the `EntityManager` figures out the SQL to generate by looking at the O-R mapping annotations on the `Bid` Entity. Recall that the `@Table` annotation specifies that the bid record should be saved in the `BIDS` table while each of the `@Column` annotations in Listing 2.7 tells JPA which `Bid` Entity field maps to which column in the `BIDS` table. For example, the `bidId` property maps to the `BIDS.BID_ID` column, the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

bidAmount property maps to the BIDS.BID_AMOUNT column and so on. As we discussed earlier the @Id and @GeneratedValue value annotations specify that the BID_ID column is the primary key of the BIDS table and that JPA should automatically generate a value for the column before the INSERT statement is issued (the 52 value in the SQL sample). This process of translating an Entity to columns in the database is exactly what Object-Relational Mapping (ORM) and JPA is really all about.

This brings us to the end of this brief introduction to the EJB 3.0 Java Persistence API and the end of this whirlwind Chapter. At this point, it should be clear to you how simple, effective and robust EJB 3.0 really is, even from a bird's eye view.

2.6 Summary

As we stated in the introduction, the goal of this Chapter was not to feed you the "guru pill" for EJB 3.0, but rather show you what to expect from this new version the Java enterprise platform.

This Chapter introduced the ActionBazaar application, a central theme to this book. Using a scenario from the ActionBazaar application, we have shown you a cross-section of EJB 3.0 functionality, including Stateless Session Beans, Stateful Session Beans, Message Driven Beans and the EJB 3.0 Java Persistence API. You also learnt some basic concepts such as deployment descriptors, metadata annotations and dependency injection.

We used a Stateless Session bean (PlaceBidBean) to implement the business logic to place a bid for an item in an auctioning system. We built a very simple Servlet client to access the bean that used dependency injection. We then saw a Stateful Session bean (PlaceOrderBean) that encapsulated the logic for ordering an item and build a simple application client that accesses the PlaceOrderBean. We saw a very simple example of a Message Driven bean, OrderBillingMDB, that processes a billing request when a message arrives on a JMS queue. Finally, we built an Entity for storing bids and used the EntityManager API to persist the Entity to the database.

Most of the rest of this book roughly follows the outline of this Chapter. Chapter 3 revisits Session Beans, Chapter 4 discusses messaging, JMS and Message Driven Beans, Chapter 5 expands on dependency injection and discusses topics such as interceptors and timers and Chapter 6 explores transactions and security management in EJB. Chapters 7 through 11 are devoted to a detailed exploration of the Persistence API. Finally, Chapters 12 through 15 cover advanced topics in EJB.

In the next Chapter, we will shift to a lower gear and talk about dive into the details of Session Beans.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 3 Building Business Logic with Session beans

At the heart of any Enterprise application resides its business logic. In an ideal world, application developers should only be concerned with defining and implementing the business logic, while concerns like presentation, persistence or integration should largely be window dressing. From this perspective, Session Beans are the most important part of EJB because their purpose in life is to model high-level business processes.

If you can think of a business system as a horse-drawn Chariot with a driver carrying the Greco-Roman champion to battle, Session Beans are the driver. Session Beans utilize data and system resources (the Chariot and the horses) to implement the goals of the user (the champion) using business logic (the skills and judgment of the driver). For this and other reasons, Sessions Beans, particularly Stateless Session Beans, have been very popular, even despite the problems of EJB 2.x. EJB 3.0 makes this vital bean type a lot easier to use.

In Chapter 1, we briefly introduced Session Beans. In Chapter 2 we saw simple examples of these beans in action. In this Chapter, we will discuss Session Beans in much greater detail, including their purpose, the different types of Session Beans, how to develop them and some of the advanced Session Bean features available to you.

We will start this Chapter by exploring some basic Session Bean concepts, and then discuss the basic characteristics of Session Beans.. Session Beans. We will then cover each Session Bean type – Stateful and Stateless - in detail before introducing bean client code. Finally, we will mention Session Bean best practices at the end of the Chapter.

3.1 Getting to know Session Beans

A typical enterprise application will have numerous business activities or processes. For example, our ActionBazaar application has processes such as creating a user, adding an item for auctioning, bidding for an item, ordering an item and many more. Session Beans can be used to encapsulate the business logic for all such processes.

The theory behind Session Beans centers on the idea that each request by a client to complete a distinct business process is completed in a *session*. So what is a session? If you have ever used UNIX server then you may have used telnet to connect to the server from a PC-client. Telnet allows you to establish a login session with the UNIX server for a finite amount of time. During this session you may execute several commands in the server. Simply put a *session* is a connection between a client and a server that lasts for a finite period of time.

A session may either be very short-lived like a HTTP request or span a long time like a login session when you telnet or ftp into a UNIX server. Similar to a typical telnet session a bean may maintain its state between calls, in which case it is stateful,, or it may be a one time call, in which case it's stateless. A typical example of a Stateful application is the module that a Bidder uses to register himself in the ActionBazaar application. That process takes place in multiple steps. An example of a

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Stateless business module is the application code that is used to place a bid for an item. Information, such as user id, item number, and amount are passed in and success or failure is returned. This happens all in one step. We will examine the differences between stateless and stateful session beans more closely in section 3.1.4.

As you might recall, Session Beans are the only EJB components that are invoked directly by clients. A client can be anything such as a web application component (Servlet, JSP, JSF and so on), a command line application or a Swing GUI desktop application. A client can even be a .NET application using Web Services access.

At this point you might be wondering what makes Session Beans special. After all, why use a session bean simply to act as a business logic holder? Glad that you asked. Before you invest more of your time, let us address this question first. Then we will dissect and show you basic anatomy of a session bean and rules that governs it before finding out about the differences between stateless and stateful session beans.

3.1.1 Why Use Session Beans?

Session Beans are a lot more than just business logic holders. Remember the EJB services we briefly mentioned in Chapter 1? A good majority of those services are specifically geared toward Session Beans. They make developing a robust, feature-rich, impressive business logic tier remarkably easy (and maybe even a little fun). Let's take a look at some of the most important of these services.

Concurrency and Thread-safety

The whole point of building server-side applications is that they can be shared by a large number of remote clients at the same time. Because Session Beans are specifically meant to handle client requests, they must support a high degree of concurrency safely and robustly. In our ActionBazaar example, it is very likely thousands of concurrent users will be using the `PlaceBid` Session Bean we introduced in Chapter 2. The container employs a number of techniques to "automagically" make sure you don't have to worry about concurrency or thread-safety. This means that we can develop Session Beans as though we were writing a standalone desktop application used by a single user. We will discuss these "automagic" techniques shortly, including pooling, session management and passivation.

Remoting and Web Services

Session Beans support both Java RMI (Remote Method Invocation) based native and SOAP based Web Services remote access. Other than some minor configuration, no work is required to make Session Bean business logic accessible remotely using either method. This goes a long way in enabling distributed computing and interoperability. We'll see Session Bean remoteability in action in juts a few Sections.

Transaction and Security Management

Transactions and security management are two enterprise-computing mainstays. Session Bean support for pure configuration based transactions, authorization and authentication makes supporting these requirements all but a non-issue. We won't discuss these services in this Chapter but will reserve the entirety of Chapter 6 to EJB transaction management and security.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Timer Services and Interceptors

Interceptors are EJB's version of lightweight Aspect Oriented Programming (AOP). Recall that AOP is the ability to isolate "crosscutting" concerns into their own modules and apply them across the application through configuration. "Crosscutting" concerns include things like auditing and logging that are repeated across an application but are not directly related to business logic. We'll discuss interceptors in great detail in Chapter 5.

Timer Services are EJB's version of lightweight application schedulers. In most medium to large scale applications, you will find that you need some kind of scheduling services. In ActionBazaar, scheduled tasks could be used to monitor when the bidding for a particular item ends and determine who won an auction. Timer Services allow us to easily turn a Session Bean into a recurring or non-recurring scheduled task. We'll save the discussion of timer services to Chapter 5 as well.

A Session Bean Alternative: Spring

Clearly, EJB 3.0 Session Beans are not your only option in developing the business tier of your application. POJOs managed by lightweight containers such as Spring could also be used to build the business logic tier. Before jumping on either the EJB 3.0 Session Bean or Spring bandwagon you should clearly think about what your needs are.

If your application needs robust remoteability or seamlessly exposing your business logic as Web Services, EJB 3.0 is the clear choice. Spring also does not have very good equivalents of instance pooling, automated session state maintenance and passivation/activation. Because of heavy use of annotations, you can pretty much avoid "XML Hell" using EJB 3.0. This is not true of Spring. Moreover, because it is an integral part of the Java EE standard the EJB container is natively integrated with components such as JSF, JSP, Servlets, the JTA transaction manager, JMS providers and JAAS security providers of your application server. With Spring, you have to worry whether your application server fully supports the framework with these native components and other high performance features like clustering, load balancing and failover.

If these are not factors you are worried about, Spring is not a bad choice at all and has a few strengths of its own. Spring has numerous simple, elegant utilities to perform many common tasks such as the JDBC and JMS templates. If you plan to use dependency injection with regular Java Classes, Spring is great since DI only works for container components in EJB 3.0. Also, Spring AOP/AspectJ is a much more feature-rich (albeit slightly more complex) choice than EJB 3.0 interceptors.

Nevertheless, if portability, standardization and vendor support is important to you, EJB 3.0 may be the way to go anyway. EJB 3.0 is a mature product that is the organic (albeit imperfect) result of the incremental effort, pooled resources, shared ownership and measured consensus of numerous distinct groups of people. This includes the grassroots Java Community Process (JCP), some of the world's most revered commercial technology powerhouses like IBM, Sun, Oracle, BEA, etc and well as spirited open source organizations like Apache and JBoss.

Now that you are convinced why you would session beans let us learn some basic characteristics about session beans.

3.1.2 Session Beans: the basics

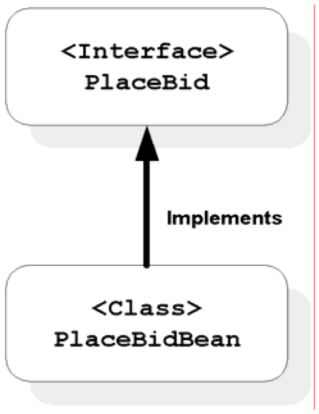
Although we briefly touched upon Session Bean code in the previous Chapter, we didn't really go into a great detail about developing them. Before we dive into the details of developing each type of Session Bean, let's revisit the code in Chapter 2 to closely examine some basic traits shared by all Session Beans.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The Anatomy of a Session Bean

Although this was understated in Chapter 2, each Session Bean implementation has two distinct parts—one or more Bean interfaces and a Bean implementation class. In the `PlaceBid` Bean example of Chapter 2, the bean implementation consisted of the `PlaceBid` interface and the `PlaceBidBean` class. This is shown in Figure 3.1.



23 **Figure 3.1: Parts of the Place Bid Session Bean. Each Session Bean has one or more interfaces and one implementation class.**

All Session Beans must be divided into these two parts. This is because clients cannot have access to the bean implementation class directly. Instead, they must use Session Beans through a business interface. Nonetheless, *interface based programming* is a very sound idea anyway, especially when using dependency injection.

Interface based programming is the practice of not using implementation classes directly whenever possible. This promotes loose coupling since implementation classes can easily be swapped out without a lot of code changes. EJB has been a major catalyst in the popularization of interface based programming and even the earliest versions of EJB followed this paradigm, later to form the basis of DI.

The Session Bean Interface

An interface that a client invokes the bean through is called a *business interface*. A business interface essentially defines the bean methods that are appropriate for access through a specific access mechanism. For example, let's revisit the `PlaceBid` interface in Chapter 2. Here is the code:

```
@Local
public interface PlaceBid {
    Bid addBid(Bid bid);
}
```

All EJB interfaces being POJI, there really isn't anything too remarkable in the code above other than the `@Local` annotation that specifies it's a local interface. Recall that a business interface could be remote or even Web Service accessible instead. We'll talk more about the three types of interfaces in Section 3.2. The interesting thing to note right now is the fact that a single EJB can have multiple interfaces. This means that EJB implementation classes can be *polymorphic*, meaning that different clients using different interfaces could use them in completely different ways.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The EJB Bean Class

Just like usual OO programming, each interface that the Bean intends to support must be explicitly included in the Bean implementation class's `implements` clause. We can see this in the code for the `PlaceBidBean` from Chapter 2:

```
@Stateless
public class PlaceBidBean implements PlaceBid {
    ...
    public PlaceBidBean() {}

    public Bid addBid(Bid bid) {
        System.out.println("Adding bid, bidder ID=" + bid.getBidderID()
             + ", item ID=" + bid.getItemID() + ", bid amount="
                 + bid.getBidAmount() + ".");

        return save(bid);
    }
    ...
}
```

The `PlaceBidBean` class provides the concrete implementation of the `addBid` method required by the `PlaceBid` interface. Session Bean implementation classes can never be abstract, this means that all methods mandated by declared business interfaces must be implemented in the class.

A nuance to note is the fact that EJB implementation classes can have non-private methods that are not accessible through any interface. This can be very useful for creating clever unit testing frameworks. This is also helpful while implementing lifecycle callback, as we will discuss in Section 3.1.4. Also, an EJB bean class can make use of OO inheritance. You could use this strategy to support a custom framework for your application. As an example, you could put commonly used logic in a parent POJO class that a set of Beans inherits from.

Unit Testing Your Session Beans

It is clear that Session Beans are POJOs. Since EJB annotations are ignored by the JVM Session Beans can be unit tested using a framework like JUnit or TestNG without having to deploy them into an EJB container. For more information on JUnit, browse http://www.junit.org.

On the other hand, since several container-provided services such as dependency injection cannot be used outside the container so you cannot perform functional testing of applications using EJBs outside the container, at least not easily.

Now that we've looked at the basic structure of Session Beans, we will outline relatively simple programming rules for a session bean.

3.1.3 Understanding the programming rules

Like all EJB 3.0 Beans, Session Beans are POJOs that follow a very small set of rules. The following summarizes the rules that apply to all types of Session Beans:

5. As we discussed earlier in section 3.1.2 a session bean must have a business interface.
6. The Session Bean class must be concrete. You cannot define a Session Bean class to be either final or abstract since the container needs to manipulate it.
7. You must have a no argument constructor in the Bean class. As we saw, this is because the

Licensed to John Sweitzer <admin@saolailaem.info>

container invokes this constructor to create a Bean instance when a client invokes an EJB. Note that complier inserts a default no-argument constructor if there is no constructor in a Java class.

8. A Session Bean class can subclass another Session Bean or any other POJO. For example a Stateless Session Bean named `BidManager` can extend another session bean `PlaceBidBean` in the following way:

```
@Stateless
public BidManagerBean extends PlaceBidBean implements BidManager {
    ...
}
```

9. The business methods and lifecycle callback methods may either be in the Bean Class or in a superclass. It's worth mentioning here that annotation inheritance is supported with several limitations with EJB 3.0 session beans. For example the bean type annotation `@Stateless` or `@Stateful` specified in the `PlaceBidBid` superclass will be ignored when you deploy the `BidManagerBean`. However any annotations in the superclass use to define lifecycle callback methods (more about that later in this section) and resource injection will be inherited by the bean class.

10. Business method names must not start with `ejb`. For example, we should avoid a method name like `ejbCreate` or `ejbAddBid` because it may interfere with EJB infrastructure processing. We must define all business methods as `public` and we cannot declare them as `final` or `static`. If you are exposing a method in a remote business interface of the EJB then you should make sure that the arguments and the return type of the method implement the `java.io.Serializable` interface.

We will see these rules applied when we explore concrete examples of Stateless and Stateful Session Beans in section 3.2 and 3.3 respectively.

Now that we've looked at the basic programming rules for the session beans, let's discuss the fundamental reasons behind splitting them into two groups.

3.1.4 Conversational State and Session Bean Types

Earlier, we talked about stateful and stateless Session beans. However we have so far avoided the real differences between them. This grouping of bean types centers on the concept of the *conversational state*.

A particular business process may involve more than one Session Bean method call. During these method calls, the Session Bean may or may not maintain a *conversational state*. This terminology will make more sense if you think of each Session Bean method call as a "conversation" or "exchange of information" between the client and the bean. A bean that maintains conversational state "remembers" the results of previous exchanges, and is a *Stateful Session Bean*. In Java terms, this means that the bean will store data from a method call into instance variables and use the cached data to process the next method call. *Stateless Session Beans* do not maintain any state. In general, Stateful Session Beans tend to model multi-step workflows, while Stateless Session Beans tend to model general-purpose, utility services used by the client.

The classic example of maintaining conversational state is the eCommerce website shopping cart. When the client adds, removes, modifies or checks out items from the shopping cart, the shopping cart is expected to store all of the items that were put into it while the client was shopping. As you can imagine, except for the most complex business processes in an application, most Session Bean

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

interactions do not require a conversational state. Putting in a bid at ActionBazaar, leaving Buyer or Seller feedback, viewing a particular item on bid are all examples of *stateless* business processes.

As we will soon see, however, this does not mean that Stateless Session Beans cannot have instance variables. Even before we explore any code, common sense should tell us that in the least, Session Beans must cache some resources like database connections for performance reasons. The critical distinction here is *client expectations*. As long as the client does not need to depend on the fact that a Session Bean uses instance variables to maintain conversational state, there is no need to use a Stateful Session Bean.

3.1.5 Bean Lifecycle Callbacks

A Session Bean has a life cycle. This mean that Beans go through a predefined set of state transitions. If you've used Spring or EJB 2.x, this should come as no surprise. If you haven't the concept can be a little tricky to grasp.

In order to understand the Bean lifecycle, it is important to revisit the concept of *managed resources*. Recall that the container manages almost every aspect of Session Beans. This means that neither the client nor the Bean is responsible for determining when Bean instances are created, when dependencies are injected, when Bean instances are destroyed or when to take optimization measures. Managing these actions enables the container to provide the abstractions that constitute some of the real value of using EJBs, including DI, automated transaction management, AOP, transparent security management and so on.

The lifecycle events

The lifecycle of a session bean may be categorized into several phases, or events. The most obvious two events of Bean life cycle is of course *creation* and *destruction*. All EJBs go through these two phases. In addition, Stateful Session Beans go through the *passivation/activation* cycle, which we will discuss in depth in section 3.3. Here, we will take a close look at the phases shared buy all Session Bean: creation and destruction.

The lifecycle for a Session Bean starts when a bean instance is created. This typically happens when a client gets a reference to the bean either by doing a JNDI lookup or using dependency injection. The following steps occur when a bean is initialized:

1) The container invokes the `newInstance` method on the bean object. This essentially translates to a constructor invocation on the bean implementation class.

2) If the bean uses DI, all dependencies on resources, other Beans and environment components are injected into the newly created bean instance.

Figure 3.2 depicts this series of events.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

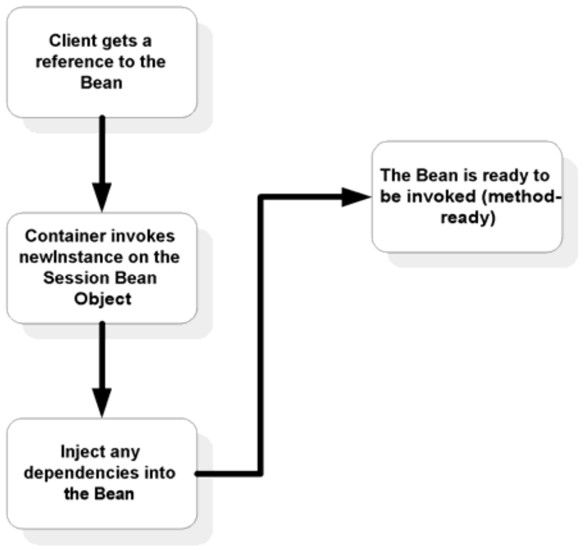

Client gets a reference to the Bean
Container invokes newInstance on the Session Bean Object
Inject any dependencies into the Bean
The Bean is ready to be invoked (method-ready)

Figure 3.2: The lifecycle of an EJB starts when a method is invoked. The container creates a bean instance and then dependencies on resources are injected. The instance is then ready for method invocation.

After the container determines that an instance is no longer needed, the instance is destroyed. This sounds just fine until you realize that the Bean might need to know when some of its life-cycle transitions happen. As an obvious example, suppose that the resource being injected into a bean is a JDBC data source. That means that it would be nice to be able to know when it is injected so you can open the JDBC database connection to be used in the next business method invocation. Similarly, the bean would also need to be notified before it is destroyed so the open database connection can be properly closed.

This is where callbacks come in.

Understanding lifecycle callbacks

Life-cycle callbacks are essentially bean methods (typically not exposed by a business interface) that the container calls to notify the bean about a life-cycle transition, or event. When the event occurs the container will invoke the corresponding callback method, and you can use these methods to perform business logic or operations such as initialization and clean up of resources .

Callback methods are Bean methods that are marked with metadata annotations such as `@PostContruct` and `@PreDestroy`. They can be `public, private, protected` or `package-protected`. As you might have already guessed, a `PostConstruct` callback is invoked just after a bean instance is created and dependencies are injected. A `PreDestroy` callback is invoked just before the bean is destroyed and is useful for cleaning up resources used by the bean.

While all Session beans have `PostConstruct` and `PreDestroy` lifecycle events, stateful Session beans have two additional ones: passivation and activation. Since Stateful session beans maintain state, there is a stateful session bean instance for each client, and there could be many instances of a stateful session bean in the container. If this happens, the container may decide to deactivate a stateful bean instance temporarily when not in use and this process is called *passivation*. It activates the bean instance again when client needs it and this process is called *activation*. The `@PrePassivate` and `@PostActivate` annotations apply to the passivation and activation life cycle events.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Note you can define a lifecycle callback method either in the bean class or in a separate interceptor class.

Table 3.1 lists the life cycle callback method annotations, where they are applied and what the callback methods are typically used for.

1.4 Table 3.1: Lifecycle callbacks are created to handle lifecycle events for an EJB. We can create these callback methods either in the bean class or in an external interceptor class

Callback annotation	Type of EJB	What do can we do
javax.annotation.PostConstruct	Stateless, Stateful, MDB	The annotated method is invoked after a bean instance is created and dependency injection is complete. Typically this callback is used to initialize resources (for example opening database connections).
javax.annotation.PreDestroy	Stateless, Stateful, MDB	The annotated method is invoked prior to a bean instance being destroyed. Typically this callback is used to cleanup resources (for example closing database connections).
javax.ejb.PrePassivate	Stateful	The annotated method is invoked prior to a bean instance being passivated. Typically this callback is used to clean up resources, such as database connections, TCP/IP sockets, or any resources that cannot be serialized during passivation.
javax.ejb.PostActivate	Stateful	The annotated method is invoked after a bean instance is activated. Typically this callback is used to restore resources, such as database connections that we cleaned up in the PrePassivate method.

In section 3.2.4 and 3.3.4 we will see how you can define lifecycle callback methods in the Bean class for stateless and stateful beans respectively. We will defer to the discussion of lifecycle callback methods in the interceptor classes to Chapter 5.

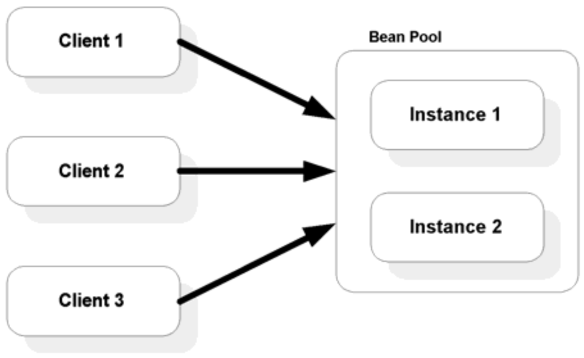
Now that we've covered the basics of session beans, we will start our detailed exploration with the simpler Stateless Session Bean model, saving Stateful Session Beans for later.

3.2 Stateless Session Beans

As we noted, Stateless Session Beans model tasks that do not maintain conversational state. This means that Session Beans model task that can be completed in a single method call, like placing a bid. However, this does not mean that all Stateless Session Beans contain a single method, as is the case for the `PlaceBidBean` in Chapter 2. In fact, real-world Stateless Session Beans often contain a bunch of closely related business methods, like the `BidManager` Bean we will introduce soon. By and large, Stateless Session Beans are most popular kind of Session Beans. They are also the most performance efficient. To understand why, take a look at Figure 3.3. It shows a high level schematic of how stateless Session Beans are typically used by clients.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



25 **Figure 3.3 Stateless Session bean instances can be pooled and may be shared between clients. When a client invokes a method in a Stateless Session bean the container either creates a new instance in the bean pool for the client or assigns one from the bean pool. The instance is returned to the pool after use.**

As we will soon talk about in greater detail, Stateless Beans are pooled. This means that for each managed bean, the container keeps a certain number of instances handy in a pool. On each client request, an instancefrom the pool is quickly assigned to the client. When the client request finishes, the instance is returned to the pool for reuse. This means that a small number of bean instances can service a relatively large number of clients.

In this section you will learn more about developing stateless session beans. We will develop part of business logic of ActionBazaar system using a stateless session to illustrate its use. You will learn use of `@Stateless` annotation, use of different types of business interfaces supported with stateless session bean and then lifecycle callbacks such as `@PostConstruct` and `@PreDestroy` supported with stateless session bean.

Before we jump into analyzing code we'll briefly discuss the ActionBazaar business logic that we will implement as a stateless session bean..

3.2.1. The BidManagerBean example

Bidding is a critical part of the ActionBazaar functionality. Users may bid on an item and view the current bids on an item, while ActionBazaar admins and Customer Service Representatives may remove bids under certain circumstances. Figure 3.4 depicts these bid related actions.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

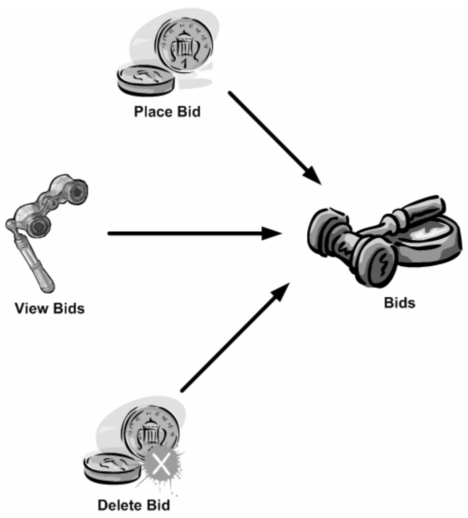


Figure 3.4: Some ActionBazaar bid related actions. While bidders can place bids and view the current bids on an item, admins can remove bids when needed. All of these actions can be modeled with a singe Stateless Session Bean.

Because all of these bid-related functions are pretty simple, single step processes, a Stateless Session Bean can be used to model all of them. The `BidManagerBean` presented in Listing 3.1 contains methods for adding, viewing and canceling (removing) bids. This is essentially an enhanced, more realistic version of the basic `PlaceBid` EJB we saw earlier. The complete code is available for download from http://ejb3inaction.com in Chapter3.zip.

> Note that we are using JDBC for simplicity only because we have not introduced the EJB 3.0 Java Persistence API in any detail quite yet and don't assume you already understand ORM. Using JDBC also happens to demonstrate the usage of dependency injection of resources and the Stateless Bean lifecycle callbacks pretty nicely! In general, you should avoid using JDBC in favor of JPA once you are comfortable with it.

10 Listing 3.1 : Stateless Session bean example

```
@Stateless(name="BidManager")                                          |#1
public class BidManagerBean implements BidManager {
    @Resource(name="jdbc/ActionBazaarDS")                              |#2
    private DataSource dataSource;
    private Connection connection;
    ...
    public BidManagerBean() {}

    @PostConstruct                                                     |#3
    public void initialize() {
        try {
            connection = dataSource.getConnection();
        } catch (SQLException sqle) {
             sqle.printStackTrace();
        }
     }

     public void addBid(Bid bid){
        try {
            Long bidId = getBidId();
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
            Statement statement = connection.createStatement();
            statement.execute(
                "INSERT INTO BID ("
                    + “BID_ID, “
                    + "BID_AMOUNT, “
                    + “BID_BIDDER_ID, “
                    + “BID_ITEM_ID) "
                    + "VALUES ("
                    + bidId + “, “
                    + bid.getAmount() + “, ”
                    + bid.getBidder().getUserId() + “, ”
                    + bid.getItem().getItemId()+ ")");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @PreDestroy                                                   |#4
    public void cleanup() {
        try {
            connection.close();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }


    private Long getBidId() { ... }

    public void cancelBid(Bid bid) {...}

    public List<Bid> getBids(Item item) {...}
    ...
}
...
@Remote                                                           |#5
public interface BidManager {
    void addBid(Bid bid);
    void cancelBid(Bid bid);
    List<Bid> getBids(Item item);
}
```
(annotation) <#1 Mark as Stateless Bean>
(annotation) <#2 Inject data source>
(annotation) <#3 Post-construct callback>
(annotation) <#4 Pre-destroy callback>
(annotation) <#5 Remote business inteface>

Briefly scanning Listing 3.1, let’s note the major features of the code.

As we’ve seen before, the `@Stateless` annotation to marks the POJO as a Stateless Session Bean#1. The `BidManagerBean` class implements the `BidManager` interface, which is marked `@Remote`#5. We use the `@Resource` annotation to perform injection of a JDBC data source#2. The `BidManagerBean` has a no argument constructor that the container will use to create instances of `BidManagerBid` EJB object. The post-construct#3 and pre-destroy#4 callbacks are used to manage a JDBC database connection derived from the injected data source. Finally, the `addBid` business method adds a bid into the database.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

We will start exploring the features of EJB 3.0 Stateless Session Beans by analyzing this code next, starting with the `@Stateless` annotation.

3.2.2 Using the @Stateless Annotation

As we know, the `@Stateless` annotation marks the `BidManagerBean` POJO as a Stateless Session Bean. Believe it or not, other than marking a POJO for the purposes of making the container aware of its purpose, the annotation really does not do very much else. The specification of the

`@Stateless` annotation is as follows:

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
   String mappedName() default "";
   String description() default "";

}
```

The single parameter, `name`, specifies the name of the Bean. Some containers use this parameter to bind the EJB to the global JNDI tree. Recall that JNDI is essentially the application server's managed resource registry. All EJBs automatically get bound to JNDI as soon as they catch the container's watchful eye. You will see real use of the `name` parameter in Chapter 11 when we discuss deployment descriptors. In Listing 3.1, the Bean name is specified to be 'BidManager'. As the annotation definition shows, the `name` parameter is optional since it is defaulted to an empty `String`. We could easily omit it as follows:

```
@Stateless
public class BidManagerBean implements BidManager {
```

If the `name` parameter is omitted, the container assumes that the bean name should be set to the name of the Class. In this case, the Bean name would be assumed to be 'BidManagerBean'. The `mappedName` is a vendor specific name that you can assign to your EJB, some containers such as Glassfish application server uses this name to assign the global JNDI name for the EJB. As we noted, the `BidManagerBean` implements a business interface named `BidManager`. Although we've touched on the idea of a business interface, we haven't really dug very deep into the concept. This is a great time to do exactly that.

3.2.3 Specifying Bean Interfaces

In section 3.1, we introduced you to EJB interfaces. Now let's explore a bit more how they work with stateless session beans. Client applications can invoke a Stateless Session Bean in exactly three different ways. In addition to local invocation within the same JVM and remote invocation through RMI, Stateless Beans can also be invoked remotely as Web Services. There are three types of business interfaces that correspond to the different access types; each is identified through a distinct annotation. Let's take a detailed look at these annotations:

Local interface

A local interface is for clients of Stateless Session Beans collocated in the same container (JVM) instance. As we've seen, we designate an interface to be a local business interface by using the `@Local` annotation. The following could be a local interface for the `BidManagerBean` class in Listing 3.1:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
@Local
public interface BidManagerLocal {
    void addBid(Bid bid);
    void cancelBid(Bid bid);
    List<Bid> getBids(Item item);
}
```

Local interfaces do not require any special measures either in terms of defining or implementing them.

Remote interface

Clients residing outside the EJB container's JVM instance must use some kind of remote interface. If the client is also written in Java, the most logical and resource efficient choice for remote EJB access is Java Remote Method Invocation (RMI). In case you are unfamiliar with RMI, we will provide a brief introduction to RMI in the Appendix. For now, all you need to know is that it is a highly efficient, TCP/IP-based remote communication API that automates most of the work needed for calling a method on a Java object across a network. EJB 3.0 enables a Stateless Bean to be made accessible via RMI through the `@Remote` annotation. The `BidManager` business interface in our example uses the annotation to make the bean remotely accessible:

```
@Remote
public interface BidManager extends Remote {
    ...
}
```

A remote business interface may extend `java.rmi.Remote` as we do above, although this is optional. Typically the container will perform byte code enhancements during deployment to extend `java.rmi.Remote` if our bean interface does not extend it. Remote business interface methods are not required to throw `java.rmi.RemoteException` unless the business interface extends `java.rmi.Remote` interface. Remote business interfaces do have *one* special requirement. All parameters and return types of interface methods *must* be `Serializable`. This is because only `Serializable` objects can be sent across the network using RMI.

Web service Endpoint Interface

We previously introduced local and remote interfaces, but there is a third one specific to Stateless Session Beans that you haven't seen yet: the Web Service Endpoint Interface (also known as SEI). The ability to expose a Stateless Session Beans as a SOAP-based Web Service is one of the most powerful features of EJB 3.0. All you need to do to make a bean SOAP accessible is marking a business interface with the `@javax.jws.WebService` annotation. The following defines a simple web service endpoint interface for the `BidManagerBean`:

```
@WebService
public interface BidManagerWS {
    void addBid(Bid bid);
    List<Bid> getBids(Item item);
}
```

Note we have omitted the `cancelBid` bean method from the interface, as we do not want this functionality to be accessible via a Web Service although it is accessible locally as well as remotely through RMI. The `@WebService` annotation does not place any special restrictions on either the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

interface of the implementing Bean. We will discuss EJB Web Services in much greater detail in Chapter 15.

Working with Multiple Business Interfaces

Although it is tempting, you *cannot* mark the same interface with more than one access type annotation. For example, you cannot mark the `BidManager` interface in Listing 3.1 with both the `@Local` and `@Remote` annotations instead of creating separate `BidManagerLocal` (local) and `BidManager` (remote) interfaces, although both interfaces expose the exact same bean methods.

However, a business interface can extend another interface and you can remove code duplication by creating a business interface that has common methods and business interfaces that extend the common "parent" interface. For example we can create a set of interfaces utilizing OO inheritance as follows:

```
public interface BidManager{
    void addBid(Bid bid);
    List<Bid> getBids(Item item);
}

@Local
public interface BidManagerLocal extends BidManager {
    void cancelBid(Bid bid);
}

@Remote
public interface BidManagerRemote extends BidManagerLocal {
}

@WebService
public interface BidManagerWS extends BidManager {
}
```

If you want, you can apply the `@Local`, `@Remote` or `@WebService` annotations in the bean class without having to implement the business interface as follows:

```
@Remote(BidManager.class)
@Stateless
public class BidManagerBean {
    ...
}
```

The preceding code marks the `BidManager` interface as remote through the bean class itself. This way, if you change your mind later, all you would have to do is change the access type specification in the bean class without ever touching the interface.

Next, we move onto discussing the EJB lifecycle in our example.

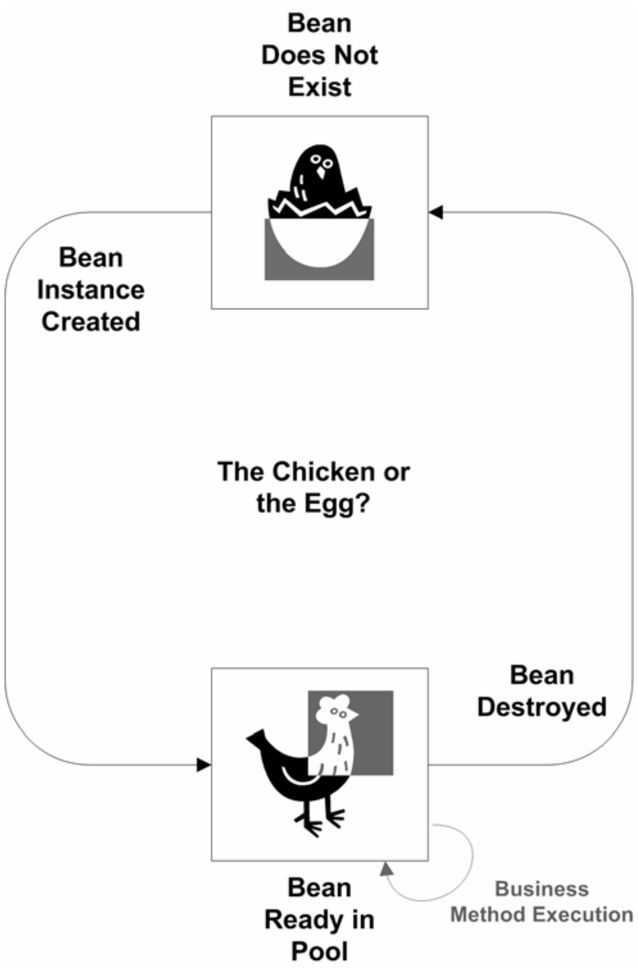
3.2.4 Using Bean Life-Cycle Callbacks

We introduced you to life cycle callback methods or callbacks, earlier in the Chapter; now let's take a deeper look at how they are used with stateless session beans. As far as EJB life cycles go, Stateless Session Beans have a very simple one as depicted in Figure 3.4. In effect, the container:

1. Creates bean instances using the default constructor as needed.
11. Injects resources such as database connections.
12. Puts instances in a managed pool.

Licensed to John Sweitzer <admin@saolailaem.info>

13. Pulls an idle bean out of the pool when an invocation request is received from the client (the container may have to increase the pool size at this point).
14. Executes the requested business method invoked through the business interface by the client.
15. When the business method finishes executing, pushes the idle bean back into the "method-ready" pool.
16. As needed, retires (a.k.a. destroys) beans out of the pool.



27 **Figure 3.4: The chicken or the egg – the stateless session bean life cycle is has three states: does not exist, idle and busy. As a result, there are only two life-cycle callbacks corresponding to bean creation and destruction.**

A very important point to notice from the Stateless Session Bean lifecycle is that since beans are allocated from and returned to the pool on a per-invocation basis, Stateless Session beans are extremely performance friendly and a relatively small number of bean instances can handle a large number of virtually concurrent clients.

As previously stated, there are two types of Stateless Session Bean lifecycle callback methods: callbacks that are invoked when the `PostConstruct` event occurs immediately after a Bean instance is created, setup and all the resources are injected, and callbacks that are invoked when the `PreDestroy` event occurs, right before the Bean instance is retired and removed from the pool.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Note, you can have multiple post-construct and pre-destroy callbacks for a given bean (although this is seldom used).

In Listing 3.1, the life cycle callback methods embedded in the bean are `initialize()` and `cleanup()`. Callbacks must follow the pattern of `public void <METHOD>()`. Unlike business methods, callbacks cannot throw checked exceptions (any exception that does not have `java.lang.RuntimeException` as a parent).

As we noted, the typical usage of these callbacks is for allocating and releasing injected resources that are used by the business methods, which is exactly what we do in our example of `BidManagerBean` in Listing 3.1. In Listing 3.1 we open and close connections to the database using an injected JDBC data source.

Recall that the `addBid` method in Listing 3.1 inserted the new bid submitted by the user. The method created a `java.sql.Statement` from an open JDBC connection and used the statement to insert a record into the `BIDS` table. The JDBC connection object used to create the statement is a classic heavy-duty resource. It is expensive to open and should be shared across calls whenever possible. It can hold a number of native resources, so it is very important to close the JDBC connection when it is no longer needed. We accomplish both these goals using callbacks as well as resource injection.

In Listing 3.1, the JDBC data source that the connection is created from is injected using the `@Resource` annotation. We will discuss more injecting resources using the `@Resource` annotation in Chapter 5. For now, though this is all that you need to know.

Let's take a closer look at how we used the callbacks in listing 3.1.

@PostConstruct callback

The `setDataSource` method saves the injected data source in an instance variable. After injecting all resources, the container checks if there are any designated `PostConstruct` methods that need to be invoked before the bean instance is put into the pool. In our case, we mark the `initialize` method in Listing 3.1 with the `@PostConstruct` annotation:

```
@PostConstruct
public void initialize() {
    ...
    connection = dataSource.getConnection();
    ...
}
```

In the `initialize` method, we create a `java.sql.Connection` from the injected data source and save it into the `connection` instance variable used in `addBid` each time the client invokes the method.

@PreDestroy callback

As we know, at some point, the container decides that our bean should be removed from the pool and destroyed (perhaps at server shutdown). The `PreDestroy` callback given us a chance to cleanly teardown bean resources before this is done. In the `cleanup` method marked with the `@PreDestroy` annotation in Listing 3.1, we tear down the open database connection resource before the container retires our bean:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
@PreDestroy
public void cleanup() {
    ...
    connection.close();
    connection = null;
    ...
}
```

Since bean instances from the pool are assigned randomly for each method invocation, trying to store client-specific state across method invocations is useless since the same bean instance may not be used for subsequent calls by the same client. On the other hand, Stateful Session Beans, which we will discuss next, are ideally suited for this situation.

3.3 Stateful Session Beans

Stateful Session Beans are the cousins of Stateless Session Beans but they are guaranteed to maintain conversational state. Stateful Session Beans are not programmatically very different from what we have already discussed for Stateless Session Beans. As a matter of fact, the only real difference between Stateless and Stateful Beans is how the container manages their lifecycle. Unlike Stateless Beans, the container makes sure that subsequent method invocations by the same client are handled by the same Stateful Bean instance. Figure 3.5 shows the one-to-one mapping between a bean instance and a client enforced behind the scenes by the container. As far as you are concerned, this one-to-one relation management happens "automagically".

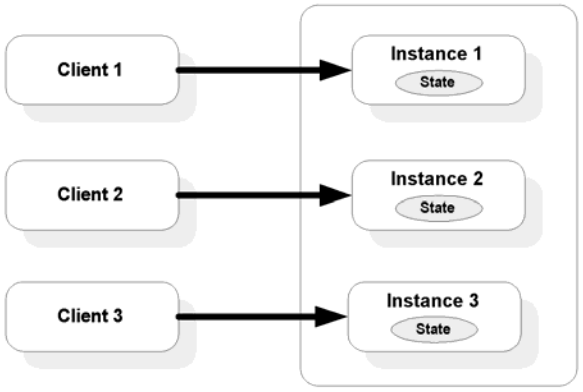


Figure 3.5: Stateful Bean session maintenance. There is a bean instance reserved for a client and each instance stores the client's state information. The bean instance exists until either removed by the client or timed out.

The one-to-one mapping between a client and a bean instance makes saving bean conversational state in a useful manner possible. However, this one-to-one correlation comes at a price. Bean instances cannot be readily returned to the pool and reused while a client session is still active. Instead, a bean instance must be squirreled away in memory to wait for the next request from the client owning the session. As a result, Stateful Session Bean instances held by a large number of concurrent clients can have a significant memory footprint. An optimization technique called *passivation*, which we will discuss soon, is used to alleviate this problem. Stateful Session Beans are

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

ideal for multi-step, workflow oriented business processes. In this Section, we will explore Stateful Beans by using the ActionBazaar bidder account creation workflow as an example.

We will examine a usecase of ActionBazaar application and implement it using a stateful session beans. You will learn additional programming rules for stateful session beans, stateful bean lifecycle callback methods `@Remove` method. Finally we will summarize differences between stateless and stateful session beans.

However, before we jump into code, we will briefly mention the rules that are specific to developing a Stateful Session Bean.

3.3.1 Additional programming rules

In section 3.1.3, we discussed the programming rules that apply to all session beans. Stateful session bean have a few minor twists on these rules:

Stateful Bean instance variables used to store conversational state must be Java primitives or `Serializable` objects. We will talk more about this requirement when we cover passivation.

17. Since stateful Session Beans cannot be pooled and reused like Stateless beans, there is a real danger of accumulating too many of them if we don't have a way to destroy them. Therefore, we have to define a business method for removing the bean instance by the client using the `@Remove` annotation. We'll talk more about this annotation soon.
18. In addition to the `@PostConstruct and @PreDestroy` lifecycle callback methods, Stateful Session Beans also have the `@PrePassivate` and `@PostActivate` lifecycle callback methods. A `@PrePassivate` method is invoked before a Stateful bean instance is passivated and `@PostActivate` is invoked after a bean instance is brought back into the memory and is method ready.

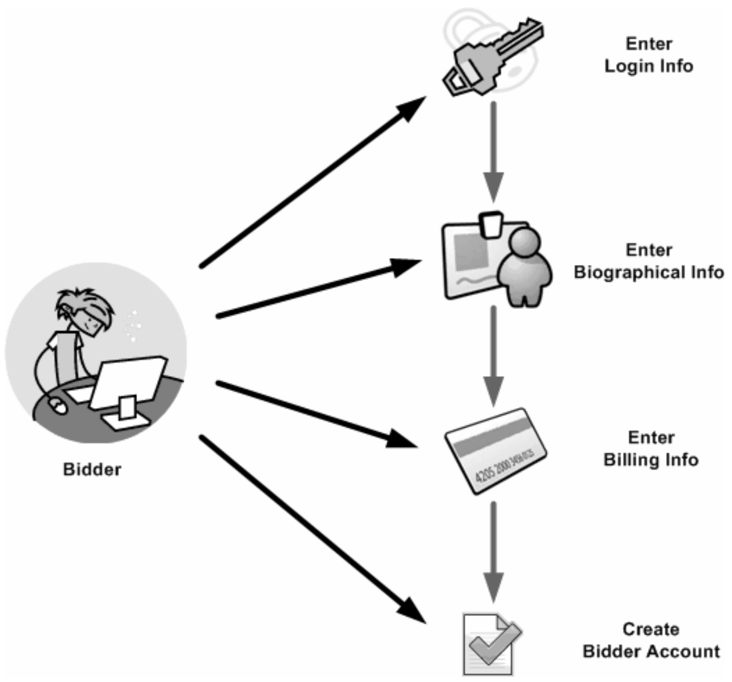
We will see these rules applied when we explore a concrete Stateful Session Beans example next. As we did for Stateless Beans, we will utilize the example as a jump off point to detail Stateful features.

3.3.2The BidderAccountCreatorBean example

The process to create an ActionBazaar bidder account is too involved to be implemented as a single-step action. As a result, account creation is implemented as a multi-step process. At each step of the workflow, the would-be bidder enters digestible units of data. For example, the bidder may enter username/password information first, then biographical information such as name, address and contact information, then billing information such as credit card and bank account data and so forth. At the end of a workflow, the bidder account is created or the entire task is abandoned. This workflow is depicted in Figure 3.6.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



29 **Figure 3.6: The ActionBazaar bidder account creation process is broken up into multiple steps: entering username/password, entering biographical information, entering billing information and finally creating the account. This workflow could be implemented as a Stateful Session Bean.**

Each step of the workflow is implemented as a method of the `BidderAccountCreatorBean` presented in Listing 3.2. Data gathered in each step is incrementally cached into the Stateful Session Bean as instance variable values. Calling either the `cancelAccountCreation` or `createAccount` methods ends the workflow. The `createAccount` method creates the bidder account in the database and is supposed to be the last "normal" step of the workflow. The `cancelAccountCreation` method on the other hand, prematurely terminates the process when called by the client at any point in the workflow and nothing is saved into the database.

The full version of the code is available for download in Chapter3.zip from http://www.ejb3inaction.com.

11 Listing 3.2 Stateful Session Bean Example
```
@Stateful(name="BidderAccountCreator")                               |#1
public class BidderAccountCreatorBean
        implements BidderAccountCreator {
    @Resource(name="jdbc/ActionBazaarDS")
    private DataSource dataSource;

    private LoginInfo loginInfo;                                      |#2
    private BiographicalInfo biographicalInfo;                        |#2
    private BillingInfo billingInfo;                                  |#2

    private Connection connection;

    public BidderAccountCreatorBean () {}

    @PostConstruct                                                    |#3
    @PostActivate                                                     |#4
    public void openConnection() {
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    try {
        connection = dataSource.getConnection();
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}

public void addLoginInfo(LoginInfo loginInfo) {
    this.loginInfo = loginInfo;
}

public void addBiographicalInfo(
        BiographicalInfo biographicalInfo) {
    this.biographicalInfo = biographicalInfo;
}

public void addBillingInfo(BillingInfo billingInfo) {
    this.billingInfo = billingInfo;
}

@PrePassivate                                                    |#5
@PreDestroy                                                      |#6
public void cleanup() {
    try {
        connection.close();
        connection = null;
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}

@Remove                                                          |#7
public void cancelAccountCreation() {
    loginInfo = null;
    biographicalInfo = null;
    billingInfo = null;
}

@Remove                                                          |#7
public void createAccount() {
    try {
        Statement statement = connection.createStatement();
        statement.execute(
            "INSERT INTO bidder(" +
                "username, " +
                ...
                "first_name, " +
                ...
                "credit_card_type, " +
                ...
                ") VALUES (" +
                "'" + loginInfo.getUsername() + "', " +
                ...
                "'" + biographicalInfo.getFirstName() + "', " +
                ...
                "'" + billingInfo.getCreditCardType() + "', " +
                ...

                ")");
        statement.close();
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
}
...
@Remote
public interface BidderAccountCreator implements Remote {
    void addLoginInfo(LoginInfo loginInfo);
    void addBiographicalInfo(BiographicalInfo biographicalInfo);
    void addBillingInfo(BillingInfo billingInfo);
    void cancelAccountCreation();
    void createAccount();
}
```

(annotation) <#1 Mark POJO Stateful>
(annotation) <#2 Stateful instance variables>
(annotation) <#3 Post construct callback>
(annotation) <#4 Post activate callback>
(annotation) <#5 Pre passivate callback>
(annotation) <#6 Pre destroy callback>
(annotation) <#7 Remove methods>

Briefly scanning Listing 3.2, we will note its major features right now. As we mentioned earlier, it should not surprise you that the code has a lot in common with the Stateless Session Bean code in Listing 3.1.

> Note that, as before, we are using JDBC for simplicity in this example because we want you to focus on the Session Bean code right now and not JPA. We'll cover JPA in the Third part of the book. An interesting exercise for you is to refactor this code using JPA and notice the radical improvement over JDBC!

We are using the `@Stateful` annotation to mark the `BidderAccountCreatorBean` POJO#1. Other than the annotation name, this annotation behaves exactly like the `@Stateless` annotation so we will not mention it any further. The bean implements the `BidderAccountCreator` remote business interface. As per Stateful Bean programming rules, the `BidderAccountCreatorBean` has a no argument constructor.

Just like in Listing 3.1, a JDBC data source is injected using the @Resource annotation. Both the PostConstruct#3 and PostPassivate#4 callbacks prepare the bean for use by opening a database connection from the injected data source. On the other hand, both the PrePassivate#5 and PreDestroy#6 callbacks close the cached connection.

The `loginInfo`, `biographicalInfo` and `billingInfo` instance variables are used to store client conversational state across business method calls#2. Each of the business methods models a step in the account creation workflow and incrementally populates the state instance variables. The workflow is terminated when the client invokes either of the `@Remove` annotated methods#7.

There is no real point to repeating the coverage of the features that are identical to the ones for Stateless Session Beans, so we will avoid doing so. However, we will cover the specific features unique to Stateful Session Beans next, starting with the Stateful Bean business interfaces.

3.3.3 Business Interfaces for Stateful Beans

Specifying Stateful Bean business interfaces works in almost exactly the same way as it does for Stateless Beans with a couple exceptions. Stateful Session Beans support local and remote invocation

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

through the `@Local` and `@Remote` annotations. However, a Stateful Session Bean *cannot* have a Web Service Endpoint Interface (SEI). This is because SOAP based Web Services are inherently stateless in nature. Also, we should always include at least one `@Remove` annotated method in our Stateful Bean's business interface. The reason for this will become clear as we discuss the Stateful Bean lifecycle next.

3.3.4 Stateful Bean Lifecycle Callbacks

As we mentioned in section 3.1, the lifecycle of the Stateful Session Bean is very different from that of a Stateless Session Bean, because of passivation. In this section, we will explain this concept in more depth. Let's start by looking at the lifecycle of a Stateful Bean, as depicted in Figure 3.7. It can be summarized as follows--the container:

1. Always creates new bean instances using the default constructor whenever a new client session is started.
19. Injects resources.
20. Stores the instance in memory.
21. Executes the requested business method invoked through the business interface by the client.
22. Waits for and executes subsequent client requests.
23. If the client remains idle for a period of time, the container *passivates* the bean instance. Essentially, passivation means that the bean is moved out of active memory, serialized and stored in temporary storage.
24. If the client invokes a passivated bean, it is activated (brought back into memory from temporary storage).
25. If the client does not invoke a passivated bean instance for a period of time, it is destroyed.
26. If the client requests the removal of a bean instance, it is first activated if necessary and then destroyed.

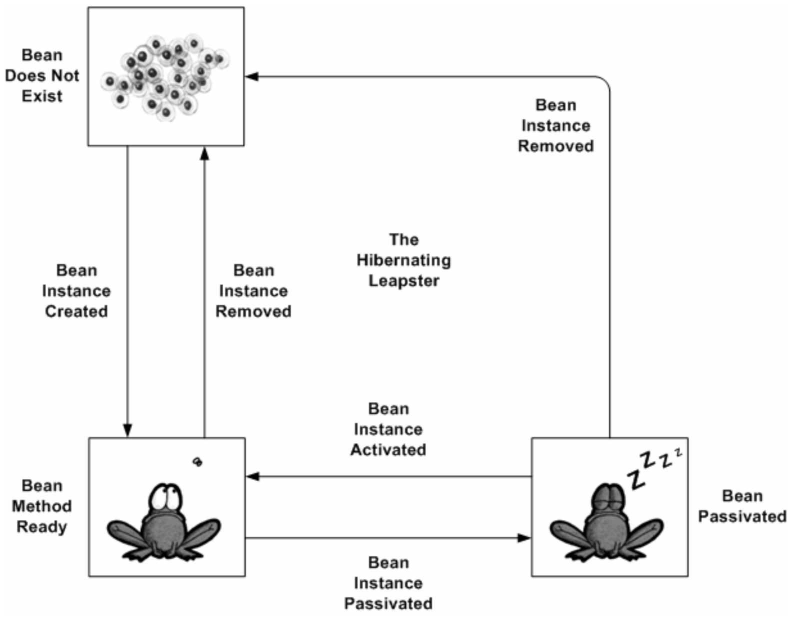


Figure 3.7: The lifecycle of a Stateful Session bean. A Stateful bean maintains client state and cannot be pooled. They may be passivated when the client is not using it and must be activated when client needs it again.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Like a Stateless Session Bean, the Stateful Session Bean has lifecycle callback methods, or callbacks, that are invoked when the `PostConstruct` event occurs, as an instance is created, and callbacks that are invoked when the `PreDestroy` event occurs, before it is destroyed. But now we have two new callback events that we can have callbacks for: PrePassivate and PostActivate, that are part of the passivation process. We will discuss them next.

Just as we do in Listing 3.1, we use a `PostConstruct` callback in Listing 3.2 to open a database connection from the injected data source so that it can be used by business methods. Also like Listing 3.1, we close the cached connection in preparation for bean destruction in a `PreDestroy` callback. However, the curious thing you should note is that we invoke the very same method for both the `PreDestroy` and `PrePassivate` callbacks:

```
@PrePassivate
@PreDestroy
public void cleanup() {
    ...
}
```

Similarly, the exact same action is taken for both the `PostConstruct` and `PostActivate` callbacks:

```
@PostConstruct
@PostActivate
public void openConnection() {
    ...
}
```

To see why this is the case, lets discuss activation and passivation in a little more detail.

Passivation and Activation

As we noted, if clients don't invoke a bean for a long enough time, it is really not a good idea to continue keeping it in memory. For a large number of beans, this could easily make the machine run out of memory. The container employs the technique of *passivation* to save memory when possible.

> Passivation essentially means saving a bean instance into disk instead of holding it in memory. The container accomplishes this task by serializing the entire bean instance and moving into permanent storage like a file or the database. *Activation* is the opposite of passivation and is done when the bean instance is needed again. The container activates a bean instance by retrieving it from permanent storage, deserializing it and moving it back into memory. Note this means that all bean instance variables that you care about and should be saved into permanent storage must either be a Java primitive or implement the `java.io.Serializable` interface.

The point of the pre-passivation callback is to give the bean a chance to prepare for serialization. This may include copying non-serializable variable values into `Serializable` variables and clearing unneeded data out of serializable variables to save total disk space needed to store the bean. Most often the pre-passivation step consists of releasing heavy-duty resources such as open database, messaging server and socket connections that cannot be serialized. A well-behaved bean should ensure that heavy-duty resources are both closed *and* explicitly set to null before passivation takes place.

From the perspective of a bean instance, there really isn't very much of a difference between being passivated and being destroyed. In both cases, the current instance in memory would cease to exist. As a result, in most cases, you will find that the same actions are performed for both the `PreDestroy` and `PrePassivate` callbacks, as we do in Listing 3.2. Pretty much the same applies

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

for the `PostConstruct-PostActivate` pair. For both callbacks, the bean needs to do whatever is necessary to get ready to service the next incoming request. Nine times out of ten, this means getting hold of resources that are either not instantiated or were lost during the serialization/deseralization process. Again, Listing 3.2 is a good specimen since the `java.sql.Connection` object cannot be serialized and must be reinstantiated during activation.

Destroying a Stateful Session Bean

In Listing 3.2, the `cancelAccountCreation` or `createAccount` methods are marked with the `@Remove` annotation. Beyond the obvious importance of these methods in implementing vital workflow logic, these methods play an important role in maintaining application server performance. Calling business methods marked with the `@Remove` annotation signifies a desire by the client to end the session. As a result, invoking these methods triggers immediate bean destruction.

To gain an appreciation for this feature, consider what would happen if it did not exist. If remove methods did not exist, the client would have no way of telling the container when a session should be ended. As a result, every Stateful Bean instance ever created will always have to be timed-out into being passivated (if the container implementation supports passivation) and timed-out again to be finally destroyed. In a highly concurrent system, this could have a drastic performance impact. The memory footprint for the server would constantly be artificially high, not to mention the wasted CPU cycles and disk space used in the unnecessary activation/passivation process. This is why it is critical that we remove Stateful Bean instances when the client is done with its work instead of relying on the container to destroy them when they time out.

Believe it or not, these are the only few Stateful Bean specific features that we needed to talk about! Before concluding this section on Stateful Beans, we will briefly summarize the differences between Stateful and Stateless Session Beans as a handy reference in Table 3.2.

1.5 Table 3.2: The main differences between Stateless and Stateful Session beans.

Features	**Stateless**	**Stateful**
Conversational state	No	Yes
Pooling	Yes	No
Performance Problems	Unlikely	Possible
Lifecycle events	PostConstruct, PreDestroy	PostConstruct, PreDestroy, PrePassivate, PostActivate
Timer (discussed in Chapter 5)	Yes	No
SessionSynchronization for Transactions (discussed in Chapter 6)	No	Yes
Web Services	Yes	No
Extended PersistenceContext (discussed in Chapter 9)	No	Yes

Thus far we have explored how to develop Session Beans. In the next section, we are going to discuss how Session Beans are actually accessed and used by clients.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

3.4 Session Bean Clients

A session bean works for a client and may either be invoked by local clients co-located in the same JVM or by remote client outside the JVM. In this section we will first discuss how can a client accesses a session bean and then see the usage of `@EJB` annotation to inject session bean references.
Almost any Java component can be a Session Bean client. POJOs, Servlets, JSPs or other EJBs can access Session Beans. In fact, Stateless Session Beans exposed through Web Services endpoints can even be accessed by non-Java clients such as .NET applications. However, in this Section, we will concentrate on clients that either access Session Beans locally or remotely through RMI. In Chapter 15 we will see how EJB Web Service clients look like.

Fortunately, in EJB 3.0, accessing a remote or local Session Bean looks exactly the same. As a matter of fact, other than method invocation patterns, Stateless and Stateful Session Beans pretty much look alike from a client's perspective too. In all of these cases, a Session Bean client follows these general steps to use a Session Bean:

1. The client obtains a reference to the Beans directly or indirectly from JNDI.
27. All Session Bean invocations are made through an interface appropriate for the access type.
28. The client makes as many method calls as is necessary to complete the business task at hand.
29. In case of a Stateful Session Bean, the last client invocation should be a remove method.

In order to keep things as simple as possible, we will explore a client that uses the `BidManagerBean` Stateless Session Beans to add a bid to the ActionBazaar site. We will leave it as a thought exercise for you to extend the client code to use the `BidderAccountCreatorBean` Stateful Session Bean. For starters, let us take a look at how the code to use the `BidManagerBean` from another EJB might look like:

```
@Stateless
public class GoldBidderManagerBean implements GoldBidderManager {
    @EJB
    private BidManager bidManager;

    public void addMassBids(List<Bid> bids) {
        for (Bid bid : bids) {
            bidManager.addBid(bid);
        }
    }
}
```

The preceding code uses dependency injection through the `@javax.ejb.EJB` annotation to get a reference to the `BidManagerBean`. This is by far the easiest method of obtaining a reference to a Session Bean. Depending on your client environment, you might have to use one of the two other options available for obtaining EJB references: using EJB context lookup or using JNDI lookup. Since neither of these options is used too often in real life, we'll focus on DI for right now. However, we'll discuss both EJB context lookup and JNDI lookup further in coming Chapters as well as the Appendix.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

3.4.1 Using the @EJB Annotation

Recall from our discussion on DI in Chapter 2 that the `@EJB` annotation is specifically intended for injecting Session Beans into client code. Also recall that since injection is only possible within managed environments, this annotation only works inside another EJB, in code running inside an Application-Client Container (ACC) or in components registered with the web container (such as a Servlet or JSF backing bean). The following is the specification for the `@EJB` annotation:

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface EJB {
    String name() default "";
    Class beanInterface() default Object.class;
    String beanName() default "";
}
```

All three of the parameters for the `@EJB` annotation are optional. The `name` element suggests the name used to identify the EJB to be injected from JNDI. The `beanInterface` specifies the business interface to be used to access the EJB. The `beanName` element allows us to distinguish between EJBs if multiple EJBs implement the same business interface. In our `GoldBidManagerBean` code, we chose to use the remote interface of the `BidManagerBean`. If we wanted to use the local interface of the `BidManagerBean` EJB instead we can use the following:

```
@EJB
private BidManagerLocal bidManager;
```

We have not specified the `name` parameter for the `@EJB` annotation in the preceding code and the JNDI name is derived from the interface name (`BidManagerLocal` in our case). If we want to inject an EJB bound to a different JNDI name we can use the `@EJB` annotation as follows:

```
@EJB(name="BidManagerRemote")
private BidManager bidManager;
```

3.4.2 Injection and Stateful Session Beans

For the most part, using DI is a no-brainer. There are a few nuances to keep an eye on while using DI with Stateful Beans, though. You can inject a Stateful Session into another Stateful Session Bean instance if you need to. For example you can inject the `BidderAccountCreator` Stateful EJB from `UserAccountRegistration` EJB that is another Stateful Session Bean as follows:

```
@Stateful
public class UserAccountRegistrationBean
        implements UserAccountRegistration {
    @EJB
    private BidderAccountCreator bidderAccountCreator;
    ...
}
```

The above code will create an instance of `BidderAccountCreatorBean` that will be specifically meant for the client accessing the instance of the `UserAccountRegistrationBean`. If the client removes the instance of `UserAccountRegistrationBean` the associated instance of `BidderAccountCreatorBean` will also be automatically removed.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Note that you must not inject a Stateful Session Bean into a stateless object such as Stateless Session Bean or Servlet that may be shared by multiple concurrent clients (you should use JNDI in such cases instead). However injecting an instance of a Stateless Session Bean into Stateful Session Bean is perfectly legal.

We will discuss how you can use EJB from other tiers in much greater detail in Chapter 12.

This concludes our brief discussion on accessing Session beans. Next, we'll briefly discuss potential performance issues of Stateful Session Beans.

3.5 Performance Considerations for Stateful Beans

Rightfully or wrongfully, Stateful Session Beans have had a bad rap for being performance bottlenecks. There is truth behind this perception, quite possibly due to poor initial implementations for most popular application servers. In recent years, these problems have been greatly alleviated with effective under the hood optimizations as well as better JVM implementations. However, there are still a few things to keep in mind in order to use Session Beans effectively. More or less, these techniques are essential for using any Stateful technology, so pay attention even if you decide against using Stateful Beans. In this section we'll familiarize you with the techniques to effectively use stateful session beans and the other alternatives for building stateful applications.

3.5.1 Using Stateful Session Beans effectively

There is little doubt that Stateful Session Beans provide extremely robust business logic processing functionality if maintaining conversational state is an essential application requirement. In addition, EJB 3.0 adds extended persistence contexts specifically geared toward Stateful Session Beans (discussed in Chapter 9 and 13), significantly increasing its power. Most popular application servers such as WebSphere, WebLogic, Oracle and JBoss provide high availability by clustering EJB containers running the same Stateful Bean. A clustered EJB container replicates session state across container instances. If a clustered container instance crashes for any reason the client is routed to another container instance seamlessly without losing state. Such reliability is hard to match without using Stateful Session Beans. Nonetheless there are a few things to watch out for while using Stateful Session Beans.

Choosing session data appropriately

Stateful Session Beans can become resource hogs causing performance problems if not used properly. Since the container stores session information in memory, if you have thousands of concurrent clients for your Stateful Session Bean you may run out of memory or cause a lot of *disk thrashing* by the container as it passivates and activates instances to try to conserve memory. As a result, you have to closely examine what kind of data you are storing in the conversation state and make sure the total memory footprint for the Stateful Bean is as small as possible. For example, it may be a lot more efficient to store just the `itemId` for an `Item` instead of storing the complete `Item` object in an instance variable.

If we cluster Stateful Beans the conversational state is replicated between different instances of the EJB container. State replication uses network bandwidth. Storing a large object in the bean state

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

may have a significant impact on the performance of your application because the containers will spend time replicating objects to other container instances to ensure high availability.

Passivating and removing beans

The rules for passivation are generally implementation specific. Improper use of passivation policies (when passivation configuration is an option) may cause performance problems. For example, the Oracle Application Server has several rules to passivate bean instances such as the expiration of idle time for a bean instance, reaching maximum number of active bean instances allowed for a Stateful Session Bean or when the threshold for JVM memory is reached. You have to check the documentation for your EJB container and appropriately set passivation rules. For example, if we set the maximum number of active instances allowed for a Stateful Bean instance to be 100 and we usually have 150 active clients, the container will keep on passivating and activating bean instances causing performance problems.

You can go a long way to solving potential memory problems by explicitly removing the bean instances no longer required instead of depending on the container to time them out. As discussed earlier you can annotate a method with the `@Remove` annotation that signals the container to remove the bean instance.

Given the fact that Stateful Session Beans can become performance bottlenecks whether through improper usage or under certain circumstances, it is worth inspecting the alternatives to using them.

3.5.2 Stateful Session Bean Alternatives

Here are a few alternative strategies to implementing stateful business processing, as well as some issues you may need to consider when using them:

The first alternative to Stateful Beans is replacing them with a combination of persistence and stateless processing. In this scheme, we essentially move state information from memory to the database on every request.

You should carefully examine whether you really want to maintain state between conversations in memory. This is completely based on the application requirements and how much tolerance of failure you have. For example, in the `BidderAccountCreator` EJB you can probably avoid the use of conversational state by not maintaining instance variables to store the user information in memory and save data in the database on each method call.

Secondly you may choose to build some mechanism at the client side to maintain state. This requires additional coding such as storing the state as an object in client memory or file.

The downside of these two approaches is that it is very difficult to guarantee high availability with these and they may not be viable options for your application. In fact, you would lose all of the advantages that the container provides by hand-coding proprietary solutions such as the ones outlined above, including automated passivation and robust, transparent state maintenance.

Thirdly you may choose to maintain session state in the web container if you are building a web application. Although HTTP is a stateless protocol, the Java Servlet API provides the ability to maintain state by using the `HttpSession` object. The Servlet container does not have to do heavy

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

lifting like passivation and activation and may perform better in certain situations. Be aware that too much data in the `HttpSession` could decrease performance of the Servlet container as well, so this is not a silver bullet either. Moreover, you cannot use this option with thick or Java SE clients.

So when you really need to maintain state in your applications and your clients are Java SE clients then the first two options we discussed earlier may be harder to implement hence Stateful Session Beans are probably the only viable option as long as you carefully weigh the performance considerations we outlined above.

Hopefully it was a smooth ride to Session beans and we are almost at the end of training track. We will close by outlining some best practices for session beans that you can use to ramp yourself up to build the business logic of your applications with the knowledge you gathered so far.

3.6 Session Bean Best Practices

In this brief Section we will outline some of the best practices for Session beans that you can use while building the business logic tier for your application.

Choose your bean type carefully. Stateless Session beans will be suitable most of the time. Carefully examine whether your application needs Stateful Session Beans because it comes with a price. If the EJB client lies in the web tier then using the `HttpSession` may be a better choice than Stateful Session Beans under some circumstances.

Carefully examine interface types for Session Beans. Remote interfaces involve network access and may slow down your applications. If the client will always be used within the same JVM as the bean, then a local interface should be used.

If you are using DI make sure you don't inject a Stateful Session Bean into a Stateless Session Bean or Servlet. Injected EJB instances are stored in an instance variable and are available globally for subsequent clients even if a Stateless Bean instance is returned to the pool and an injected Stateful Bean instance may contain inaccurate state information that will be available to a different client. It's legal to inject a Stateful Bean instance to another Stateful Session Bean or an application client.

Separate cross cutting concerns such as logging and auditing using business interceptors that we discuss in Chapter 5 instead of spreading these all over the business logic.

Closely examine what kind of data you are storing in the conversation state. Try to use small, primitive instance variables in a Stateful Bean whenever possible as opposed to large nested composite objects.

Don't forget remove methods in a Stateful Session Bean.

Tune passivation and timeout configuration to find optimal values for your application.

These brief pointers on using Session Beans brings us to the end of this Chapter! We'll finish things off by recapping what we covered and telling you what's coming next.

3.7 Summary

In this chapter, you learned the Session Bean types and how Stateless Session Beans and Stateful Session Beans differ. We looked at the programming rules for both Stateless and Stateful Session Beans and built comprehensive examples of both bean types. Stateless Session beans have a very simple lifecycle and can be pooled. Stateful Beans require instances for each client and for that the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

reason they may take up a lot of resources. In addition, passivation and activation of Stateful Beans may impact performance if used inappropriately. There are alternatives for Stateful Session Beans that you should consider and make an educated choice. Session Bean clients can either be local or remote. Dependency injection makes the use of EJB simpler instead of making complex JNDI lookups. Finally we provided some best practices for developing Session Beans.

At this point you have all the ammunition necessary to build the business logic of your application using Stateless and Stateful Session Beans. In the next Chapter we will discuss how you can build messaging applications with Message Driven beans.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 4: Messaging and Developing Message Driven Beans

In this chapter we will take a closer look at developing Message Driven Beans as well as overview the concepts and technologies these powerful EJB 3 components build on. We will first introduce you to basic messaging concepts. We will then explore the premier Java messaging, JMS (Java Message Service.) by creating a message producer. Finally, we will take a look at Message Driven Beans (MDB), the EJB 3 answer to messaging.

There are two essential reasons to gain an understanding of messaging and JMS before diving into MDB. First, most MDBs you will encounter are glorified JMS message consumers implementing JMS interfaces (such as *javax.jms.MessageListener*) and using JMS components (such as *javax.jms.Message*). Secondly, for most solutions with MDB, your messaging will involve much more than simply consuming messages. For the simplest of these tasks, such as sending messages, you will have to understand JMS. This chapter assumes that you have familiarity with JMS and we briefly discuss JMS.

If you are pretty comfortable with Messaging and JMS, feel free to skip to the sections on MDB. It is good to reinforce what you know from time to time though, so you just might want to quickly jog though first few sections with us anyway.

4.1 Messaging Concepts

When we talk about messaging in the Java EE context, what we really mean is the process of *loosely coupled, asynchronous communication* between system components. Most communication between components is synchronous, such as simple method invocation or Java RMI (Remote Method Invocation). In both cases, the invoker and the invocation target have to be currently present for the communication to succeed. Synchronous communication also means that the invoker must wait for the target to complete the request for service before proceeding further.

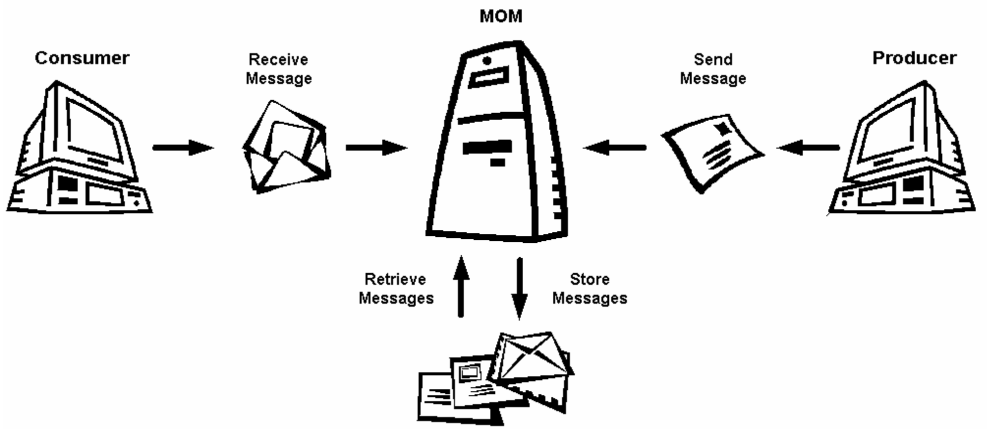
As an example from real-life, we are communicating synchronously when we (the invoker) call and talk to someone over the phone. But what if the person (the invocation target) is not available? If possible we could leave a message. The answering machine would make the communication asynchronous by storing your message so that the receiver could listen to it later and respond. Message Oriented Middleware (MOM) enables messaging in almost exactly the same way that an answering machine does – by acting as the middleman between a message sender and receiver so that they do not have to be available simultaneously. In this section we will briefly introduce Message Oriented Middleware, will do a quick review of use of messaging in ActionBazaar and examine popular messaging models.

4.1.1 Message Oriented Middleware (MOM)

Message Oriented Middleware is software that enables asynchronous messages between system components. When a message is sent to MOM it stores the message in a location specified by the

Licensed to John Sweitzer <admin@saolailaem.info>

sender and acknowledges receipt immediately. The message sender is called a *producer* and the location where the message is stored is called a *destination*. At a later point in time, any software component interested in messages at that particular destination can retrieve currently stored messages. The software components receiving the messages are called the message *consumers*. Figure 4.1 depicts different components of MOM.



31 **Figure 4.1: Basic MOM message flow. When the producer sends a message to the MOM, it is stored immediately and later collected by the consumer. Looks a lot like email, doesn't it?**

32

MOM is not a new concept by any means. MOM products include IBM WebsphereMQ, Tibco Rendezvous, SonicMQ, ActiveMQ, and Oracle Advanced Queuing giving it a vibrant market.

To flush out messaging concepts a bit more, let us explore a problem in the ActionBazaar application. We will continue working on this problem as we progress through the chapter.

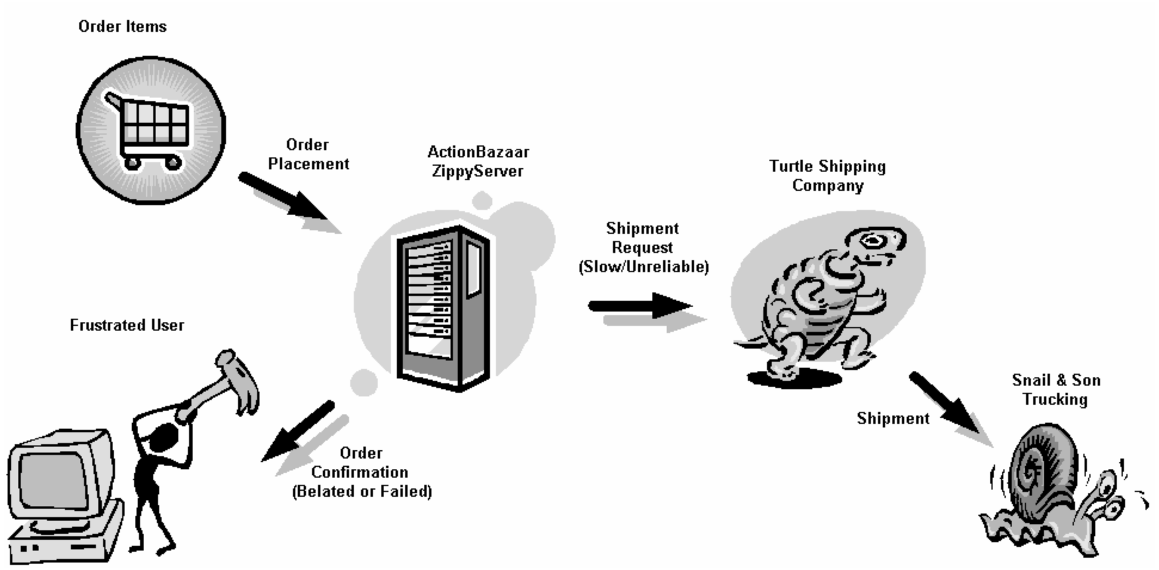
4.1.2 Messaging in ActionBazaar

As an additional source of revenue, ActionBazaar will itself list items for bid when the company is able to find good bulk deals though its extensive purchasing network. These items are displayed on the site as "ActionBazaar Specials" and come with complete satisfaction guarantees. ActionBazaar automatically ships these items from their warehouse to winning bidders as soon as they order them. When ActionBazaar started as a two-person Internet operation, Joe and John, the two founders, made a sweet deal with Turtle Shipping Company's founder Dave Turtle. As a part of the deal, Joe and John agreed to ship with Turtle for a few years.

As soon as a user places as order for an "ActionBazaar Special," a shipping request is sent to the Turtle system via a B2B (business-to-business) connection as depicted in Figure 4.2. The order confirmation page is loaded only after Turtle confirms receipt. Now that the number of ActionBazaar customers has gone through the roof, the slow Turtle servers and B2B connection simply cannot keep up and completing a shipping order takes forever. To make matters worse, the Turtle server occasionally goes down, making orders fail altogether.

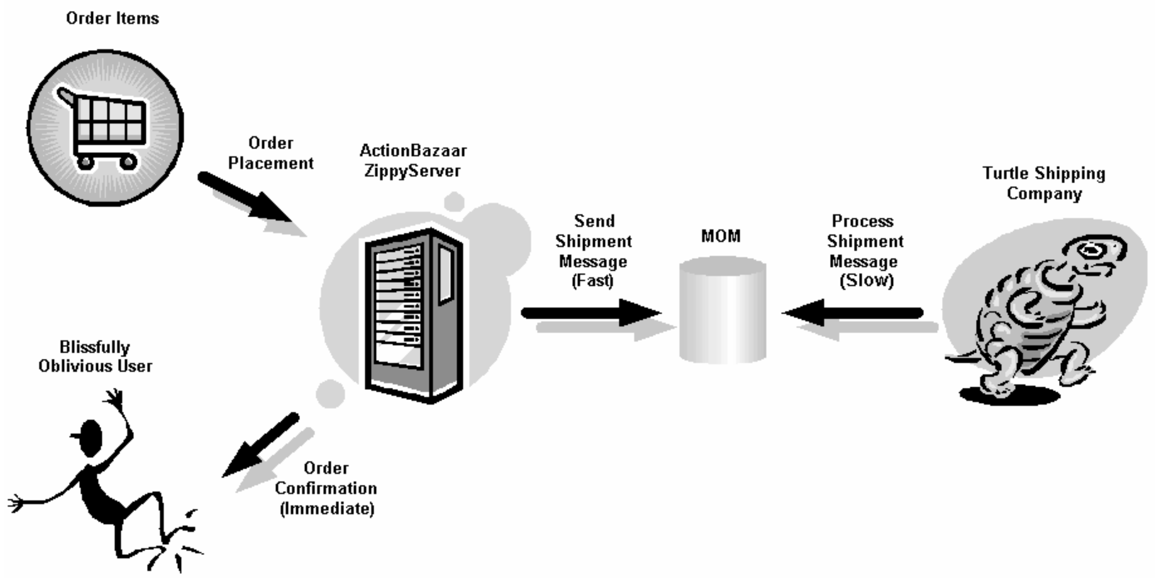
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



33 **Figure 4.2: ActionBazaar ordering before MOM is introduced. Slow B2B processing is causing customer dissatisfaction.**

Taking a closer look at things, we see that we could make the forwarding process of the shipping request asynchronous and solve this problem. Instead of communicating directly with the Turtle server, the ActionBazaar ordering process could send a message containing the shipping request to MOM as depicted in Figure 4.3. As soon as the message is stored in MOM, the order can be confirmed without making the user wait. At a later point in time, the Turtle server could request pending shipping request messages from the MOM and process them at its own pace.


Order Items
Order Placement
ActionBazaar ZippyServer
Turtle Shipping Company
Send Shipment Message (Fast)
MOM
Process Shipment Message (Slow)
Blissfully Oblivious User
Order Confirmation (Immediate)

34 **Figure 4.3: ActionBazaar ordering after MOM is introduced. Messaging enables both fast customer response times and reliable processing.**

In this case, the most obvious advantage MOM is offering is increasing reliability. The reliability stems from not insisting that both the ActionBazaar and Turtle servers be up and running at the same time and also not insisting that they function at the same processing rate. In the most extreme case,

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

even if the Turtle server is down at any given time, the shipping request is not lost and is just delivered later. Another significant advantage of messaging that might not be obvious is loosely coupled system integration. We could, if we wanted to, easily switch from the Turtle Shipping Company to O'Hare Logistics once the current contract runs out. Note how different this is from having to know the exact interface details of the Turtle servers in case of synchronous communication technologies like RMI or even remote Session Beans.

So far we have described a particular form of messaging named 'point-to-point' to explain basic messaging concepts. This is a good time to move away from this simplification and fully discuss messaging models.

4.1.3 Messaging Models

A 'messaging model' is simply a particular way of messaging and number of senders, consumers involved. It will be more obvious what this means as we describe each model. There are two popular messaging models standardized in Java EE -- Point-to-Point messaging and Publish-Subscribe messaging. We will discuss each of these messaging models next.

Point-to-Point (PTP)

You can probably guess from the names of the messaging models how they function. In the PTP scheme, a single message travels from a single producer (point A) to a single consumer (point B). PTP message destinations are called *queues*. It is important to note that PTP does not guarantee that messages are delivered in any particular order -- the name 'queue' is more symbolic than anything else. Also, if there is more than one potential receiver for a message, a random receiver is chosen, as depicted by Figure 4.4. The classic message in a bottle story is a good analog of PTP messaging. The message in a bottle is set afloat by the lonely castaway (the producer). The ocean (the queue) carries the message to an anonymous beach dweller (the consumer) and the message can only really be "found" once.

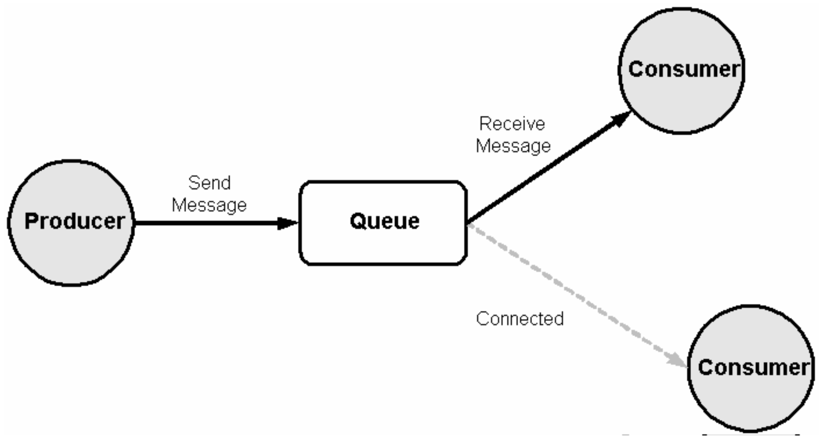


Figure 4.4: The PTP messaging model with one producer and two consumers.

The ActionBazaar shipping request forwarding problem is an excellent candidate for the PTP model, as we want to be guaranteed that the message is received once and only once.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Publish-Subscribe (Pub-Sub)

Publish-Subscribe messaging is much like posting to an Internet newsgroup. As shown in Figure 4.5, a single producer produces a message that is received by any number of consumers that happen to be connected to the destination at the time. To make the likeness to Internet postings even closer, the message destination in this model is called a *topic* and a consumer is called a *subscriber*.

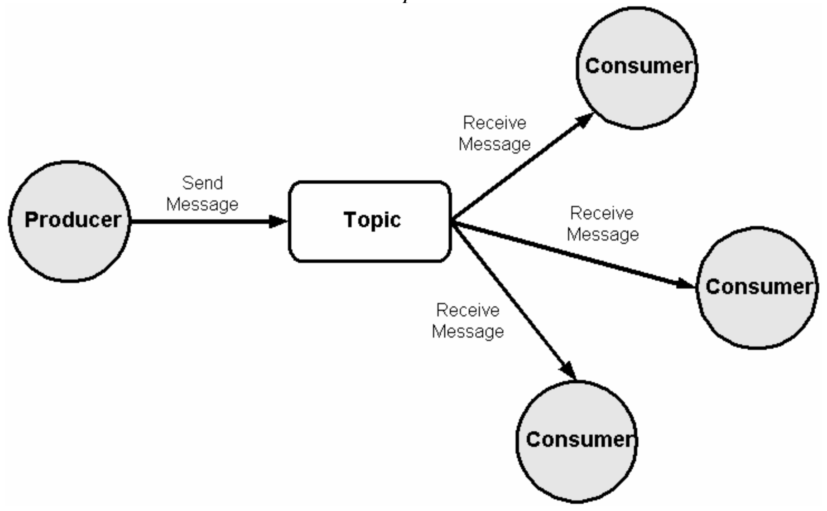


Figure 4.5: The publish-subscribe messaging model with one producer and three consumers. Each topic subscriber receives a copy of the message.

Pub-sub messaging works particularly well in broadcasting information across systems. For example, it could be used to broadcast a system maintenance notification several hours before an outage to all premium sellers whose systems are directly integrated with ActionBazaar and are listening at the moment.

The Request-Reply Model

In the ActionBazaar example, you might want a receipt confirmation from Turtle once they get the shipping request you sent to the queue.

A third kind of model called request-reply comes in handy in these kinds of situations. In this model, we give the message receiver enough information so that they might "call us back". This model is an "overlay" model because it is typically implemented on top of either the PTP or pub-sub models.

For example, in the PTP model, the sender specifies a queue to be used to send a reply back to (in JMS, this is called the 'reply to' queue) as well as a unique ID shared by both the outgoing and incoming messages ('correlation ID' in JMS). The receiver receives the message and sends a reply to the reply queue, copying the correlation ID. The sender receives the message on the reply queue and figures out what message received a reply by matching the correlation ID.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

At this point, you should have a good conceptual foundation of messaging and are perhaps be eager to get a taste of some code. Next, we will take a brief look at JMS and actually implement the ActionBazaar message producer for sending the message.

4.2 Introducing Java Messaging Service

In this section we will provide an overview to JMS API by building a simple message producer.

JMS is a deceptively simple and small API to a very powerful technology. The JMS API is to messaging what the Java Database Connectivity (JDBC) API is to database access. JMS provides a uniform, standard way of accessing MOM in Java and is therefore an alternative to using product-specific APIs. With the exception of Microsoft Message Queue (MSMQ), most major MOM products support JMS.

The easiest way to learn JMS might be by looking at code in action. We are going to explore JMS by first developing the ActionBazaar code that sends out the shipping request. In this section we will develop a message producer using JMS and learn about structure of Message interface and then in the next section, we will develop the message consumer using MDB.

4.2.1 Developing the JMS Message Producer

As we described in our scenario in section 4.1.2, when a user places an order for an "ActionBazaar Special", a shipping request is sent to a queue shared between ActionBazaar and Turtle. The code in Listing 4.1 sends the message out and could be part of a method in a simple Java Object invoked by the ActionBazaar application. All relevant shipping information such as the item number, shipping address, shipping method and insurance amount is packed into a message and sent out to 'ShippingRequestQueue'.

12 Listing 4.1: JMS Code to Send Out Shipping Request from ActionBazaar

```
@Resource(name="jms/QueueConnectionFactory")
 private ConnectionFactory connectionFactory;   |#1


@Resource(name="jms/ShippingRequestQueue")
 private Destination destination;                |#2


Connection connection = connectionFactory.createConnection();        |#3
Session session = connection.createSession(true,                     |#4
    Session.AUTO_ACKNOWLEDGE);

MessageProducer producer = session.createProducer(destination);      |#5

ObjectMessage message = session.createObjectMessage();               |#6
ShippingRequest shippingRequest = new ShippingRequest();             |#7
shippingRequest.setItem(item);                                       |#7
shippingRequest.setShippingAddress(address);                         |#7
shippingRequest.setShippingMethod(method);                           |#7
shippingRequest.setInsuranceAmount(amount);                          |#7
message.setObject(shippingRequest);                                  |#8

producer.send(message);                                             |#9
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
session.close();                                                    |#10
connection.close();                                                 |#10
```

(annotation) <#1 Inject Connection Factory>
(annotation) <#2 Inject Destination>
(annotation) <#3 Connecting to MOM>
(annotation) <#4 Creating Session>
(annotation) <#5 Creating A Producer>
(annotation) <#6 Creating Message>
(annotation) <#7 Creating PayLoad>
(annotation) <#8 Setting PayLoad>
(annotation) <#9 Sending Message>
(annotation) <#10 Cleaning up>

As we explain each logical step of this code in the following sections, we will go through a large subset of the JMS API components and see usage patterns.

Retrieving the ConnectionFactory and Destination

JMS has a concept called *administered objects* that is very similar to `JDBC javax.sql.DataSource` Objects. These are resources that are created and configured outside code and stored in JNDI. JMS has two administrative objects, `javax.jms.ConnectionFactory` and `javax.jms.Destination`, both of which we use in Listing 4.1. The connection factory we then retrieved using dependency injection with `@Resource` annotation and it encapsulates all configuration information needed to connect to the MOM#1. We also inject the queue to forward the shipping request to, aptly named 'ShippingRequestQueue' #2. With EJB 3.0, using resources is much easier and you do not have to deal with complexity of JNDI and configuring resource references in deployment descriptors. We will discuss more about dependency injection in Chapter 5.

The next step in Listing 4.1 is creating a connection to the MOM and getting a new JMS session.

Opening the Connection and Session

The `javax.jms.Connection` object represents a live MOM connection, which we create using the `createConnection()` method of the connection factory#3 (in Listing 4.1). Connections are thread-safe and designed to be sharable because opening a new connection is resource intensive. A JMS session (`javax.jms.Session`) on the other hand, provides a single-threaded, task-oriented context for sending and receiving messages. We create a session from the connection using the `createSession` method (#4 in listing 4.1). The first parameter of the method specifies if the session is transactional. We have decided that our session should be transactional and set the parameter to *true*. This means that the requests for messages to be sent will not be realized until either the session's `commit()` method is called or the session is closed. If the session were not transactional, messages would be sent as soon as the `send` method is invoked. The second parameter of the `createSession` method specifies the acknowledge mode and only has an effect for non-transactional sessions receiving messages, which we will discuss later. Having set up the session, we are now ready to take on the meat of the matter: sending the message.

Licensed to John Sweitzer <admin@saolailaem.info>

Preparing and Sending the Message

The session is not directly used for sending or receiving messages[4].Instead, a javax.jms.MessageProducer – needed to send messages to the shipping request queue is constructed using the session's createProducer method in listing 4.1#5. Then #6, #7 and #8 create and populate the javax.jms.Message to be sent. In our example, we send the Serializable Java Object ShippingRequest to Turtle, so the most appropriate message type for us is javax.jms.ObjectMessage, which we create using the createObjectMessage method#6. We then create an instance of the ShippingRequest object and set the item number, shipping address, shipping method and insurance amount fields#7. Once ShippingRequest is set up, we set it as the payload of the message using setObject#8. Finally, we instruct the message producer to send the message out using the send method in listing 4.1#9.

Releasing Resources

A large number of resources are allocated under the hood for both the Session and Connection Objects, so it is very important to explicitly close both once we are finished with them, as we do in #10 (Listing 4.1). Closing the session is even more important in our case since no messages are sent out until our transactional session is committed when we close the session.

If all goes well, a message containing the shipping request winds up in the queue. Before we look at the message consumer code that receives this message, we are going to discuss the javax.jms.Message Object in a little more detail.

4.2.2 The JMS Message Interface

The Message interface standardizes what is exchanged across JMS and is an extremely robust data encapsulation mechanism. As figure 4.6 shows, JMS message has the following parts: the message header, message properties and the message body, each of which is detailed in the sections that follow.

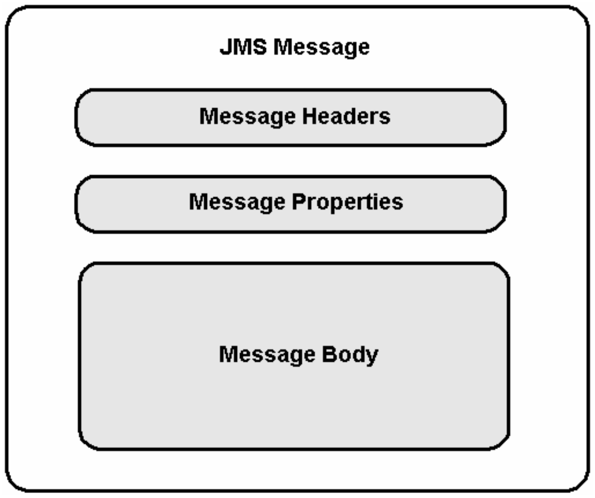


Figure 4.6: The Anatomy of a Message. A JMS message has a header, properties and a body.

An analogy for JMS messages is mailing envelopes. We will see how this analogy fits next.

[4] We could argue that having it do so would simplify the JMS API

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Message Headers

Headers are name-value pairs common to all messages. In our envelope analogy, the message header is the information on an envelope that is pretty standard: the to and from addresses, postage and postmark. For example, the JMS message version of a postmark is the JMSTimestamp header. The MOM sets this header to the current time when the message is sent.

Some other commonly used JMS headers are: `JMSCorrelationID`, `JMSReplyTo` and `JMSMessageID`.

Message Properties

Message properties are just like headers, but are explicitly created by the application instead of being standard across messages. In the envelope analogy, if you decide to write "Happy Holidays" on the envelope to let the receiver know the envelope contains a gift or note, the text is a property instead of a header. In the ActionBazaar example, one way to mark a shipping request as fragile would be to add a boolean property called 'Fragile' and set it to true. The code to do this would look like the following:

```
message.setBooleanProperty("Fragile", true);
```

A property can be a boolean, byte, double, float, int, long, short, String or Object.

Message Body

The message body is the contents of the envelope; it is the payload of the message. What we are trying to send in the body determines what message type we should use. In listing 4.1, we chose `javax.jms.ObjectMessage` because we were sending out the `ShippingRequest` Java Object. Alternatively, we could have chosen to send a `BytesMessage`, `MapMessage`, `StreamMessage` or `TextMessage`. Each of these message types has a slightly different interface and usage pattern. There are no hard and fast rules dictating the choice of message types. You should explore all the choices before taking a decision on what message type to use for your application.

The Spring JMSTemplate

Spring's JmsTemplate greatly simplifies common JMS tasks like sending messages by automating generic code. Using JmsTemplate, our entire message producer code could be reduced to a few lines. This is a great way of getting work done, as long as you are not doing anything too complicated like using temporary queues, JMS headers and properties, etc.

At the time of writing, Spring does not have very robust asynchronous message processing capabilities when compared to MDB. Any future MDB like features in Spring is likely to utilize the relatively arcane JCA container, which leaves room for a great Spring/EJB3 integration case.

Believe it or not, we just finished reviewing most of the major parts of JMS that you need to send and use with MDB. A full coverage of JMS is obviously beyond the scope of this chapter and not

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

really necessary to start discussing Message Driven Beans in the next section. However, we encourage you to fully explore the fascinating JMS API by visiting http://java.sun.com/products/jms/docs.html. In particular, you should explore how JMS message consumers work.

Having taken a closer look at JMS messages, the time is now ripe to look at the Turtle server message consumer built using an MDB.

4.3 Working with Message Driven Beans (MDB)

We will now build on the brief coverage of Message Driven Beans in chapter 1 and 2 and explore in detail what MDBs are, why we should consider using them and how to develop them. We will also discuss some best practices and pitfalls to avoid when developing MDBs.

Put very plainly, Message Driven Beans are EJB components that are designed to consume the asynchronous messages we have been talking about. Although MDBs are designed to handle many different kinds of messages (note the discussion in the sidebar titled "JCA Connectors and Messaging"), we will primarily focus on MDBs that process JMS messages because most enterprise applications use JMS. From this perspective, you might ask why we would need to employ EJBs to handle the task of consuming messages at all when we could use the code we just developed for the JMS message consumer. We will address this question next. We wil soon learn why you would use MDB and its programming rules. We will develop a simple message consumer application using MDB and learn use of @MessageDriven annotation, more about MessageListener interface, activation config properties and MDB lifecycle.

JCA Connectors and Messaging

Although by far JMS is the primary messaging provider for Message Driven beans, as of EJB 2.1, they are not the only one. Thanks to the Java EE Connector Architecture (JCA), MDBs can receive messages from any Enterprise Information System (EIS), such as PeopleSoft HR or Oracle Manufacturing, not just MOMs that support JMS.

Suppose that you have a legacy application that wants to send messages to an MDB. You can do this by implementing a JCA compliant adapter/connector that includes a Message Inflow Contract. Once your JCA resource adapter or connector is deployed to a Java EE container, you can use the Message Inflow Contract to have an asynchronous message delivered to an 'end-point' inside the container. A JCA end-point is essentially the same idea as a JMS destination – it acts as a server proxy to an MDB (a message consumer/listener in JMS terms). As soon as a message arrives at the end point, the container triggers any registered MDBs listening to the end point and delivers the message to it.

For its part, the MDB implements a listener interface that is suitable to the JCA connector/message type and passes appropriate activation configuration parameters to register as a listener of your JCA connector (we will discuss more about message listeners and activation configuration parameters shortly). JCA also enables MOM providers to integrate with Java EE containers in a standardized manner using a JCA compliant connector or resource adapter.
For more information on JCA, visit

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

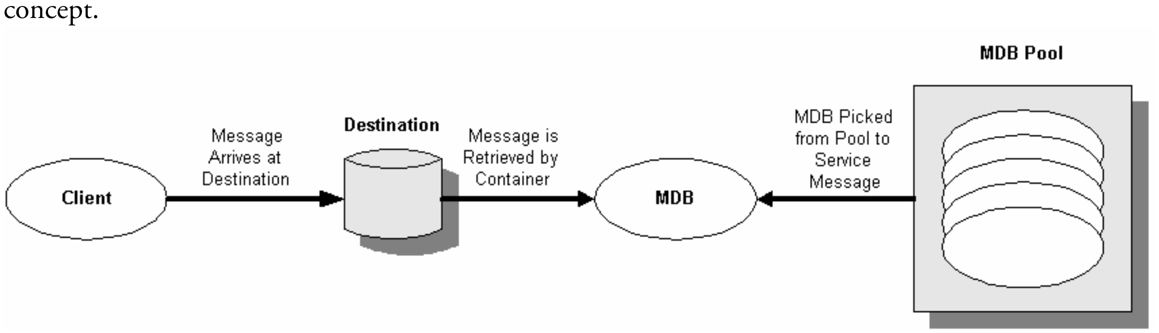
Licensed to John Sweitzer <admin@saolailaem.info>

4.3.1 Why Use Message Driven Beans?

Given the less than stellar reputation of EJB 2.1, it is fair to question the value EJB 3.0. MDBs have to offer. The truth is MDBs have enjoyed a reasonable degree of success even in the darkest hours of EJB. Following are some of the reasons why this is the case and why you should take a serious look at MDBs:

Multithreading

You business application may require multi-threaded message consumers that can process messages concurrently. You can avoid building complexity of building multithreaded by depending upon MDBs because they handle multithreading right out of the box, without any additional code. This is done essentially by managing incoming messages among multiple instances of beans (in a pool) that have no special multithreading code themselves. As soon as a new message reaches the destination, an MDB instance is retrieved from the pool to handle the message as figure 4.5 demonstrates this concept.


MDB Pool
Destination
Message Arrives at Destination
Message is Retrieved by Container
MDB Picked from Pool to Service Message
Client
MDB

37 **Figure 4.5: As soon as a message arrives at the destination, the container retrieves it and assigns a servicing MDB instance from the pool.**

This is popularly known as MDB-pooling that we learn while discussing MDB lifecycle.

Simplified Messaging Code

Other than coding the message consumer (`onMessage)` method, MDBs relieve you from coding the mechanical aspects of processing messages, such as looking up connection factories/destinations, creating connections, opening sessions, creating consumers and attaching listeners. As we will see when we build Turtle message consumer MDB, all of these tasks are handled behind the scenes for you. In EJB 3.0, using sensible defaults for common circumstances even eliminates most of the configuration. In the worst-case scenario, you will have to supply configuration information using simple annotations or through the deployment descriptor.

Starting Message Consumption

If your building a message consumer for the Turtle server message consumer, someone needs to invoke the method in your code in order to start picking up messages from the shipping request queue. In a production environment, it is not clear how this will be accomplished. Doing this through a user-driven manual process obviously is not very desirable. In a server environment, almost every way to execute the method on server startup would be highly system-dependent, not to

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

mention awkward. The same is true about stopping message receipt manually. On the other hand, registered Message Driven Beans would be bootstrapped or torn-down gracefully by the container when the server is started or stopped.

We will continue consolidating these three points as we start investigating a real example of developing MDBs soon. Before we do that, we need to examine the simple rules for developing an MDB.

4.3.2 Programming Rules

Like all EJBs, MDBs are plain Java objects that follow a simple set of rules and sometimes have annotations. We list these rules now before moving on. Do not take these too seriously yet; simply note them in preparation for going through the code-intensive sections that follow:

1. The MDB class must directly (by using the implements keyword in the class declaration) or indirectly (through annotations or descriptors) implement a message listener interface.
2. The MDB class must be concrete. It cannot be either a final or an abstract class.
3. The MDB must be a top-level class and not a subclass of another MDB.
4. The MDB class must be declared public.
5. The bean class must have a no-argument constructor. If you do not any constructors in your java class the compiler will create a default constructor. The container uses this constructor to create a bean instance.
6. We cannot define a `finalize` method in the bean class. If any cleanup code is necessary, it should be defined in a method designated as `PreDestroy`.
7. We must implement the methods defined in the message listener interface. The message listener methods must be public and cannot be static or final.
   - We must not throw the `javax.rmi.RemoteException` or any runtime exceptions. If a `RuntimeException` is thrown, the MDB instance is terminated.

We will apply these rules next in developing our example MDB.

4.3.3 Developing a Message Consumer with MDB

We will now explore how to develop an MDB by reworking the Turtle server JMS message consumer as a MDB. To make the code a bit more interesting, we will actually implement the `processShippingRequest` method mentioned in the JMS code. Listing 4.2 shows the MDB code retrieves shipping requests sent to the queue and saves each request in the Turtle database table named `SHIPPING_REQUEST`. Note that we are using JDBC for simplicity and it gives us ability to demonstrate the MDB lifecycle methods for opening and closing JDBC connections. We recommend that you consider EJB3 Java Persistence API, discussed in part 3 of this book, for persisting your data instead of using straight-JDBC.

13 Listing 4.2: Turtle Server Shipping Request Processor MDB

```
package ejb3inaction.example.buslogic;

import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;
import javax.annotation.PostConstruct;
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
import javax.annotation.PreDestroy;
import javax.annotation.Resource;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import java.sql.*;
import javax.sql.*;

@MessageDriven(                                                     |#1
    name="ShippingRequestProcessor",                                |
    activationConfig = {                                            |
        @ActivationConfigProperty(                                  |
            propertyName="destinationType",                         |
            propertyValue="javax.jms.Queue"),                       |
        @ActivationConfigProperty(                                  |
            propertyName="destinationName",                         |
            propertyValue="jms/ShippingRequestQueue")               |
    }                                                               |
)                                                                   |
public class ShippingRequestProcessorMDB                            |#2
        implements MessageListener {                                |
    private java.sql.Connection connection;
    private DataSource dataSource;

    @Resource
    private MessageDrivenContext context;                           |#4

    @Resource(name="jdbc/TurtleDS")                                 |#7
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }


    @PostConstruct                                                  |#6
    public void initialize() {
        try {
            connection = dataSource.getConnection();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

    @PreDestroy                                                     |#6
    public void cleanup() {
        try {
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

    public void onMessage(Message message) {                        |#3
        try {
            ObjectMessage objectMessage = (ObjectMessage)message;
            ShippingRequest shippingRequest =
                (ShippingRequest)objectMessage.getObject();
            processShippingRequest(shippingRequest);              #9
        } catch (JMSException jmse) {
            jmse.printStackTrace();
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
            context.setRollBackOnly();                                  | #5
        } catch (SQLException sqle) {
            sqle.printStackTrace();
            context.setRollBackOnly();                                  |#5
        }
    }

    private void processShippingRequest(ShippingRequest request)     |#8
            throws SQLException {
        Statement statement = connection.createStatement();
        statement.execute(
            "INSERT INTO "
                + "SHIPPING_REQUEST ("
                + "ITEM, "
                + "SHIPPING_ADDRESS, "
                + "SHIPPING_METHOD, "
                + "INSURANCE_AMOUNT ) "
                + "VALUES ( "
                + request.getItem() + ", "
                + "\'" + request.getShippingAddress() + "\', "
                + "\' " + request.getShippingMethod() + "\', "
                + request.getInsuranceAmount() + " )");
    }
}
```
(annotation) <#1 MessageDriven annotation>
(annotation) <#2 Message listener interface>
(annotation) <#3 Message listener implemenation>
(annotation) <#4 Message driven context>
(annotation) <#5 Using the MDB context>
(annotation) <#6 Life-cycle callbacks>
(annotation) <#7 Resource injection>
(annotation) <#8 Business logic>
(annotation) <#9 Persist object>

Taking a bird's eye view of listing 4.2, the `@MessageDriven` annotation identifies this object as an MDB and specifies the MDB configuration, including the fact that we are listening on the shipping request queue #1. #2 marks this MDB as a JMS message listener. The `onMessage` method provides the implementation for the message listener interface #3 and processes incoming messages. A message driven context is injected in #4 and used inside the `onMessage` method #5 to rollback transactions as needed. A database resource is injected in #7. The life-cycle callbacks #6 open and close a connection derived from the database resource. Finally, the shared JDBC connection is used by the business logic #8 called in `onMessage` #9 to save each shipping request into the database. We will discuss major MDB features by analyzing this code in greater detail presently - starting with the `@MessageDriven` annotation.

4.3.4 Using the @MessageDriven Annotation

MDBs are one of the simplest kinds of EJBs to develop and support the smallest number of annotations. In fact, the `@MessageDriven` annotation and the `@ActivationConfigProperty` annotation nested inside it are the only MDB-specific annotations to discuss. The `@MessageDriven` annotation we use for the example probably is very representative of what you will be using most of the time. The annotation is defined as follows:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface MessageDriven {
    String name() default "";
    Class messageListenerInterface default Object.class;
    ActivationConfigProperty[] activationConfig() default {};
   String mappedName();
    String description();

}
```

Notice that all three of the annotation's arguments are optional. If you are a minimalist, you can keep the annotation as simple as the code that follows, leaving any details to be added elsewhere, such as the deployment descriptor:

```
@MessageDriven
public class ShippingRequestProcessorMDB
```

The first element, `name`, specifies the name of the MDB. In our case, the name is specified to be 'ShippingRequestProcessor'. If omitted, the name is set as the name of the class -- `ShippingRequestProcessorMDB` in our example. The second parameter, `messageListenerInterface` specifies what message listener the MDB implements. The last parameter, `activationConfig` is used to specify listener-specific configuration properties. Both of these two last parameters deserve to be covered on their own, as we will do next.

4.3.5 Implementing the MessageListener

An MDB implements a message listener interface for the very same reason our plain JMS consumer implemented the `javax.jms.MessageListener` interface. The container uses the listener interface to register the MDB with the message provider and pass incoming messages by invoking implemented message listener methods. Using the `messageListenerInterface` parameter of the `MessageDriven` annotation is just one way to specify a message listener and instead we could have done the following:

```
@MessageDriven(
    name="ShippingRequestJMSProcessor",
    messageListenerInterface="javax.jms.MessageListener")
public class ShippingRequestProcessorMDB {
```

Instead, we chose to omit this parameter and specified the interface using the `implements` keyword:

```
public class ShippingRequestProcessorMDB implements MessageListener {
```

Yet another option is to specify the listener interface through the deployment descriptor and leave this detail out of code altogether. Which approach you choose is largely a matter of your likes and dislikes. We like the second approach because it is looks a lot like our JMS example.

The MDB feature of being able to specify a message listener with relative flexibility looks especially cool if you consider the following scenario: suppose that we switch messaging technologies

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

and decide to use JAXM (Java API for XML Messaging[5]) to send shipping requests instead of JMS. Thanks to JCA support, you can use still use MDBs to receive shipping requests (see the sidebar titled "JCA Connectors and Messaging" to see how this might be done). All we would have to do is switch to the JAXM message listener interface, `javax.jaxm.OneWayMessageListener` instead of `javax.jms.MessageListener` and reuse most of the MDB code and configuration:

```
public class ShippingRequestProcessorMDB implements
    javax.jaxm.OneWayMessageListener {
```

Whichever way you choose to specify the message listener, make sure you provide a valid implementation of all methods required by your message listener, especially when using the deployment descriptor approach, where there are no compile-time checks to watch your back. Now, let's take a look at the last (but definitely not least) parameter of the `MessageDriven` annotation: `activationConfig`.

4.3.6 Using ActivationConfigProperty

The `activationConfig` property of the message driven annotation lets you provide messaging-system specific configuration information through an array of `ActivationConfigProperty` instances. `ActivationConfigProperty` is defined as follows:

```
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

Each activation property is essentially a name-value pair that the underlying messaging provider understands and uses to setup the MDB. The best way of understanding how this works is through example. In this example, we provide three of the most common JMS activation configuration properties: `destinationType`, `connectionFactoryJndiName` and `destinationName`.

```
@MessageDriven(
    name="ShippingRequestProcessor",
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/QueueConnectionFactory"
        ),
        @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/ShippingRequestQueue")
    }
)
```

The `destinationType` property tells the container this JMS MDB is listening to a queue. If we were listening to a topic instead, the value could be specified to 'javax.jms.Topic'. The

[5] JAXM is essentially a SOAP-based XML messaging API. For more information, visit http://java.sun.com/webservices/jaxm/.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

connectionFactoryJndiName specifies the JNDI name of the connection factory that should be used to create JMS connections for the MDB. Lastly, the destinationName parameter specifies that we are listening for messages arriving at a destination with the JNDI name of 'jms/ShippingRequestQueue'. There are a few other configuration properties for JMS that we will describe in the sections that follow. Visualizing what happens behind the scenes can help you understand and remember these configuration properties. The container does something very similar to our JMS message consumer setup steps (as shown in listing 4.2) to bootstrap the MDB. Most of the method parameters that we specify during those steps are made available as configuration properties in the MDB world.

acknowledgeMode

Messages are not actually removed from the queue until the consumer acknowledges them. There are many 'modes' through which messages can be acknowledged. By default, the acknowledge mode for the underlying JMS session is assumed to be AUTO_ACKNOWLEDGE, which meant the session acknowledged messages on our behalf in the background. This is the case for our example as we omitted this property. If we wanted to we could change the acknowledge mode to DUPS_OK_ACKNOWLEDGE (or any other acknowledge mode we discussed in the JMS section) using the following:

```
@ActivationConfigProperty(
    propertyName="acknowledgeMode",
    propertyValue="DUPS_OK_ACKNOWLEDGE")
```

All of the acknowledgment modes supported by JMS are listed in table 4.1:

1.6 **Table 4.1: JMS session acknowledge modes. For non-transacted sessions, you should choose the mode most appropriate for your project. In general, AUTO_ACKNOWLEDGE is the most common and convenient.**

Acknowledgement Mode	**Description**
AUTO_ACKNOWLEDGE	The session automatically acknowledges receipt after a message has been received or is successfully processed.
CLIENT_ACKNOWLEDGE	We have to manually acknowledge the receipt of the message by calling the *acknowledge()* method on the message.
DUPS_OK_ACKNOWLEDGE	The session can lazily acknowledge receipt of the message. This is similar to AUTO_ACKNOWLEDGE but useful when the application can handle delivery of duplicate messages and rigorous acknowledgement is not a requirement.
SESSION_TRANSACTED	This is returned for transacted sessions if the *Session.getAcknowledgeMode()* method is invoked.

subscriptionDurability

If our MDB is listening on a topic, we can specify if the topic subscription is durable or non-durable.

Recall that in the pub-sub domain, a message is distributed to all currently subscribed consumers. In general, this is very much like a broadcast message in that anyone who is not connected to the topic at the time does not receive a copy of the message. The exception to this rule is what is called a *durable subscription*. A durable subscription means that once such a subscription is obtained on a topic, all messages sent to the topic are guaranteed for delivery to a consumer holding a subscription.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

If the *durable subscriber* is not connected to a topic when a message is received, the MOM retains a copy of the message until the subscriber connects and delivers the message. A durable subscriber is created as follows:

```
MessageConsumer playBoySubscriber = session.createDurableSubscriber(
    playBoyTopic, "JoeOgler");
```

In the preceding code, we are creating a durable subscription message consumer to the `javax.jms.Topic playBoyTopic` with a subscription ID of 'JoeOgler'. From now on, all messages to the topic will be held until a consumer with the subscription ID 'JoeOgler' receives them. You can remove this subscription with the following code when needed:

```
session.unsubscribe("JoeOgler");
```

If you want the MDB to be a durable subscriber then the action config property would look like the following:

```
@ActivationConfigProperty(
    propertyName="destinationType",
    propertyValue="javax.jms.Topic"),
@ActivationConfigProperty(
    propertyName="subscriptionDurability",
    propertyValue="Durable")
```

For non-durable subscriptions, we explicitly set the value of the `subscriptionDurability` property to `'NonDurable'`, which is also the default.

messageSelector

The `messageSelector` property is the MDB parallel to applying a selector for a JMS consumer. In our code consumed all messages at the destination. If we needed to, we could filter what messages we retrieve by using a *message selector*. A message selector is essentially a criteria applied to the headers and properties of messages specifying which messages the consumer wants to receive. For example, if we wanted to receive all shipping requests that had a 'Fragile' property set to true, we would use the following code:

```
MessageConsumer consumer = session.createConsumer(destination,
    "Fragile IS TRUE");
```

As you might have noticed, the selector syntax is almost identical to the where clause in SQL-92, but the selector syntax usess message header and property names instead of column names. Selector expressions can be as complex and expressive as you need them to be and can include literals, identifiers, white spaces, expressions, standard brackets, logical and comparison operators, arithmetic operators, null comparisons, etc.

Using our JMS message selector example from above we could specify in our MDB that we want to handle only fragile shipping requests as follows:

```
@ActivationConfigProperty(
    propertyName="messageSelector",
    propertyValue="Fragile IS TRUE")
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Table 4.2 summarizes some common message selector token.

1.7 **Table 4.2: Commonly used message selector token. The syntaxs of message selectors are quite similar to SQL WHERE clause**

Type	**Description**	**Example**
Literals	This can either be strings, exact or approximate numeric values or booleans	'BidManagerMDB' 100 TRUE
Identifiers	Identifiers can either be a message property or header name and are case sensitive	RECEPIENT NumOfBids Fragile JMSTimestamp
Whitespace	Same as defined in the Java language specification: space, tab, form feed and line terminator	
Comparison Operator	Comparison operators such as =, >, >=, <=, <>	RECIPIENT='BidManagerMDB' NumOfBids>=100
Logical operators	All three types of logical operators NOT, AND, OR are supported	RECIPIENT='BidManagerMDB' AND NumOfBids>=100
Null comparison	IS NULL and IS NOT NULL comparisons	FirstName IS NOT NULL
True/false comparison	IS [NOT] TRUE and IS [NOT] FALSE comparisons	Fragile IS TRUE Fragile IS FALSE

Having discussed the last of the MessageDriven annotation parameters, we are now ready to examine life-cycle callbacks in MDB and their typical uses.

4.3.8 Using Bean Life-Cycle Callbacks

As you might remember from Chapter 3, similar to stateless session beans, MDBs have a very simple life cycle. The container:

- Creates MDB instances and sets them up.
- Injects resources, including the message driven context discussed in the next chapter in detail.
- Places instances in a managed pool.
- Pulls an idle bean out of the pool when a message arrives (the container may have to increase the pool size at this point).
- Executes the onMessage method.
- When the onMessage method finishes executing, pushes the idle bean back into the "method-ready" pool.
- As needed, retires (a.k.a. destroys) beans out of the pool.
- Figure 4.7 depicts the MDB lifecycle.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

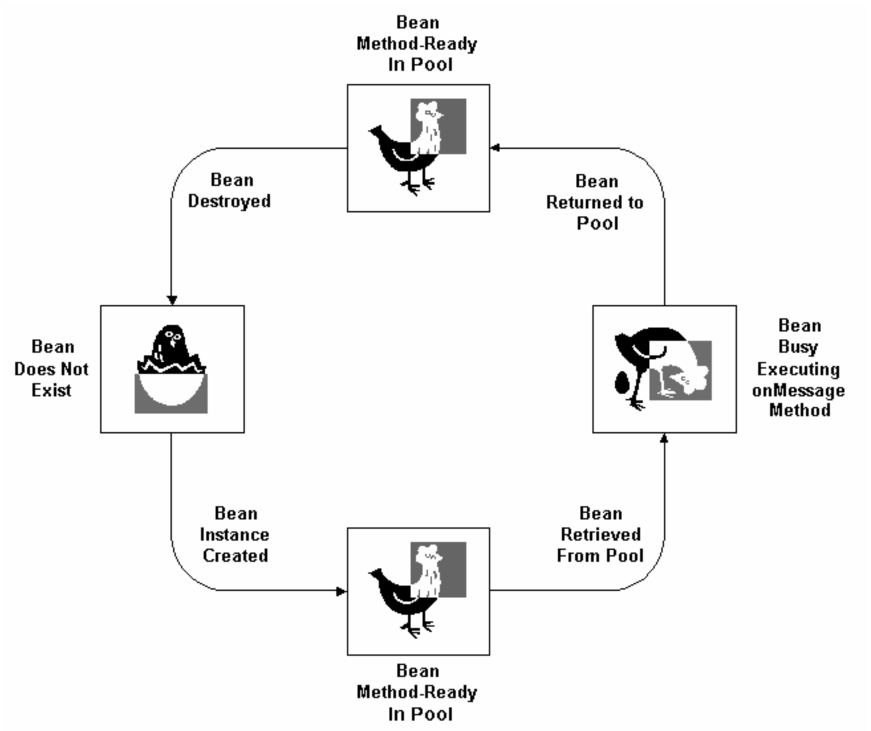


Figure 4.7: The chicken or the egg - the MDB life cycle is has three states: does not exist, idle and busy. As a result, there are only two life-cycle callbacks corresponding to bean creation and destruction.

The MDB's two lifecycle callbacks are `PostConstruct`, which is called immediately after an MDB is created, set up and all the resources are injected, and `PreDestroy`, which is called right before the bean instance is retired and removed from the pool. The typical usage of these callbacks in MDB are for allocating and releasing injected resources that are used by the `onMessage` method, which is exactly what we do in our example.

The `processShippingRequest` method saves off shipping requests that the `onMessage` method extracts from the incoming JMS message:

```
private void processShippingRequest(ShippingRequest request)
        throws SQLException {
    Statement statement = connection.createStatement();
    statement.execute(
        "INSERT INTO "
            + "SHIPPING_REQUEST ("
          …
            + request.getInsuranceAmount() + " )");
}
```

The method creates a statement from an open JDBC connection and uses it to save a record into the `SHIPPING_REQUEST` table containing all the fields from the `ShippingRequest` object. The JDBC connection object used to create the statement is a classic heavy-duty resource. It is expensive to open and should be shared whenever possible. On the other hand, it can hold a number of native resources, so it is very important to close the connection when it is no longer needed. We accomplish both these goals using callback methods as well as resource injection.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Firstly, the JDBC data source that the connection is created from is injected using the `@Resource` annotation:

```
@Resource(name="jdbc/TurtleDS")
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
```

The `@Resource` annotation tells the EJB container that it should lookup a `java.sql.DataSource` named `jdbc/TurtleDS` from JNDI and pass it to the `setDataSource` method after creating a new instance of the bean. The `setDataSource` method, in turn, saves the data source in an instance variable. After injecting resources, the container checks if there are any designated `PostConstruct` methods that need to be invoked before the MDB is put into the pool. In our case, we mark the `initialize` method with the `@PostConstruct` annotation:

```
@PostConstruct
public void initialize() {
   ...
   connection = dataSource.getConnection();
   ...
}
```

In the `initialize` method, we are creating a java.sql.Connection from the injected data source and saving it into the `connection` instance variable used in `processShippingRequest`. At some point, the container decides that our bean should be removed from the pool and destroyed (perhaps at server shutdown). The `PreDestroy` callback given us a chance to cleanly teardown bean resources before this is done. In the `cleanup` method marked with the `@PreDestroy` annotation, we tear down the database connection resource before the container retires our bean:

```
@PreDestroy
public void cleanup() {
    ...
    connection.close();
    connection = null;
    ...
}
```

Although database resources and their management are the predominant uses of resource injection and life-cycle methods in MDBs, another important resource we have explicitly seen being used in the JMS sections are also important for MDB. These are the JMS destination and connection factory administrative objects as well as the JMS connections. We will explore how these are utilized in MDBs next.

4.3.9 Sending JMS Messages from MDB

Somewhat ironically, a task you will find yourself doing time and again in an MDB is sending JMS messages. As a simple example, we might have to communicate back to ActionBazaar from the ShippingRequestProcessorMDB if a shipping request is incomplete. It is only natural that this notification be done via JMS messages sent to an 'error queue' that ActionBazaar listens to. Fortunately, we have already seen how to send a JMS message in listing 4.1. This task is even simpler

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

and more robust in MDB. The queue named 'jms/ShippingErrorQueue' and the connection factory named 'jms/QueueConnectionFactory' could be injected using the `@Resource` annotation as follows:

```
    @Resource(name="jms/ShippingErrorQueue")
    private javax.jms.Destination errorQueue;
    @Resource(name="jms/QueueConnectionFactory")
    private javax.jms.ConnectionFactory connectionFactory;
```

A shared `javax.jms.Connection` instance could then be created and destroyed using life-cycle callbacks just as the JDBC connection was managed in the previous section:

```
    @PostConstruct
    public void initialize() {
        ...
        jmsConnection = connectionFactory.createConnection();
        ...
      }
    @PreDestroy
    public void cleanup() {
        ...
        jmsConnection.close();
        ...
      }
```

Finally, the business method to send the error message would look very much like the rest of the JMS session code in listing 4.1:

```
    private void sendErrorMessage(ShippingError error) {
        Session session = jmsConnection.createSession(true,
            Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(errorQueue);
        ...
        producer.send(message);
        session.close();
    }
```

Although we did not explicitly show it in our example, there is one more MDB feature you should know about – MDB transaction management. We will discuss EJB transactions in general in much more detail in the next chapter, so we will simply give you the "bargain basement" version for dealing with the specifics of MDBs in the next section.

4.3.10 Managing MDB Transactions

In our plain JMS examples, we specified whether the JMS session would be transactional when we created it. On the other hand, if you look closely at the MDB example, it does not specify anything about transactions at all. Instead, we are letting the container use the default transactional behavior for MDBs. By default, the container will start a transaction before the `onMessage` method is invoked and will commit the transaction when the method returns unless the transaction was marked rolled back through the message driven context. We will learn more about transactions in Chapter 6.

This brief discussion of transaction management finishes off our analysis of the basic features that MDBs offer. We have discussed how we can use MDBs to leverage the power of messaging without dealing with the low-level details of the messaging API. In addition, MDBs give us a whole host of EJB features for free, such as multithreading, resource injection, life cycle management and container-managed transactions. We have tried to formulate our code samples such that you can use

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

them as templates for solving real business problems and carrying things to much deeper waters. Before concluding this chapter, we will give you some tips for dealing with the nuances of MDBs that might help you navigate those deeper waters safely and with confidence.

4.4 MDB Tips and Tricks

Like all technologies, MDBs have some pitfalls to look out for and some best practices to keep in mind. This is particularly true in the demanding environments where messaging is typically deployed. If you are reading this chapter, this probably means you. We will now share the most common such concepts you should keep an eye on.

Choose your messaging models carefully. Before you wade knee deep in code, consider your choice of messaging model carefully. You might find that PTP will solve your problem nine times out of ten. In some cases, though, pub-sub really is better, especially if you find yourself broadcasting the same message to more than one receiver (such as our system outage notification example). Luckily, most messaging code is domain independent and you should strive to keep it that way. For the most part, switching domains should be just a matter of configuration.

Remember modularization. Because MDBs are so similar to Session Beans, it is very natural to start putting business logic right into message listener methods. Business logic should be decoupled and modularized away from messaging-specific concerns. We tried to follow this principle by coding the `processShippingRequest` method and invoking it from `onMessage`. An excellent practice (but one that would have made this chapter unnecessarily complicated) is to put business logic in Session Beans and invoke them from the `onMessage` method.

Make Good Use of Message Filters. There are some valid reasons for using a single messaging destination for multiple purposes. Message selectors come in very handy in these circumstances. For example, if you are using the same queue for both shipping requests and order cancellation notices, you can have the client set a message property identifying the type of request. You can then use message selectors on two separate MDBs to isolate and handle each kind of request.

Conversely, in some cases, you might dramatically improve performance and keep your code simple by using separate destinations instead of using selectors. In the example just mentioned, using separate queues and MDBs for shipping requests and cancellation orders could make message delivery much faster. In this case, the client would have to send each request type to the appropriate queue.

Choose Message Types Carefully. The choice of message type is not always as obvious as it seems. For example, it is a very compelling idea to use XML strings for messaging. Among other things, this tends to promote loose coupling between systems. In our example, the Turtle server would know about the format of the XML message and not the `ShippingRequest` object itself.

The problem is that XML tends to bloat the size of the message, significantly degrading MOM performance. In certain circumstances, it might even be the right choice to use binary streams in the message payload, which puts the least amount of demand on MOM processing as well as memory consumption.

Be Wary of Poison Messages. Imagine that a message is handed to you that your MDB was not able to consume. Using our example, let us assume that we receive a message that is not an `ObjectMessage`. As you can see from code snippet below, if this happens, the cast in `onMessage` will throw a `java.lang.ClassCastException` #1:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
try {
    ObjectMessage objectMessage = (ObjectMessage)message;           |#1
    ShippingRequest shippingRequest =
            (ShippingRequest)objectMessage.getObject();
        processShippingRequest(shippingRequest);
} catch (JMSException jmse) {
    jmse.printStackTrace();
    context.setRollBackOnly();
}
```
(annotation) <#1 Wrong message type will fail cast>

Since `onMessage` will not complete normally, the container will be forced to roll back the transaction and put the message back on the queue instead of acknowledging it (in fact, since a runtime exception is thrown, the bean instance will be removed from the pool). The problem is, since we are still listening on the queue, the same message will be delivered to us again and we will be stuck in the accept/die loop indefinitely! Messages that cause this all-too-common scenario are called "poison messages".

The good news is that there are several mechanisms that many MOMs and EJB containers provide to deal with poison messages, including "redelivery" counts and "dead message" queues. If you we set up the redelivery count and dead message queue for the shipping request destination, the message delivery will be attempted for the specified number of times. After the redelivery count is exceeded, the message will be moved to a specially designated queue for poison messages called the "dead message" queue. The bad news is that these mechanisms are not standardized and are vendor-specific.

Configure MDB Pool Size. Most EJB containers will let you specify the maximum number of instances of a particular MDB the container can create. In effect, this controls the level of concurrency. If there are five concurrent messages to process and the pool size is set to three, the container will wait until the first three messages are processed before assigning any more instances. This is a double-edged sword and requires careful handling. If we set our MDB pool size to be too small, messages will be processed slowly. At the same time, it is desirable to place *reasonable* limits on the MDB pool size so that many concurrent MDB instances do not choke the machine. Unfortunately, at the time of this writing, setting MDB pool sizes is not standardized and is provider-specific.

4.4 Summary

In this chapter, we have covered basic messaging concepts, the Java Messaging Service, as well as Message Driven beans. Messaging is an extremely powerful technology for the enterprise and it helps build loosely coupled systems. Java Messaging Service allows you to use message-oriented middleware from enterprise Java applications. Use JMS API to build a message consumer application can be pretty time consuming and involved task and MDBs make using MOM in a standardized manner through Java EE extremely easy.

Note, however, that messaging and MDBs are not right for all circumstances and can be overused. One such case at hand is using the Request/Reply domain (discussed in the sidebar titled "The Request-Reply Model"). This model entails a lot of extra complexity as compared to simple PTP or pub-sub messaging. If you find yourself using this model extensively and in ways very close to

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

synchronous messaging, it might be worth thinking about switching to a synchronous technology such as RMI, SOAP or remote Session Bean calls.

Few major EJB features we skated over in this chapter are dependency injection, interceptors, timers, transaction and security. EJB 3.0 largely relieves us from these system level concerns while providing extremely robust and flexible functionality. We will discuss dependency injection, timers and interceptors in next chapter.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 5 Learning Advanced EJB Concepts

In the last two Chapters we focused on developing Session Beans and Message Driven Beans. Although we discussed a few bean type specific features in detail, we generally avoided covering topics not closely related to introducing the basics. In this Chapter we will build on the material in the previous Chapters and introduce a few advanced concepts applicable to Message Driven Beans and Session Beans. It is very likely that you will find these EJB 3.0 features extremely helpful while using EJB in the real world.

In this Chapter we start by discussing the how containers provide the services behind the scene and how to access environment information. Armed with that knowledge, we then move on to advanced usage of dependency injection, JNDI lookups, EJB interceptors, and the EJB timer service. EJB 3.0 largely relieves us from these system level concerns while providing extremely robust and flexible functionality.

As foundation to the rest of the Chapter, we will very briefly examine these EJB internals first.

5.1 EJB Internals

Although we've talked about the role of the container and the concept of managed services, we haven't talked about how most containers go about providing managed services. The secret to understanding and remembering these and the other EJB services is knowing how the container provides them. Without going into too much details we will discuss about EJB objects that does the magic of providing the service and then about EJB context that a bean can use to access runtime environment and use container services.

5.1.1 EJB Behind the Scenes

EJB centers on the idea of managed objects. As we saw in the previous Chapters, EJB 3.0 beans are just annotated POJOs themselves. When a client invokes an EJB method using the bean interface it actually do not work directly on the bean instance. The container makes beans "special" by acting as a *proxy* between the client and the actual Bean instance. This enables the container to provide EJB services to the client on behalf of the bean instance.

EJB Object

For each bean instance, the container automatically generates a proxy named an *EJB Object*. The EJB Object has access to all the functionality of the container including the JNDI registry, security, transaction-management, thread-pools, session-management and pretty much anything else that is necessary to provide EJB services. The EJB Object is aware of the bean configuration and what services the POJO is supposed to provide.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Since all requests to the EJB instance is passed through the EJB Object proxy, the EJB Object can "insert" container services to client requests as needed, including managing all aspects of the bean life-cycle. Figure 5.1 is a typical visual representation of this technique.

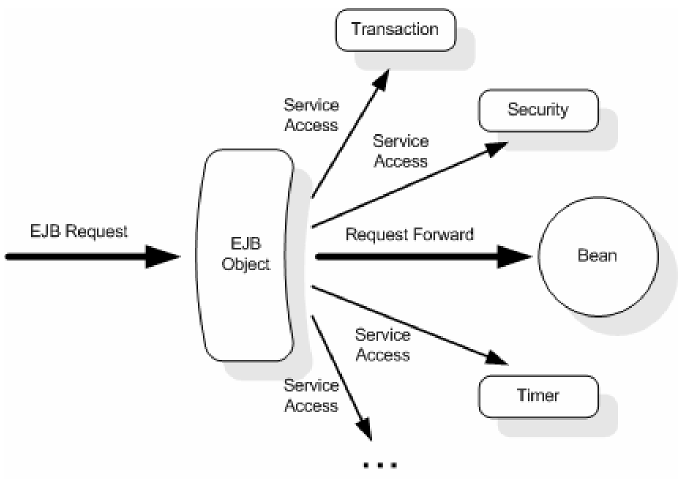


Figure 3.1: The "magic" of EJB. The container-generated EJB Object receives all EJB client requests as the proxy, reads configuration and inserts container services as required before forwarding client requests to the bean instance.

As we've seen in the previous Chapters, the beauty of technique is that all the service details are completely transparent to bean clients and even to bean developers. In fact, a container implementation is free to implement the services in the most effective way possible as well as providing vendor-specific feature and performance enhancements. This is really fundamentally all there is to the "magic" parts of EJB. For Session Beans, the client interacts with the EJB Object through the business interface. On the other hand, for Message Driven Beans, the EJB Object or Message Endpoint sits between the message provider and the bean instance.
Let's now take a look at how EJBs can access the container environment in which the EJB Object itself resides.

5.1.2 EJB context: accessing the Runtime Environment

EJB components are generally meant to be agnostic of the container. This means that in the ideal case, EJB components should merely hold business logic and never access the container or use container services directly. As we saw in the previous Chapters and will see further in the coming Chapters, services like transaction management, security, dependency injection and so forth are meant to be "overlaid" on the bean through configuration.

However, in the real world, it is sometimes necessary for the bean to explicitly use container services in code. These are the situations the EJB context is designed to handle. The `javax.ejb.EJBContext` interface is essentially your backdoor into the mystic world of the container. We will see the definition of EJBContext, its use and using dependency injection to retrieve EJBContext.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Defining the EJBContext Interface

As you might be able to see from the interface definition in Listing 5.1, the interface allows direct programmatic access to some of the services such as transaction, security, etc. typically specified through configuration and completely managed by the container.

Listing 5.1: javax.ejb.EJBContext interface

```
public interface EJBContext {
    public Principal getCallerPrincipal();                             |#1
    public boolean isCallerInRole(String roleName);                    |#1
    public EJBHome getEJBHome();
    public EJBLocalHome getEJBLocalHome();
    public boolean getRollbackOnly();                                  |#2
    public UserTransaction getUserTransaction();                       |#2
    public void setRollbackOnly();                                     |#2
    public TimerService getTimerService();                             |#3

    public Object lookup(String name);                                 |#4
}
```

(annotation) <#1 Bean-managed security>
((annotation) <#2 Bean transaction management>
(annotation) <#3 Access to timer service>
(annotation) <#4 JNDI lookup>

We will briefly mention what each of these methods do now. A detailed analysis for most of them will be left for when we discuss the services that each of the methods is related to. For now, you should note the array of services offered through the EJB context as well the method patterns, shown in table 5.1.

1.8 Table 5.1 You can use javax.ejb.EJBContext to access runtime services.

Methods	Description
`getCallerPrincipal` `isCallerInRole`	These methods are used in bean-managed security. We will discuss these two methods further in Chapter 6 when we talk about programmatic security.
`getEJBHome` `getEJBLocalHome`	These methods are used to obtain the bean's "remote home" and "local home" interfaces respectively. Both are optional for EJB 3.0 and are hardly used beyond legacy EJB 2.1 beans. We will not discuss these methods beyond this basic introduction. They are mainly provided for backwards compatibility.
`getRollbackOnly,` `setRollbackOnly` `getUserTransaction`	These methods are used for EJB transaction management in the case of *container-managed transactions*. We will discuss container-managed transactions in greater detail in Chapter 6.
`getUserTransaction`	This method is used for EJB transaction management in the case of *bean-managed transactions*. We will discusse bean-managed transactions in greater detail in Chapter 6.
`getTimerService`	This method is used to get access to the EJB timer service. We will discuss EJB timers later on in this Chapter.
`lookup`	This method is used to get references to Objects stored in the JNDI registry. With the introduction of DI in EJB 3.0, direct JNDI lookup has largely been made unnecessary. However, there are some edge cases that DI cannot handle or DI is simply not available. This method comes in very handy in these circumstances. Discussed later in this section.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Both Session and Message Driven Beans have their own subclasses of the `javax.ejb.EJBContext` interface. As shown in Figure 5.2, the Session Bean specific subclass is `javax.ejb.SessionContext` while the MDB specific subclass is `javax.ejb.MessageDrivenContext`.

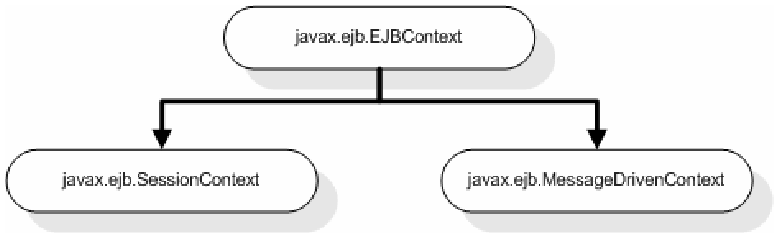


Figure 5.2: The EJB Context interface has a subclass for both the Session and Message Driven Bean types.

Each subclass is designed to suit the particular runtime environment of each bean type. As a result, they either add methods to the superclass or invalidate methods not suited for the bean type.

Using EJBContext

As we discussed earlier you can get access to several container services such as transaction or security by using EJBContext. Interestingly you can get access to the EJBContext through DI. For example, a `SessionContext` could be injected into a bean as follows:

```
@Stateless
public class PlaceBidBean implements PlaceBid {
   @Resource
   SessionContext context;
   ...
}
```

In the code snippet, the container detects the `@Resource` annotation on the `context` variable and figures out that the bean wants an instance of it's session context. The `SessionContext` adds a number of methods specific to the Session Bean environment, including `getBusinessObject`, `getEJBLocalObject`, `getEJBObject`, `getInvokedBusinessInterface` and `getMessageContext`. All of these are fairly advanced methods that are rarely used. Note that `getEJBLocalObject` and `getEJBObject` methods are meant for EJB 2.x beans and will generate exceptions if used with EJB 3.0 beans. We will not discuss these methods further and will leave them for you to explore on your own.

The `MessageDrivenContext` adds no methods specific to MDB. Rather, it throws exceptions if the `getCallerPrincipal`, `isCallerInRole, getEJBHome or getEJBLocalHome methods are called since they make no sense in a messaging based environment (recall that a message driven bean has no business`

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

`interface and is never invoked directly by the client). Much like a session context, a MessageDrivenContext` can be injected as follows:

```
@MessageDriven
public class OrderBillingMDB {
   @Resource MessageDrivenContext context;
   ...
}
```

Note it is illegal to inject a `MessageDrivenContext` into a Session Bean or a `SessionContext` into a Message Driven Bean.This is about as much time as we need to spend on the EJB context right now. Rest assured that we will see more of its use in Chapter 6.

In the meanwhile, we'll turn our attention back to a vital part of EJB 3.0—dependency injection. We provided a brief overview of dependency injection in Chapter 2 and have been seeing EJB DI in action in last few Chapters. We just saw a pretty intriguing use-case in injecting EJB contexts. In reality, EJB DI is a like a Swiss army knife, it is an all-in-one tool that can be used in very unexpected ways. Let's take a look at some of these advanced usages next.

5.2 Accessing Resources using DI and JNDI

We've seen EJB 3.0 DI in its primary incarnations already—the `@javax.ejb.EJB` and `@javax.annotation.Resource` annotations. EJB 3.0 DI comes in three more forms—the `@javax.persistence.PersistenceContext` and `@javax.persistence.PersistenceUnit` annotations. We'll see these two annotations in action in Part 3 of this book.

We've also seen only a small subset of the power of the `@Resource` annotation. So far, we've used the `@Resource` annotation to inject JDBC data sources, JMS connection factories and JMS destinations. Unlike some lightweight containers such as Spring, EJB 3.0 does not allow injection of POJOs that aren't beans. However, the `@Resource` annotation allows for a variety of other uses, some of which we will cover in the coming section. In this section we will learn how to use `@Resource` annotations, its parameters, difference between setter and field injection and see `@Resource` annotation in action to inject a variety of resources such as mail, environment entries, timer service, etc. Finally we will see learn lookuping of resources using JNDI and lookup method in EJBContext.

5.2.1 Resource Injection using @Resource

The `@Resource` annotation is by far the most versatile mechanism for DI in EJB 3.0. As we noted, in most cases the annotation is used to inject JDBC data sources, JMS resources and EJB contexts. However, the annotation can also be used for email server resources, environment entries or even EJB references. We will briefly take a look at each of these cases. For convenience, we will use the familiar JDBC data source example to explain to the basic features of the `@Resource` annotation before moving on to the more involved cases. As review, here is how the code to inject a data source into the `PlaceBid` Bean from Chapter 2 looks like:

```
@Stateless
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
public class PlaceBidBean implements PlaceBid {
    ...
    @Resource(name="jdbc/actionBazaarDB")
    private javax.sql.DataSource dataSource;
```

In this particular case, the container would not have to work very hard to figure out what resource to inject because the `name` parameter is explicitly specified. As we know, this parameter specifies the JNDI name of the resource to be injected, which in out case is specified to be 'jdbc/actionBazaarDB'. Although we didn't mention this little detail before, the value specified by the `name` parameter is actually interpreted further by the container to match a value specified in the `res-ref-name` in the `resource-ref` tag in deployment descriptor as in the following example.

```
<resource-ref>
     <res-ref-name>jdbc/actionBazaarDB</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
</resource-ref>
```

The value of the `name` parameter in `@Resource` (or `res-ref-name`) is actually translated to a fully qualified JNDI mapping in the form: java:comp/env/[value of the name parameter] (see the sidebar item on ENC). In our example, the actual complete JNDI path for the resource will be: java:comp/env/jdbc/actionBazaarDB. If you don't specify the `name` element in the `@Resource` annotation the JNDI name for the resource will be of the format java:comp/env/ [bean Class name including package]/[value of the annotated field/property]. If we didn't specify the `name` element in the `@Resource` annotation, the container would be looking for the JNDI path: java:comp/env/actionbazaar.buslogic.PlaceBidBean/dataSource.

The Environment Naming Context and Resolving Global JNDI Names

If you are familiar with how JNDI references worked in EJB 2.x, you will remember about the Environment Naming Context (ENC). ENC allows portability of the application without having to depend upon on global JNDI names. Global JNDI names for resources differ between application server implementations and ENC allows us to use a JNDI location that starts with java:comp/env/ instead of hard coding actual global path names. EJB 3.0 essentially assumes that all JNDI names used in code are local references and automatically prepends names with the 'java:comp/env/' prefix.

This automatic interpretation of EJB 3.0 JNDI names into local references is a very nice alternative to mentioning relative the local ENC prefix over and over again. However, this nice convenience does come at a price. Since you cannot use global names with the name parameter, you have to make sure to perform the mapping between the ENC and global JNDI names in all cases. Fortunately, many application servers will automatically resolve the ENC name to global JNDI name if a resource with same global JNDI name exists. For example, if you are using the Sun Glassfish or Oracle Application Server and you define a data source as below, the application server will automatically map the data source to the global JNDI resource bound to jdbc/ActionBazaarDS even if you didn't explicitly map the resource.

```
@Resource(name="jdbc/ActionBazaarDS")
private javax.jdbc.DataSource myDB;
```

Moreover, application servers will allow you to explicitly specify a global JNDI name using the mappedBy parameter of the @Resource annotation. For example, if you are using the JBoss Application Server and you have a data source with a global JNDI name of java:/DefaultDS you can specify the resource mapping as follows:

```
@Resource(name="jdbc/ActionBazaarDS", mappedName="java:/DefaultDS")
private javax.jdbc.DataSource myDB;
```

In this case, the data source with the global JNDI name of java:/DefaultDS will be looked up when the ENC java:comp/env/jdbc/ActionBazaarDS is resolved.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

However remember that using the mappedName parameter makes code less portable and we recommend you use deployment descriptors for mapping global JNDI names instead.

Note, just as the @Resource annotation, the @EJB annotation has a mappedBy parameter as well.

Behind the scenes, the container resolves the JNDI references to the resources and binds the resource to the ENC (see the sidebar item) during deployment. If the resource is not found during injection, the container throws a runtime exception and the bean becomes unusable.

Beyond JNDI name mapping, the `@Resource` annotation is meant to be a lot more flexible when it needs to be than what is apparent in our deliberately straightforward data source injection example. To understand some of these robust features, let us take a look at the definition for the annotation:

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Resource {
  String name() default "";
  String type() default Object.class;
  AuthenticationType authenticationType() CONTAINER;
  boolean shareable() default false;
  String description default "";
  String mappedName() default "";
}
```

The first thing you should note from the definition of the `@Resource` annotation is that it is not limited to being applied to instance variables. As the `@Target` value indicates, it can be applied to *setter methods*, method parameters and even to classes.

Setter vs. Field Injection

Other than field injection, setter injection is the most commonly used option for injection. To see how it works, let us translate our data source example to use setter injection:

```
@Stateless
public class PlaceBidBean implements PlaceBid {
    ...
    private DataSource dataSource;
    ...
    @Resource(name=”jdbc/actionBazaarDB”)
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

As you can see, setter injection relies on JavaBeans property naming conventions. In case you are unfamiliar with them, the conventions dictate that the instance variables of an object should always be private so that they cannot be externally accessible. Instead, an instance variable named *XX* should have corresponding non-private methods named *getXX* and *setXX* that allow it to be accessed and set externally. We’ve seen how the *setter* for the `PlaceBidBean dataSource` variable looks like. The *getter* could look like the following:

```
public DataSource getDataSource() {
    return dataSource;
}
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Just as in instance variable injection, the container inspects the `@Resource` annotation on the `setDataSource` method before a bean instance becomes usable, looks up the data source from JNDI using the `name` parameter value and calls the `setDataSource` method using the retrieved data source as parameter.

> Whether or not to use setter injection is largely a matter of taste. Although setter injection might seem like a little more work, they provide a few distinct advantages. Firstly it is easier to unit test since the public setter method by invoking it from a testing framework like JUnit. Secondly, it is easier to put initialization code in the setter if you need it.

In our case, we can open a database connection in the `setDataSource` method as soon as injection happens:

```
private DataSource dataSource;
private Connection connection;
...
@Resource(name="jdbc/actionBazaarDB")
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.connection = dataSource.getConnection();
}
```

The optional `type` parameter of the `@Resource` annotation can be used to explicitly set the type of the injected resource. For example, we could have chosen to tell the container that the injected resource is of type `javax.sql.DataSource` as following:

```
@Resource(name="jdbc/actionBazaarDB",
          type=javax.sql.DataSource.class)
private DataSource dataSource;
```

If omitted, the type of the injected resource is assumed to be the same as the type of the instance variable or property.

The `type` element is mandatory when `@Resource` annotation is used at the class level and use JNDI to obtain reference to the resource. We will deviate from dependency injection and we will see this usage next.

Using @Resource at the class level

You may recall from our earlier discussion that dependency injection is supported only in the managed classes and you cannot injection in helper or utility classes. In most applications use of helper classes is a reality and you have to use JNDI to lookup a resource. If you are not familiar with JNDI we recommend that you rerference to Appendix A for a brief discussion on JNDI. For looking up a resource from the helper class you have to reference the resource in the EJB class as follows:

```
@Resource(name="jdbc/actionBazaarDB",mappedName="jdbc/actionBazaarDS",
          type=javax.sql.DataSource.class)
@Stateless
public class PlaceBidBean implements PlaceBid
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

You can lookup the resource either from the EJB or the helper class as follows:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("java:comp/env/jdbc/ActionBazaarDS")
```

Before we conclude this section let us look at some remaining parameters of @Resource annotation. The other parameters for the `@Resource` annotation, `authenticationType`, `shareable`, `description` and `mappedName` are not used very often and we will not cover them in great detail. Table 5.1 describes the use of these parameters.

Table 5.1: @Resource annotation can be used to inject resources. The parameters in are the least used parameters of the annotation that you should note in case you need them.

Parameter	**Type**	**Description**	**Default**
AuthenticationType	Enum AuthenticationType {CONTAINER, APPLICATION}	The type of authentication required for accessing the resource. The CONTAINER value means that the container's security context is used for the resource. On the other hands, the APPLICATION value means that authentication for the resource must be provided by the application. We will talk more about EJB security in Chapter 6.	CONTAI NER
Shareable	Boolean	Specifies if the resource can be shared.	false
Description	String	The description of the resource.	""
MappedName	String	A vendor specific name that the resource may be mapped to, as opposed to the JNDI name. See the side bar tilted "Environment Naming Context and resolving global JNDI names in EJB 3.0" for details on this parameter.	""

Using injection for JDBC data sources is just the tip of the iceberg. We will start taking a look at the other uses of EJB DI next. In general, we are avoiding talking about how the resources are actually defined in the deployment descriptor for now. We will discuss this in much greater detail when we talk about application packaging and deployment descriptor tags in Chapter 11.

5.2.2 @Resource annotation In Action

In the previous sections we discussed different parameters of @Resource annotation and learnt how to use field or setter injection with @Resource to inject JDBC datasources. Next we will see @Resource annotation in action to inject resources such as JMS objects, mail resources, EJBContext, environment entries and timer Service.

Injecting JMS resources

Recall the discussion on messaging and Message Driven Beans in Chapter 4. If your application has anything to do with messaging, it is going to need to use JMS resources such as a `javax.jms.Queue`, `javax.jms.Topic`, `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`. Just like JDBC data sources, these resources are stored in the application server's JNDI context and can be injected through the `@Resource` annotation. As

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

an example, the following code injects a `Queue` bound to the name 'jms/actionBazaarQueue' to the `queue` field:

```
@Resource(name="jms/actionBazaarQueue")
private Queue queue;
```

EJBContext

Recall the discussion on the uses of the `EJBContext`, `SessionContext` and `MessageDrivenContext` interfaces in Section 5.2. One of the most common uses of injection is to get access to EJB contexts. The following code, used in the `PlaceBid` Session Bean, injects the EJB type specific context into the `context` instance variable:

```
@Resource SessionContext context;
```

A critical nuance to note is that the injected session context is not stored in JNDI. In fact, it would be incorrect to try to specify the `name` parameters in this case at all and servers will probably ignore the element if specified. Instead, when the container detects the `@Resource` annotation on the `context` variable, it figures out that the EJB context specific to the current bean instance must be injected by looking at the variable data type, `javax.ejb.SessionContext`. Since `PlaceBid` is a Session Bean, the result of the injection would be the same if the variable were specified to be the parent class, `EJBContext`. In the code below, an underlying instance of `javax.ejb.SessionContext` is still injected into the context variable, even if the variable data type is `javax.ejb.EJBContext`.

```
@Resource EJBContext context;
```

Using the above code in a Session Bean would make a lot of sense if you do not plan to use any of the bean type specific methods available through the `SessionContext` interface anyway.

Accessing environment entries

If you have been working with enterprise applications for a little while, it is very likely you have come across situations where some parameters of your application change from one deployment to another (such as customer site information, product version and so on). It is overkill to save this kind of "semi-static" information in the database. This is exactly the situation environment entry values are designed to solve.

For example, in the ActionBazaar application, we could have a need to set the censorship flag for certain countries. If this flag is on, the ActionBazaar application checks items posted against a censorship list specific to the country the application deployment instance is geared toward.

We can inject an instance of an environment entry as follows:

```
@Resource
private boolean censorship;
```

Environment entries are specified in the deployment descriptor and are accessible via JNDI. The ActionBazaar censorship flag could be specified as following:

```
<env-entry>
    <env-entry-name>country</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>China</env-entry-value>
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
</env-entry>
<env-entry>
    <env-entry-name>censorship</env-entry-name>
    <env-entry-type>java.lang.Boolean</env-entry-type>
    <env-entry-value>true</env-entry-value>
</env-entry>
```

Environment entries are essentially meant to be robust application constants and support a relatively small range of data types. Specifically, the values of the `env-entry-type` tag are limited to these Java types: `String`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Double`, and `Float`. Because environment entries are accessible via JNDI they can be injected by name. We could inject the censorship flag environment entry into any EJB by explicitly specifying the JNDI name as follows:

```
@Resource(name="censorship")
private boolean censorship;
```

As you might gather, the data types of the environment entry and the injected variable must be compatible. Otherwise, the container throws a runtime exception while attempting DI.

Accessing email resources

Other than JDBC data sources and JMS resources, the other heavy-duty resource enterprise applications often use is JavaMail API `javax.mail.Session`. JavaMail `Sessions` that abstract email server configuration can be stored in the application server JNDI registry. The `Session` can then be injected into an EJB using the `@Resource` annotation and used to send email. In the ActionBazaar application, this is useful for sending the winning bidder a notification after bidding on an item is over. The DI code to inject the mail `Session` could look like the following:

```
@Resource(name="mail/ActionBazaar")
private javax.mail.Session mailSession;
```

We will show you how to configure a mail session using the deployment descriptor in the Appendix.

Accessing the timer service

The container-managed timer service provides EJBs the ability to schedule tasks in a very simple way. We will learn more about timers very soon in Section 5.5. We inject the container timer service into an EJB using `@Resource` annotation using the following code:

```
@Resource
javax.ejb.TimerService timerService;
```

Just as in the case of the EJB context, the timer service is not saved in JNDI, but the container resolves the resource by looking at the data type of the injection target.

The `@Resource` annotation may be used for injecting EJB references accessible via JNDI into other EJBs. However, the `@EJB` annotation is provided specifically for this purpose and should be used in these circumstances instead. Refer to the discussion in Chapter 3 for details of this annotation.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

EJB 3.0 and POJO Injection

As you might have noted, the one DI feature glaringly missing is the ability to inject resources to POJOs and injection of POJOs that are not EJBs. You can still indirectly accomplish this by storing POJOs in the JNDI context (not a particularly easy thing to do) or using proprietary extension of your container vendor. We will hope that a future version of EJB 3.0 will provide expanded support for POJO injection similar to other lightweight DI capable frameworks like Spring.

You can also use POJO injection with Spring-enabled EJB 3.0 beans if you really need POJO injection in your EJB applications.

@Resource and annotation inheritance

In chapter 3, we learnt that an EJB bean class may inherit from another EJB class or a POJO. If the superclass defines any dependencies on resources using @Resource annotation those are inherited by the subclass. For example BidManagerbean extends another stateless EJB PlaceBidBean where PlaceBidBean defines a resource as in the following example:

```
@Stateless
public class PlaceBidBean implements PlaceBid{
@Resource(name="censorship")
   private boolean censorship;
..
}

@Stateless
public class BidManagerBean extends PlaceBidBean implements BidManager{
..
}
```

The environment entry defined in the PlaceBidBean will be inherited by the BidManagerBean and dependency injection will occur when an instance of BidManager is created.

As useful as DI is, it cannot always solve every problem. There are some cases where you must programmatically lookup resources from a JNDI registry yourself. We'll talk about some of these cases next as well as showing you how to do programmatic lookups.

5.2.3 Looking Up Resource and EJBs

Although we can use the `@EJB` or `@Resource` annotation to inject resource instances you may still need to lookup items from JNDI in several advanced cases (if you are unfamiliar with JNDI itself, check out the brief tutorial in the Appendix). You can use @EJB or @Resource annotation at the EJB class level to define dependency on an EJB or a resource. There are two ways of using programmatic lookups—either using the EJB context or a JNDI initial context. We'll look at both methods.

Recall from our earlier discussion that you can lookup any object stored in JNDI using the `EJBContext.lookup` method (including Session Bean references).

This technique can be used to accomplish one extremely powerful feature that DI cannot accomplish. Using lookups instead of DI allows you to determine what resource to use dynamically at runtime

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

instead of being constrained to using static configuration that cannot be changed programmatically. All you have to do is specify a different name in the lookup method to retrieve a different resource. As a result, program logic driven by data and/or user input can determine dependencies instead of deploy-time configuration.

The following code shows the usage of the EJB context lookup method:

```
@EJB(name="ejb/BidderAccountCreator", beanInterface =
BidderAccountCreator.class)
@Stateless
public class GoldBidderManagerBean implements GoldBidderManager {
@Resource SessionContext sessionContext;
...
BidderAccountCreator accountCreator
    = (BidderAccountCreator)
        sessionContext.lookup(
            "ejb/BidderAccountCreator");
...
accountCreator.addLoginInfo(loginInfo);
...
accountCreator.createAccount();
```

Note that while using the lookup method, you must explicitly specify the complete JNDI path name yourself. Also note that once an EJB context is injected as in the sample lookup code, it could be passed into any non-bean POJO to perform the actual lookup.
While both DI and lookup using the EJB context are relatively convenient, the problem is that they are only available inside the Java EE container (or an application client container). For POJOs outside a container, you are limited to the most basic method of looking up JNDI references—using a JNDI initial context. The code to do so is a little mechanical, but it really isn't too complex. Here is how the lookup code looks like:

```
Context context = new InitialContext();
BidderAccountCreator accountCreator
    = (BidderAccountCreator)
        context.lookup("java:comp/env/ejb/BidderAccountCreator");
...
accountCreator.addLoginInfo(loginInfo);
...
accountCreator.createAccount();
```

The `InitialContext` object can be created by any code that has access to the JNDI API. Also the object can be used to connect to a remote JNDI server, not just a local one. Although this code probably looks harmless enough, you should avoid it if at all possible. Mechanical JNDI lookup code was one of the major pieces of avoidable complexity in EJB 2.x, particularly when these same bits of code are repeated hundreds of times across an application.

Believe it or not, this is all we needed to say about EJB 3.0 DI and JNDI lookups. In the next Section, we will cover one of the most exciting new features in EJB 3.0—interceptors.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

5.3 AOP in the EJB World: Interceptors

Have you ever in a situation that requirements changing almost towards the end of the project and you were asked to add some common missing feature such as logging or auditing for EJBs in your application. It will be too much boring exercise to go and add logging code in each of your EJB classes. This type of common code also causes maintainability issues and requires modifying a lot of java classes for minor changes. Well, EJB 3.0 Interceptors solves this probelm. For example, in this situation, you create a logging interceptor that does the logging and you can make this as the default interceptor for your application. The logging interceptor will be executed when any bean method is executed and does the logging. If the requirement for logging changes then just change only one class. In section, we're going to see how they work. First we will provide an introduction to AOP, EJB 3.0 interceptors , defining an interceptor for an EJB and building of a business method interceptor. You may remember that we briefly discussed in Chapter 3 that you can define lifecycle-callbacks in external classes and we will learn how you can use interceptors configuration to define external lifecycle callback listeners.

5.3.1 What is AOP?

It is very likely you have at least briefly come across the term *Aspect-Oriented Programming (AOP)*. The essential idea behind AOP is that for most applications, there is common code repeated across methods that is not necessarily directly involved in solving the core business problem, but has to do with infrastructure concerns.

The most commonly cited example of this is logging, especially at the basic debugging level. To use an ActionBazaar example, let us assume that we log the entry into every method in the system. Without AOP, this would mean adding logging statements at the beginning of every single method in the system to log the action of "entering method XX"! Some other common examples where AOP applies are auditing, profiling, statistics and so on.
The common term used to describe these cases are called *crosscutting concerns*—concerns that cut across application logic. An AOP system allows the separation of crosscutting concerns into their own modules. These modules are then applied across the relevant cross-section of application code, such as the beginning of every method call. Tools like AspectJ have made AOP relatively popular. For great coverage of AOP, please read *AspectJ in Action* by Ravnivas Laddad.

EJB 3.0 supports AOP-like functionality by providing ability to intercept business methods and lifecycle callbacks. Buckle up and get ready to jump into the world of EJB 3.0 interceptors where you will learn what interceptors are and how to build business method and lifecycle callback interceptors.

5.3.2 What are Interceptors

Interceptors are essentially the EJB rendition of AOP. Interceptors are objects that are automatically triggered when an EJB method is invoked (believe it or not, Interceptors are not new concepts and date back to technologies like CORBA). While EJB 3.0 Interceptors provide good enough functionality to handle most common crosscutting concerns (such as our logging example), it does

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

not try to provide the level of functionality a full scale AOP package like AspectJ provides. On the flip side of things, EJB 3.0 Interceptors are also generally a lot easier to use.

Recall our discussion in Section 5.1 on how the EJB object provides services like transactions and security. In essence, the EJB object is essentially a very sophisticated built-in interceptor that provides a whole host of functionality. If you wanted to you could create your own EJB-esque services using Interceptors if you wanted to.

In the pure AOP world, interception happens at various points (called *point cuts*) including at the beginning of a method, at the end of a method, when an exception is triggered and so on. If you are familiar with AOP, an EJB Interceptor is the most general form of interception—it is an around invoke advice. EJB 3.0 Interceptors are triggered at the beginning of a method, is around when the method returns, can inspect the method return value or any exceptions thrown by the method. Interceptors can be applied to both Session and Message Driven Beans.

We will examine business method Interceptors further by implementing basic logging on the `PlaceBid` Session Bean presented in Chapter 2. Once you understand how this works, applying it to a Message Driven Bean should be a snap. Figure 5.3 depicts a business method interceptor that implements common logging code in the ActionBazaar application.

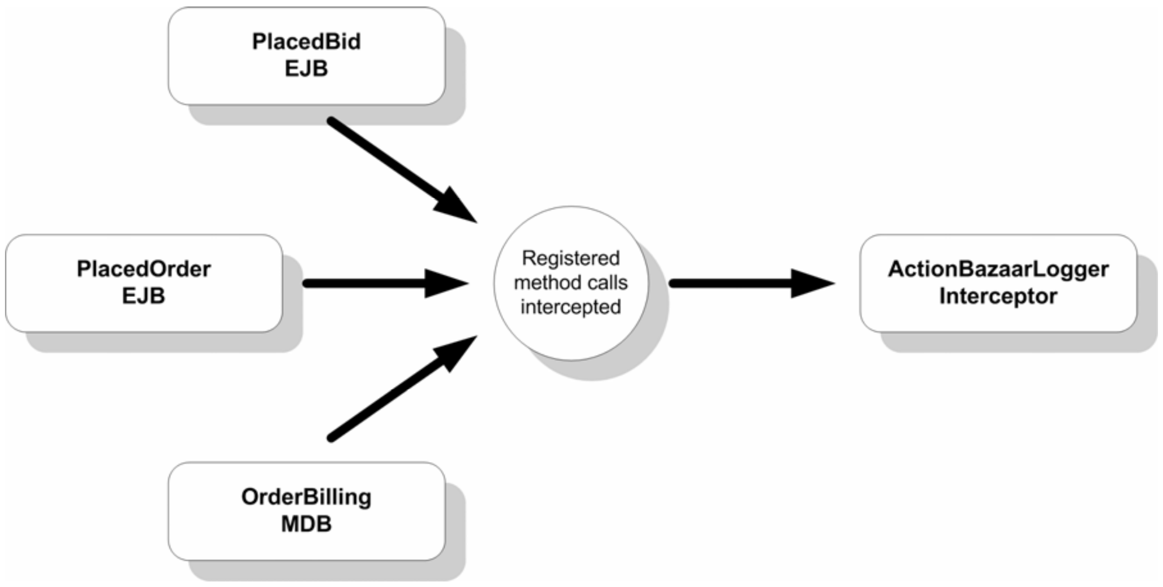

PlacedBid EJB
PlacedOrder EJB
OrderBilling MDB
Registered method calls intercepted
ActionBazaarLogger Interceptor

Figure 5.3: Business Interceptors are typically used to implement common code. The ActionBazaarLogger implements common logging code used by all EJBs in the ActionBazaar system.

Listing 5.1 details how the code looks like. The Interceptor attached to the `addBid` method will print a log message out to the console each time the method is invoked. In a real-world application, this could be used as debugging information (and perhaps printed out using `java.util.logging` or Log4J).

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Listing 5.1: EJB business method interceptors

```
@Stateless
public class PlaceBidBean implements PlaceBid {
    ...
    @Interceptors(ActionBazaarLogger.class)                          |#1
    public void addBid(Bid bid) {
        ...
    }
}

public class ActionBazaarLogger {
    @AroundInvoke                                                    |#2
    public Object logMethodEntry(
        InvocationContext invocationContext)
            throws Exception {
        System.out.println("Entering method: "
            + invocationContext.getMethod().getName());
        return invocationContext.proceed();
    }
}
```

(annotation) <#1 Attaching the interceptor>
(annotation) <#2 Specifying the interceptor method>

We will take a bird's eye view of this code first before analyzing each feature in detail in the coming Sections. The Interceptor Class, `ActionBazaarLogger` is attached to the `addBid` method of the `PlaceBid` Stateless Session bean using the `@javax.interceptor.Interceptors` annotation#1. The `ActionBazaarLogger` object's `logMethodEntry` method is annotated with the `@javax.interceptor.AroundInvoke` annotation and will be invoked when the `addBid` method is called#2. The `logMethodEntry` method prints out a log message to the system console including the method name entered using the `javax.interceptor.InvocationContext`. Finally, the invocation context's `proceed` method is invoked to signal to the container that the `addBid` invocation can proceed normally.
We will now start a detailed analysis of the code, starting with attaching the interceptor using the `@Interceptors` annotation.

5.3.3 Specifying interceptors

The `@Interceptors` annotation allows us to specify one or more Interceptor classes for a method or Class. In Listing 5.2 we attach a single interceptor to the `addBid` method:

```
@Interceptors(ActionBazaarLogger.class)
public void addBid (...
```

The `@Interceptors` annotation can also be applied to an entire class. When the `@Interceptors` annotation is applied to a class, the Interceptor would be triggered if any of the target class's methods are invoked. For example if the `ActionBazaarLogger` is applied at the class level as in the following code, our `logMethodEntry` method will be invoked when either the `PlaceBid` Class's `addBid` or `addTimeDelayedBid` methods are called by the client (imagine that the `addTimeDelayedBid` method adds a bid after a specified interval of time):

```
@Interceptors(ActionBazaarLogger.class)
@Stateless
public class PlaceBidBean implements PlaceBid {
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    public void addBid (...
    public void addTimeDelayedBid (...
}
```

As we mentioned, the `@Interceptors` annotation is fully capable of attaching more than one interceptor either at a class or method level. All you have to do is provide a comma-separated list as parameter to the annotation. For example, a generic logger and a bidding statistics tracker could be added to the `PlaceBid` Session Bean as follows:

```
@Interceptors({ActionBazaarLogger.class, BidStatisticsTracker.class})
public class PlaceBidBean { ... }
```

Besides specifying method and class level interceptors, we may also create what is called a *default Interceptor*. A default interceptor is essentially a "catch-all" mechanism that attaches to all methods of all Beans in the EJB module. Unfortunately, you cannot specify these kind of Interceptors using annotations and must use deployment descriptor settings instead. We will not discuss deployment descriptors in any great detail at this point but will show you how setting the `ActionBazaarLogger` class as a default interceptor for the ActionBazaar application might look like:

```
<assembly-descriptor>
    <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>
            actionbazaar.buslogic.ActionBazaarLogger
        </interceptor-class>
    </interceptor-binding>
</assembly-descriptor>
```

An interesting question that might have already crossed your mind is what happens if we specify a default, class and method level Interceptors for a specific target method (yes, this is perfectly legal). In which order do you think the interceptors would be triggered?

Somewhat counterintuitive to how Java scoping typically works out, the interceptors are called from the larger scope to the smaller scope. That is, the default interceptor is triggered first, then the class level interceptor and finally the method interceptor. Figure 5.4 shows this behavior.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

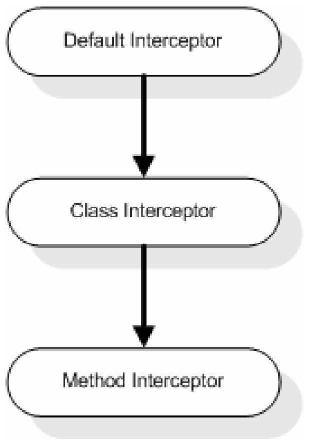


Figure 5.4: The order in which business method Interceptors are invoked. Default Interceptors apply to all methods of all EJBs in an ejb-jar package. Class level Interceptors apply to all methods of a specific class. Method level Interceptors apply to one specific method in a class. Default application level Interceptors are invoked first, then class level Interceptors, then method level Interceptors.

If more than one interceptor is applied at any given level, they are executed in the order that they are specified. In our `ActionBazaarLogger` and `BidStatisticsTracker` example, the `ActionBazaarLogger` is executed first since it appears first in the comma-separated list in the `@Interceptors` annotation:

```
@Interceptors({ActionBazaarLogger.class, BidStatisticsTracker.class})
```

Unfortunately, the only way to alter this execution order is using interceptor-order element in deployment descriptor and there is no annotations to change interceptor order.. However you can disable Interceptors at the default or class levels if you need to. Applying the `@javax.interceptor.ExcludeDefaultInterceptors` annotation on either a class or a method disables all default interceptors on the class or method. Similarly the `@javax.interceptor.ExcludeClassInterceptors` annotation disables class level Interceptors for a method. For example, both default and class level Interceptors may be disabled for the `addBid` method using the following code:

```
@Interceptors(ActionBazaarLogger.class)
@ExcludeDefaultInterceptors
@ExcludeClassInterceptors
public void addBid (...
```

Having looked at how to specify Interceptors, we will take a detailed look the Interceptor classes themselves next.

5.3.4 Implementing business Interceptors

Like the EJB lifecycle callback methods that we discussed in Chapters 3 and 4, business Interceptors can either be implemented in the bean class itself or separate classes. However we recommend that you create Interceptor methods external to the bean class because it allows you to separate crosscutting concerns from business logic and shared them between multiple beans. After all isn't that the whole point of AOP?

As we can see from Listing 5.2, following the general EJB 3.0 philosophy, an Interceptor class is simply a POJO that may have a few annotations.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Around Invoke Methods

What is important to note from the Listing is that an Interceptor must always have only one method that is designated as the *around invoke* (@AroundInvoke) *method*. @AroundInvoke methods must NOT be a business method, which means that it should not be a public method in the bean's business interface(s).

An around invoke method is automatically triggered by the container when a client invokes a method that has designated it to be its Interceptor. In Listing 5.2, the triggered method is marked with the `@AroundInvoke` annotation:

```
@AroundInvoke
public Object logMethodEntry(
    InvocationContext invocationContext)
        throws Exception {
    System.out.println("Entering method: "
            + invocationContext.getMethod().getName());
    return invocationContext.proceed();
}
```

In effect, this means that the `logMethodEntry` method would be executed whenever the `ActionBazaarLogger` Interceptor is triggered. As you might gather from the preceding code, any method designated `AroundInvoke` must follow this pattern:

```
public Object <METHOD>(InvocationContext) throws Exception
```

The `InvocationContext` interface passed in as the single parameter to the method provides a number of features that makes the AOP mechanism extremely flexible. The `logMethodEntry` method uses just two of the methods included in the interface. The `getMethod().getName()` call returns the name of the method being intercepted–'addBid' in our case.

The call to the `proceed` method is extremely critical to the functioning of the Interceptor. In our case, we always return the object returned by the `InvocationContext.proceed()` in the `logMethodEntry` method. This tells the container that it should proceed to the next Interceptor in the execution chain or call the intercepted business method. On the other hand, not calling the proceed method will bring processing to a halt and avoid the business method (and any other Interceptor down the execution chain) from being called.

This feature can be extremely useful for things like security validation. For example, the following Interceptor method prevents the intercepted business method from being executed if security validation fails:

```
@AroundInvoke
public Object validateSecurity(InvocationContext invocationContext)
    throws Exception {
    if (!validate(...)) {
        throw new SecurityException("Security cannot be validated. " +
            "The method invocation is being blocked.");
    }
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    return invocationContext.proceed();
}
```

The InvocationContext interface

The `InvocationContext` interface has a number of other very useful methods. Here is the definition of the interface:

```
public interface InvocationContext {
    public Object getTarget();
    public Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[]);
    public java.util.Map<String,Object> getContextData();
    public Object proceed() throws Exception;
}
```

The `getTarget` method retrieves the bean instance that the intercepted method belongs to. This method is particularly useful for checking the current state of the bean through its instance variables or accessor methods.

> The `getParameters` method returns the parameters passed to the intercepted method as an array of objects. The `setParameters` method, on the other hand, allows us to change these values at runtime before they are passed to the method. These two methods are extremely useful for interceptors that manipulate bean parameters to change behavior at runtime.

An Interceptor in ActionBazaar to transparently round off all monetary values to two decimal places for all methods across the application could use the `getParameters` and `setParameters` methods to accomplish its task.

The key to understanding the need for the `InvocationContext.getContextData` method is the fact that contexts are shared across the interceptor chain for a given method. As a result, data attached to an `InvocationContext` can be used to communicate between Interceptors. For example, let us assume that our security validation Interceptor stores the member status into invocation context data after the user is validated. This can be done as follows:

```
invocationContext.getContextData().put("MemberStatus", "Gold");
```

As we can see, the invocation context data is simply a `Map` used to store name-value pairs. Another Interceptor in the invocation chain can now retrieve this data and take specific actions based on the member status. For example, a discount calculator Interceptor can reduce the ActionBazaar item listing charges for a Gold member. The code to retrieve the member status would look like the following:

```
String memberStatus =
    (String) invocationContext.getContextData().get("MemberStatus");
```

Following is the `AroundInvoke` method of the `DiscountVerifierInterceptor` that actually uses the invocation context as well as most of the methods we discussed earlier:

```
@AroundInvoke
public Object giveDiscount(InvocationContext context)
    throws Exception {
    System.out.println("*** DiscountVerifier Interceptor"
        + " invoked for " + context.getMethod().getName() + " ***");
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    if (context.getMethod().getName().equals("chargePostingFee")
        && (((String)(context.getContextData().get("MemberStatus")))
            .equals("Gold"))) {
        Object[] parameters = context.getParameters();
        parameters[2] = new Double ((Double) parameters[2] * 0.99);
        System.out.println (
            "*** DiscountVerifier Reducing Price by 1 percent ***");
        context.setParameters(parameters);
    }

    return context.proceed();
}
```

We may throw or handle a runtime or checked exception in a business method Interceptor. If a business method interceptor throws an exception before invoking the `proceed` method, the processing of other Interceptors in the invocation chain and the target business method will be terminated.

Recall our discussion on life-cycle callback methods in Chapter 3 and 4. Although this isn't readily obvious, life-cycle callbacks are a form of interception as well. While method Interceptors are triggered when a business method is invoked, life cycle callbacks are triggered when a bean transitions from one life-cycle state to another. Although this was not the case in our previous life-cycle examples, in some cases life-cycle callback methods can be used for crosscutting concerns that can be shared across beans such as logging and profiling. For this reason, you can define life-cycle callbacks in Interceptor classes in addition to business method Interceptors. Let's take a look at how to do this next.

5.3.5 Lifecycle callback methods in the Interceptor class

Recall that the `@PostConstruct`, `@PrePassivate`, `@PostActivate` and `@PreDestroy` annotations can be applied to bean methods to receive life-cycle callbacks. When applied to Interceptor class methods, lifecycle callbacks work in exactly the same way. Lifecycle callbacks defined in an interceptor class are known as *lifecycle callback interceptors* or *lifecycle callback listeners*. When the target bean transitions lifecycles, annotated methods in the Interceptor class is triggered.

The following Interceptor class logs when ActionBazaar beans allocate and release resources when beans instances are constructed and destroyed:

```
public class ActionBazaarResourceLogger {
    @PostConstruct
    public void initialize (InvocationContext context) {
        System.out.println ("Allocating resources for bean: "
            + context.getTarget());
        context.proceed();
    }

    @PreDestroy
    public void cleanup (InvocationContext context) {
        System.out.println ("Releasing resources for bean: "
            + context.getTarget());
        context.proceed();
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    }
}
```

As the code sample shows, lifecycle Interceptor methods cannot throw checked exceptions (it doesn't really make sense since there is no client for lifecycle callbacks to bubble a problem up to).

Note a bean can have the same lifecycle callbacks both in the bean itself as well as in one or more Interceptors. That is the whole point of calling the `InvocationContext.proceed` method in lifecycle Interceptor methods as in the resource logger code. This makes sure that the next lifecycle Interceptor method in the invocation chain or the bean lifecycle method is triggered. There is absolutely no difference between applying an Interceptor class with or without lifecycle callbacks. The resource logger, for example is applied as follows:

```
@Interceptors({ActionBazaarResourceLogger.class})
public class PlaceBidBean { ... }
```

You might find that you will use lifecycle callbacks as bean methods to manage resources a lot more often than using Interceptor lifecycle callbacks to encapsulate crosscutting concerns like logging, auditing and profiling. However, Interceptor callbacks are extremely useful when you need them.

As a recap, Table 5.2 contains a summary of both business method Interceptors and lifecycle callbacks.

Table 5.2 Differences between lifecycle and business method interceptors. Lifecycle interceptors are created to handle EJB lifecycle callbacks. Business method interceptors are associated with business methods and are automatically invoked when a user invokes the business method.

	Lifecycle callback methods	**Business method interceptor**
	Gets invoked when a certain lifecycle event occurs.	Gets invoked when a business method is called by a client.
Location	In a separate Interceptor class or in the bean class.	In the class or an interceptor class.
Method signature	public void <METHOD>(InvocationContext) – in a separate Interceptor Class. public void <METHOD>() – in the bean class.	public Object <METHOD>(InvocationContext) throws Exception
Annotation	@PreDestroy, @PostConstruct, @PrePassivate, @PostActivate	@AroundInvoke
Exception handling	May throw run-time exceptions but must not throw checked exceptions. May catch and swallow exceptions. No other lifecycle callback methods are called if an exception is thrown.	May throw application or run time exception. May catch and swallow runtime exceptions. No other business Interceptor methods or the business method itself are called if an exception is thrown before calling the proceed method.
Transaction and security context	No security and transaction context.	Share the same security and transaction context within which the original business method was invoked.

This is all that we want to say about Interceptors right now. Clearly, Interceptors are an extremely important addition to EJB. It is very likely that the AOP features in future releases of EJB will get more and more robust. Interceptors certainly have the potential to evolve into a robust way of extending the EJB platform itself; with vendors offering new out-of-the-box Interceptor based services.

We will now move onto the final vital EJB 3.0 feature we will cover in this Chapter– the Timer service. Timers can be used only by Stateless Session Beans and Message Driven Beans.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

5.4 Scheduling: the EJB 3.0 Timer Service

Scheduled tasks are a reality for most non-trivial applications. For example, your business application may have to run a daily report to determine inventory levels and automatically send out restocking requests. For most legacy applications is it typical to have a batch job to cleanup temporary tables at the start or end of each day. If fact, it is fair to say schedulers are an essential holdover from the days of big iron batch computing. As a result, scheduling tools, utilities and frameworks have been a development mainstay for a long time. UNIX cron is probably the most popular and well-loved scheduling utility. The System Task Scheduler, generally lesser known, is the Microsoft Windows counterpart of cron.

In the Java EE world, you have a few options for scheduling tasks and activities. Most Java EE application servers come with a scheduling utility that is sufficiently useful. There are also a number of feature-rich, full-scale scheduling packages available for enterprise applications. *Flux* is an excellent commercial scheduling package, while *Quartz* is a good quality open source implementation. EJB Timer services are the standard Java EE answer to scheduling. As you will see in this part of the Chapter, while it does not try to compete with full-scale scheduling products, the EJB 3.0 Timer service is probably sufficient for most day-to-day application development requirements. Because it is so lightweight, the EJB 3.0 Timer service is also extremely easy to use.

In the next few Sections, we will take a look at how EJB 3.0 Timers work. We will build scheduling service using EJB 3.0 timers and hence learn use of @Timeout annotation. Finally we will brainstorm the ideal circumstances to use timers.

5.4.1 What are Timers?

In a sense, the EJB 3.0 Timer service is based on the idea of time-delayed callbacks. In other words, the EJB Timer service allows you to specify a method (appropriately called the *timeout method*) that is automatically invoked after a specified interval of time. The container invokes the timeout method on your behalf when the time interval you specify elapses. As we will see, we can use the Timer service to register for callbacks triggered once at a specific time or at regular intervals.

> We can only use Timers in Stateless Session Beans and Message Driven Beans because of their asynchronous, stateless nature. However, unlike Stateless Session Beans and MDBs, Timers are persistent and can survive a container crash or restart. Timers are also *transactional*, that is, a transaction failure in a timeout method rolls back the actions taken by the Timer.

Figure 5.5 illustrates how Timers work.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

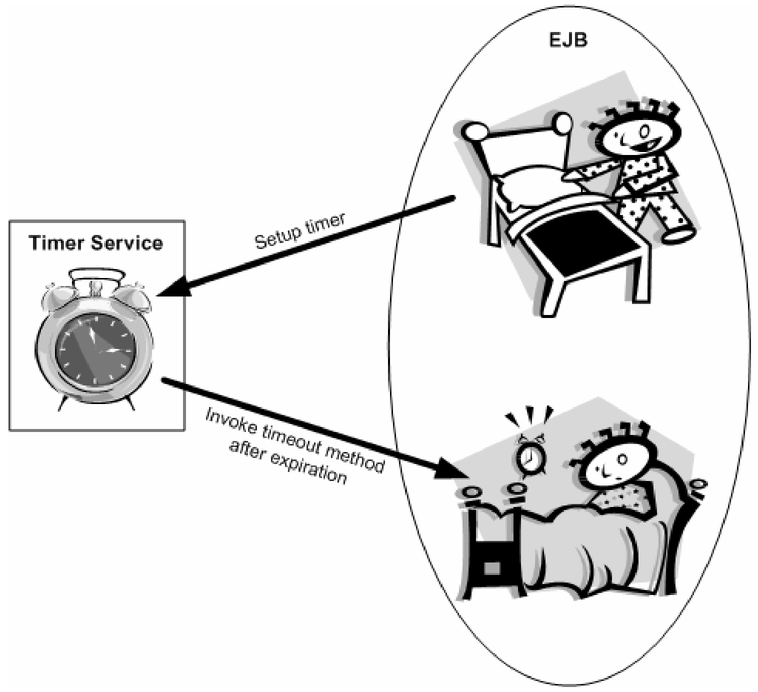


Figure 5.5: How an EJB Timer works. A client may invoke an EJB method that creates a 'Timer' that registers a callback in the EJB Timer Service. The EJB container invokes the timeout method in the bean instance when the Timer expires.

As the Figure demonstrates, an EJB method can register a time-driven callback with the container Timer service. When the time interval specified by the EJB expires, the Timer service invokes the timeout method pointed to by the EJB. We will see how this works by way of a simple example next.

5.4.2 Using the Timer Service

We will explore the features of the EJB 3.0 Timer services by adding a Timer to the `PlaceBid` EJB we introduced in Chapter 2. We will add a Timer in the `addBid` method to check the status of the newly placed bid every fifteen minutes. Although we won't code it, another very compelling use case is to create a Timer when an item is added for bidding. Such a timer could be triggered when the auction time expires and determine who the winning bidder is. We'll leave the implementation of this Timer as an exercise for you.

Among other things the Timer we will implement would notify the bidder via email if they have been outbid. We have omitted most of the code that is not absolutely necessary to explain Timer functionality in Listing 5.2. The complete code is included in the downloadable code samples if you are interested in exploring further.

Listing 5.2: Using the EJB 3.0 Timer service

```
public class PlaceBidBean implements PlaceBid {
    ...
    @Resource TimerService timerService;                         |#1
    ...
    public void addBid(Bid bid) {
        ... Code to add the bid ...
        timerService.createTimer(15*60*1000, 15*60*1000, bid);   |#2
        ...
    }
    ...
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @Timeout                                                      |#3
    public void monitorBid(Timer timer) {
        Bid bid = (Bid) timer.getInfo();
        ... Code to monitor the bid ...
    }
}
```
(annotation) <#1 Timer service injected>
(annotation) <#2 Timer created>
(annotation) <#3 Timeout method>

We will not explore this code in close detail right now, but will do a brief "fly over".

We use EJB 3.0 resource injection to get access to the Timer service#1. In the `addBid` method, after we add the bid, we schedule a Timer service callback to occur every fifteen miniues#2. The newly added `Bid` is attached as Timer info when the timer is registered. At regular intervals, the `monitorBid` method is called by the Timer service, which is designated with the `@Timeout` annotation. The `monitorBid` method retrieves the `Bid` instance attached as Timer info and monitors the bid.

We will explore EJB timer services details using Listing 5.2 as a jump off point in the next few sections, starting with the ways to get access to the EJB 3.0 Timer Service.

Accessing the Timer Service

As we just saw in Listing 5.2, the EJB Timer service can be injected into a Java EE component using the `@Resource` annotation. Alternatively, you can also get access to the container Timer service through the EJB context:

```
@Resource SessionContext context;
...
TimerService timerService = context.getTimerService();
```
Which method you choose is largely a matter of taste. In general, if you are already injecting an EJB context, you should avoid injecting the Timer Service too in order to avoid redundant code. Instead, you should use the `getTimerService` method like the preceding code. However, if you are not using the EJB context for anything else, it makes perfect sense to simply inject the `TimerService` as in Listing 5.2.

We will take a closer look at the injected Timer service itself next.

Using the TimerService interface

In Listing 5.2, we use the `TimerService` interface to register a `Timer#2`. As we will soon see, a `Timer` is simply a Java EE representation of a scheduled task. The `createTimer` method used in Listing 5.2 is one of four overloaded methods provided in the `TimerService` interface to add `Timers`. The one we used specified that the `Timer` should initially trigger in 15*60*1000 milliseconds (15 minutes), repeat every 15*60*1000 milliseconds (15 minutes) and added a `Bid` instance as `Timer` info:

```
timerService.createTimer(15*60*1000, 15*60*1000, bid);
```
Let's take a look at the complete definition of the `TimerService` interface to get clearer picture of the range of options available:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Listing 5.3: Specification for the TimerService interface is used to create either single-event or recurring timers

```
public interface javax.ejb.TimerService {
    public Timer createTimer(long duration,                             |#1
        java.io.Serializable info);                                     |#1
    public Timer createTimer(long initialDuration,                      |#2
        long intervalDuration, java.io.Serializable info);              |#2
    public Timer createTimer(java.util.Date expiration,                 |#3
        java.io.Serializable info);                                     |#3
    public Timer createTimer(java.util.Date initialExpiration,          |#4
        long intervalDuration, java.io.Serializable info);              |#4
    public Collection getTimers();                                      |#5
}
```

(annotation) <#1 Single-event timer with initial timeout >
(annotation) <#2 Recurring timer with initial timeout >
(annotation) <#3 Single-event timer with expiration time >
(annotation) <#4 Recurring timer with expiration time >
(annotation) <#5 Retrieve list of timers >

The first version of the `createTimer`#1 method allows us to create a single-event timer that is fired only once and not repeated. The first parameter, `duration` specifies the time, in milliseconds, after which the timeout method should be invoked. The second parameter, `info` allows us to attach an arbitrary piece of information to the timer. Note that timer info objects must always be `Serializable`, as is the `Bid` object we used in Listing 5.2. Note also that the `info` parameter can be left `null` if it is not really needed.

We have already seen the second version of the `createTimer`#2 method in action in Listing 5.2. It allows us to create recurring timers with initial timeout and interval durations set in milliseconds. The third version#3 is very similar to the first version in that it allows us to create a timer that fires once and only once. However, this version allows us to specify the `expiration` value as a specific instant in time represented by a `java.util.Date` instead of a `long` time offset. The fourth#4 and second versions of `createTimer` methods have differ from each other in the same way. Using a concrete date instead of an offset from the current time generally makes sense for events that should be fired farther out in the future. However, this is largely a matter of taste. All of these methods return a generated `Timer` reference. In general, this returned value is not used very often. Behind the scenes, all of the `TimerService` methods associate the current EJB as the callback receiver for the generated `Timers`. The final method of the `TimerService` interface, `getTimers#5`, retrieves all of the active `Timers` associated with the current EJB. This method is rarely used and we will not discuss it further.

Having looked at the `TimerService` interface and how to create timers, let's now take a closer look at how to implement timeout methods.

Implementing timeout methods

In Listing 5.2, we mark `monitorBid` to be the timeout method using the `@Timeout` annotation:

```
@Timeout
public void monitorBid(Timer timer) {
```

When the timer or timers created for the `PlaceBid` EJB expire, the container invokes the designated timeout method—`monitorBid`. Using the `@Timeout` annotation is by far the simplest, but not the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

only way to specify timeout methods. As you might have guessed, methods marked with the `@Timeout` annotation are expected to follow this convention:

```
public void <METHOD>(Timer timer)
```

The `Timer` for which the callback was invoked is passed in as a parameter for the method as processing context. This is because multiple `timers`, especially in case of repeating intervals, may invoke the same timeout method. Also, as we saw in Listing 5.2, it is often necessary to use the `TimerService` interface to pass around data to the timeout methods as `Timer` info.
We will finish off our analysis of the EJB 3.0 Timer service code by taking a closer look at the `Timer` interface next.

Using the Timer Interface

As we mentioned, the container passes us back the `Timer` instance that triggered the timeout method. In the `monitorBid` method, we use the interface to retrieve the `Bid` instance stored as timer info through the `getInfo` method:

```
@Timeout
public void monitorBid(Timer timer) {
    Bid bid = (Bid) timer.getInfo();
    ... Code to monitor the bid ...
}
```

There are a number of other useful methods defined in the `Timer` interface. We are going to explore them through the definition of the `Timer` interface below in listing 5.3:

Listing 5.3: The javax.ejb.Timer interface

```
public interface javax.ejb.Timer {
    public void cancel();

    public long getTimeRemaining();

    public java.util.Date getNextTimeout();

    public javax.ejb.TimerHandle getHandle();

    public java.io.Serializable getInfo();
}
```

The `cancel` method is particularly useful in canceling a timer prior to its expiration. We can use this method to stop timers prematurely. In our bid-monitoring example, we can use this method to stop the chain of recurring callbacks when bidding on the item is over.

> It is vital to invoke the `cancel` method for recurring `Timers` when they are no longer needed. Otherwise, the EJB will spin in an infinite loop unnecessarily. This is a subtle, common and easy mistake to make.

The `getTimeRemaining` method can be used on either a "single use" or interval timer. The return value of this method indicates the remaining time for the timer to expire, in milliseconds. You might find that this method is rarely used. The `getNextTimeout` method indicates the next time a recurring `Timer` will time out, as a `java.util.Date` instead of a `long` time-offset. Similar to the `getTimeRemaining` method, this method is useful in the rare occasion that you might need to determine whether or not to cancel a `Timer` based on when it will fire next.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The `getHandle` method returns a `Timer` handle. `TimerHandle` is a serialized object that we can store and obtain information about the `Timer` by using the `getTimer` method available through it. This is a relatively obscure method that we will leave for you to explore on your own if you need it. We have already seen the `getInfo` method#5 in action. As we've seen, this method is extremely useful in writing non-trivial timeout functions and accessing extra "processing information" attached to the `Timer` by the bean method creating the `Timer`.
Believe it or not, that is all there is to using the EJB 3.0 Timer service. We will now finish off this Section by discussing the situations where EJB `Timers` are an appropriate fit.

EJB Timers and Transactions

EJB Timers are transactional objects.
If the transaction that a timer is triggered under rolls back for some reason (as a result of a runtime exception in the timeout method for example) the timer creation is undone. In addition, the timeout method can be executed in a transactional context. You can specify a transactional attribute for the timeout method to be 'Required' or 'RequiresNew' and the container will start a transaction before invoking the timeout method. If the transaction fails the container will make sure the changes made by the failed method does not take effect and will retry the timeout method.
We will talk about EJB transactions in much greater detail in the next Chapter.

5.4.3 When to use EJB Timers

Clearly, although EJB Timers are relatively feature-rich, they are not intended to go toe-to-toe against full-fledged scheduling solutions like *Flux* or *Quartz*. However, under some circumstances, they are very sufficient if not ideal. Like almost all other technology choices, this decision comes down to weighting features against needs for your specific situation and environment.

Merits of Timers

The following are some of the merits of using EJB 3.0 Timers:

- Timers are part of the EJB specification. Hence, applications using EJB Timers will remain portable across containers instead of being locked into the non-standard APIs of job schedulers like Quartz.
- Since the EJB Timer service comes as a standard part of a Java EE application server there is no additional cost in terms of time or money to use it. No extra installation or configuration is required as would be the case for an external job scheduler, and you will not need to worry about integration and support.
- The Timer is a container-managed service. No separate thread pools or user threads are required for them, as would be the case with an external scheduler. For the very same reasons, the EJB Timer service is likely to have better out-of-the-box performance than third-party products.
- Transactions are fully supported with Timers (see the side bar titled 'EJB Timers and Transactions'), unlike external job schedulers, in which you may need to do extra setup for supporting JTA.
- By default, EJB Timers are persisted and survive EJB lifecycles and container restarts. The same cannot be said of all third-party schedulers.
-

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Limitations for Timers

The following are the primary limitations of EJB Timers:

- EJB Timers are meant for long-running business processes and not real-time applications where precision timing is absolutely critical. Commercial schedulers may provide much better guarantees in terms of precision than the EJB 3.0 Timer service.
- EJB Timers lack support for extremely flexible cron-type timers, blackout dates, workflow modeling for jobs and so on. These advanced features are commonly available with external job schedulers.
- There is no robust GUI admin tool to create, manage and monitor EJB 3.0 Timers. This is generally available for third-party job schedulers.

This concludes our analysis of EJB 3.0 Timers and marks the end of this Chapter.

In general, you should attempt to use EJB 3.0 Timers first. You should only resort to third-party schedulers if you run into serious limitations that cannot be easily overcome.

Although robust schedulers are a compelling idea, in general they are complex and should not be used frivolously. However, there are many complex, scheduling intensive applications where robust schedulers are a must, especially in industries like Banking and Finance.

5.5 Summary

In this Chapter, we covered a few advanced concepts common to all EJB types: such as EJB Object, EJB Context, using resources using dependency injection in EJB 3.0 and EJB Timer service.

The EJB Object acts as a proxy between clients and container where you can use EJBContext to access container runtime information and services.

Interceptors are lightweight AOP features in EJB 3.0 for dealing with cross cutting concerns such as logging and auditing. You can use interceptors either at the EJB module level, class level or method level.

EJB Timers provide a lightweight scheduling service that you can use in your applications.

You will likely find these advanced features very useful in moderate-sized real-life applications.
The only two features common to Session and Message Driven Beans that we did not cover in this Chapter are transaction and security management. We have decided to defer the discussion of these relatively involved topics to Chapter 6. This is a sensible ordering as it would be difficult to broach the topic of transaction and security management before understanding a few of the features introduced in this Chapter like Interceptors and managed components. We'll also defer the topic of EJB packaging to Chapter 11, after we finish talking about the Java Persistence API (JPA).

Let's dive into two of the most important EJB services—transactions and security in the next Chapter.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 6 Transactions and Security

Transaction and security management are very important aspects of any serious enterprise development effort. By the same token, both of these are system level concerns rather than true business application development concerns, which is why they often become an afterthought. In the worst-case scenario, these critical aspects of application development are overlooked altogether. With these facts in mind, EJB 3.0 provides functionality in both of these realms that is both robust enough for the most demanding environments, yet simple enough for those who prefer to focus on developing business logic. Although we have briefly mentioned these features in previous chapters, we have not dealt with them in any detail until this chapter. The first part of this chapter is devoted to exploring the rich transaction management features of EJB 3.0. We will briefly discuss about transactions and explore more about container managed and bean managed transactions support in EJB. The second part deals with security features of EJB where we briefly explore declarative and programmatic security support in EJB.

6.1 What Transactions Are

We engage in transactions almost every day when withdrawing money from an ATM or paying a phone bill. Transactions in computing are a closely related concept but differ slightly and are a little harder to define. In the most basic terms, a transaction is a grouping of tasks that must be processed as an inseparable unit. This means every task that is part of the transaction must succeed for the transaction to succeed. If any of the tasks fail, the transaction fails as well. You can think of a transaction as a three-legged wooden stool. All three legs must hold for the stool to stand. If any of them break, the stool collapses. In addition to this *all or nothing* value proposition, transactions must guarantee a degree of reliability and robustness. We will come back to exactly what this last statement means when we describe what are called the ACID properties of transactions. A successful transaction is *committed*, meaning its results are made permanent, whereas a failed transaction is *rolled back*, as if it never happened.

To explore transaction concepts further, let us take a look at an example problem in the ActionBazaar application. Before exploring transaction support in EJB, we will briefly discuss about ACID properties, transaction management concepts such as Resource Manager and Transaction Manager and two phase commit.

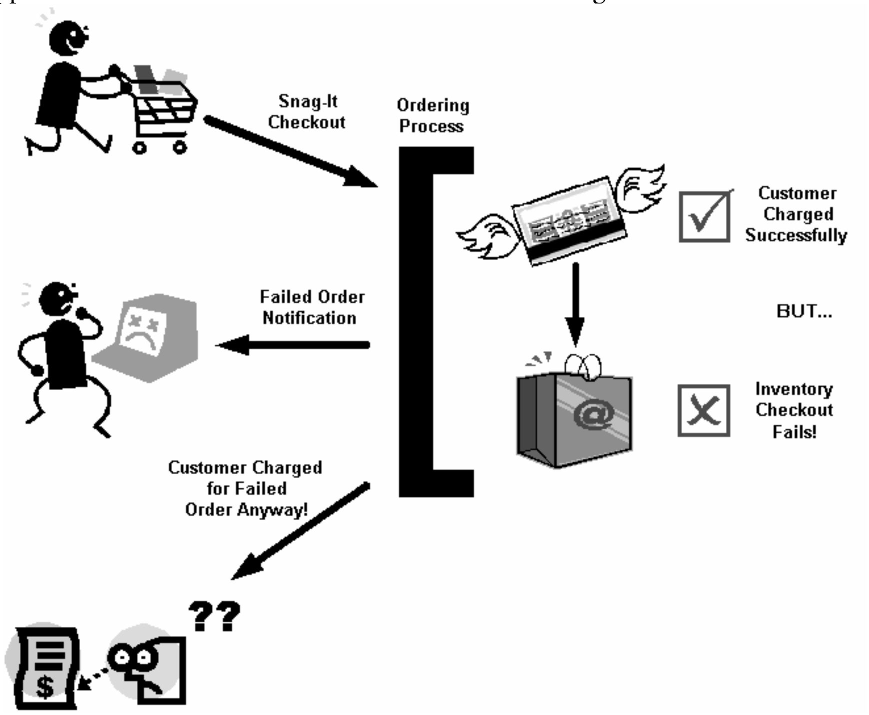
6.1.1 A Transactional Solution in ActionBazaar

Some items on ActionBazaar have a "Snag-It" ordering option. This option allows a user to purchase an item on bid at a set price before anyone else bids on it. As soon as the first bid is placed on an item, the "Snag-It" option disappears. This feature has become very popular because neither

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

the buyer nor the seller needs to wait for bidding to finish as long as they both like the initial "Snag-It" price tag. As soon as the user presses the "Snag-It" button, the ActionBazaar application makes sure no bids have been placed on the item, validates the buyer's credit card, charges the buyer and removes the item from bidding. Imagine what would happen if one of these four actions failed due to a system error but the rest of the actions were allowed to succeed. For example, let us assume that we validate and charge the customers credit card successfully. However, the order itself fails because the operation to remove the item from bid fails due to a sudden network outage and the user receives an error message. Since the credit card charge was already finalized, the customer is billed for a failed order! To make matters worse, the item would remain available for bidding. Another user could put a bid on the item before anyone could fix the problem, creating an interesting situation for the poor customer support folks to sort out! We can see this situation in figure 6.1.


Snag-It Checkout
Ordering Process
Customer Charged Successfully
BUT...
Failed Order Notification
Inventory Checkout Fails!
Customer Charged for Failed Order Anyway!
??

39 **Figure 6.1: Because of the fact that the ordering process is not covered by a transaction, ActionBazaar reaches a strange state when a Snag-It order fails halfway through. The customer is essentially billed for a failed order!**

While creating ad-hoc application logic to automatically credit the customer back in case of an error is a band-aid to the problem, transactions are ideally suited to handle such situations. A transaction covering all of the ordering steps would ensure that no actual ordering operation changes are finalized until the entire operation finishes successfully. If any errors occur, all pending data changes, including the credit card charge, will be aborted. On the other hand, if all the operations succeed, the transaction will be marked successful and all ordering changes will be made permanent.

Although this "all or nothing" value proposition is a central theme of transactional systems, they are not their only attribute. There are a number of properties for transactional systems; we will discuss them next.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

6.1.2 ACID Properties

The curious acronym ACID stands for Atomicity, Consistency, Isolation and Durability. All transactional systems are said to exhibit these four characteristics. Let us take a look at exactly what each of these characteristics is.

Atomicity

As we have seen in our ActionBazaar scenario, transactions are atomic in nature; they either commit or rollback together. In coding terms, you band together an arbitrary body of code under the umbrella of a transaction. If something unexpected and irrecoverable happens during the execution of the code, the result of the attempted execution is completely undone so that it has no effect on the system. Otherwise, the results of a successful execution are allowed to become permanent.

Consistency

This is the trickiest of the four properties because it involves more than writing code. This is the most common way of describing the consistency property: if the system is in a state consistent with the business rules before a transaction begins, it must remain in a consistent state after the transaction is rolled back or committed. A corollary to this statement is that the system *need not be* in a consistent state *during* the transaction. Think of a transaction as a sandbox or sanctuary – you are temporarily protected from the rules while inside it. As long as we make sure all the business rules in the system remain intact after the last line of code in a transaction is executed, it does not matter if you are in an inconsistent state at an arbitrary point in the transaction. Using our example, it is fine if we charge the customer even though we really have not removed the item from bidding yet, because the results of our code will have no impact on the system until and unless our transaction finishes successfully. In the real world, setting up rules and constraints in the database (such as primary keys, foreign key relationships, field constraints and so on) ensures consistency, so that transactions encountering error conditions are rejected and the system is returned to its pre-transactional state.

Isolation

If you understand thread synchronization or database locking, you already know what isolation is. The isolation property makes sure transactions do not step on each other's toes. Essentially, the transaction manager (a concept we will define shortly) makes sure nobody touches your data while you are in the transaction. This concept is especially important in concurrent systems where any number of processes can be attempting to manipulate the same data at any given time. Usually isolation is guaranteed by using low-level database locks hidden away from the developer. The transaction manager places some kind of lock on the data accessed by a transaction so that no other processes can modify them until the transaction is finished.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Isolation Levels

The concept of isolation as they pertain to databases is not as cut and dried as we just suggested. As you might imagine, making transactions wait for each other's data locks limits the number of concurrent transactions that can run on a system. However, different isolation strategies allow for a balance between concurrency and locking, primarily by sacrificing lock acquisition strictness. Each isolation strategy corresponds to an *isolation level*. The following are the four most common isolation levels, from the highest level of concurrency to the lowest:

Read uncommitted: At this isolation level, your transaction can read the uncommitted data of other transactions – also known as a 'dirty' read. You should not use this level in a multithreaded environment.
Read committed: Your transaction will never read uncommitted changes from another transaction. This is the default level for most databases.
Repeatable read: The transaction is guaranteed to get the same data on multiple reads of the same rows until the transaction ends.
Serializable: This is the highest isolation level and guarantees that none of the tables you touch will change during the transaction, including adding new rows. This isolation level is very likely to cause performance bottlenecks.

A good rule of thumb is to use the highest isolation level that yields an acceptable performance level. Generally, you do not directly control isolation levels from EJBs – the isolation level is set at the database resource level

In terms of our example, the transaction isolation property is what makes sure that no bids can be placed on the item while we are in the middle of executing the "Snag-It" ordering steps since our "snagged" item record would be locked in the database.

Durability

The last of the four ACID properties is durability. Transaction durability means that a transaction, once committed, is guaranteed to become permanent. This is usually implemented by using transaction logs in the database server.[6] Essentially, the database keeps a running record of all data changes made by a transaction before it commits. This means that even if a sudden server error occurs during a commit, once the database recovers from the error, changes can be reverted to be properly reapplied (think of untangling a cassette tape and rewinding it to where the tape started tangling). Changes made during the transaction are applied again by executing the appropriate entries from the transaction log (replaying the rewound tape to finish). This property is the muscle behind transactions making sure 'commit' really does mean commit.

In the next section, we will examine the internals of transaction management and define concepts such as distributed transactions, transaction managers, and resource managers.

6.1.3 Transaction Management Internals

As you have probably already guessed, application servers and enterprise resources like the database management system do most of the heavy lifting in transaction management. Ultimately, everything that we do in code translates into low level database operations such as locking and unlocking rows or tables in a database, beginning a transaction log, committing a transaction by

[6] The application server can also maintain a transaction log. However, we will ignore this fact for the time being.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

applying log entries or rolling back a transaction by abandoning the transaction log. In enterprise transaction management, the component that takes care of transactions for *a particular resource* is called a *resource manager*. Remember that a *resource* need not just be a database like Oracle. It could be a message server like IBM MQSeries or an Enterprise Information System like PeopleSoft CRM.

Most enterprise applications only involve a single resource. A transaction that only uses a single resource is called a *local transaction*. However, many enterprise applications use more than one resource. If you look carefully at our "Snag-It" order example, it most definitely involves more than one database – the credit card provider's database used to charge the customer, as well as the ActionBazaar database to manage bids, items and ordering. It is fairly apparent that for sane business application development some kind of abstraction is needed to manage multiple resources in a single transaction. This is exactly what the *transaction manager* is – a component that, under the hood, coordinates a transaction over multiple distributed resources.

From an application's view, the transaction manager is the application server or some other external component that provides simplified transaction services. As we see in Figure 6.2, the application program (ActionBazaar) asks the Transaction Manager to start, commit, and rollback transactions. The *transaction manager* coordinates these requests among multiple resource managers and each transaction phase may translate to possibly numerous low-level resource commands issued by the *resource managers*.

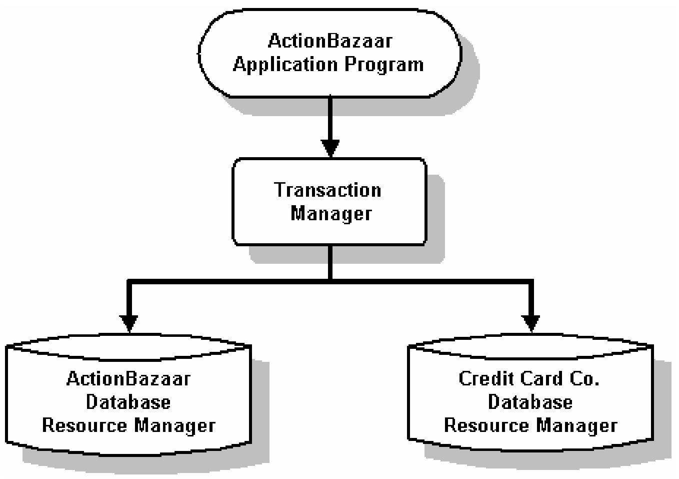

ActionBazaar Application Program
Transaction Manager
ActionBazaar Database Resource Manager
Credit Card Co. Database Resource Manager

Figure 6.1: Distributed Transaction Management. The application program delegates transaction operations to the transaction manager who coordinates between resource managers.

Next, we will discuss exactly how transactions are managed across multiple resources. In EJB, this is done with Two-Phase commits.

6.1.4 Two-Phase Commit

How transactions are managed in a distributed environment involving more than one resource is extremely interesting. The protocol commonly used to achieve this is called *two-phase commit*.

Imagine what would happen if no special precautions were taken while attempting to commit a transaction involving more than one database. Suppose that the first database commits successfully, but the second fails. It would be extremely difficult to go back and 'undo' the finalized changes to the first database. To avoid this problem, the two-phase commit protocol performs an additional

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

preparatory step before the final commit. During this step, each resource manager involved is asked if the current transaction can be successfully committed. If any of the resource managers indicate that the transaction cannot be committed if attempted, the entire transaction is abandoned (rolled back). Otherwise, the transaction is allowed to proceed and all resource managers are asked to commit. As we see in Table 6.1, only distributed transactions use the two-phase commit protocol.

1.9 Table 6.1: A transaction may be either local or global. A local transaction involves one resource and a global transaction involves multiple resources

	Local	**Global Transaction**
Number of resources	One	Multiple
Coordinator	Resource Manager	Transaction Manager
Commit protocol	Single-Phase	Two-Phase

1.10
1.11

We have just reviewed how transactions work and what makes them reliable, now we will take a look at how EJB provides these services for the application developer.

The XA Protocol

To coordinate the two-phase commit across many different kinds of resources, the transaction manager and each of the resource managers must "talk the same tongue" or use a common protocol. In the absence of such a protocol, imagine how sophisticated even a reasonably effective transaction manager would have to be. The transaction manager would have to be developed with the proprietary communication protocol of every supported resource.

The most popular distributed transaction protocol used today is the XA protocol, which was developed by the X/Open group. JAVA EE uses this protocol for implementing distributed transaction services.

6.1.5 Transaction Management in EJB

Transaction management support in EJB is provided through the Java Transaction API (JTA). JTA is a small, high-level API exposing functionality at the distributed transaction manager layer, typically provided by the application server. As a matter of fact, for the most part, as an EJB developer, you will probably only need to know about only one JTA interface — `javax.transaction.UserTransaction`. This is because the container takes care of most transaction management details behind the scenes. As an EJB developer, you simply tell the container where the transaction begins and ends (called transaction demarcation or establishing transaction boundaries) and whether to rollback or commit.

There are two ways of using transactions in EJB. Both provide abstractions over JTA, one to a lesser and one to a greater degree. The first is to declaratively manage transactions through Container-Managed Transactions (CMT) – this can be done through annotations or the deployment descriptor. On the other hand, Bean-Managed Transactions (BMT) require you to explicitly manage transactions programmatically. It is very important to note that in this version of EJB, only Session Beans and Message Driven Beans support BMT and CMT. The EJB 3.0 Java Persistence API is not

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

directly dependent on either CMT or BMT but can transparently plug into any transactional environment including BMT and CMT while used inside a Java EE container. We will cover this functionality when we discuss Persistence in Chapters 7, 8, 9, 10, 11 and 12. In this chapter, we explore CMT and BMT as they pertain to the two bean types we discussed in Chapter 3 (Session Beans) and Chapter 4 (Message Driven Beans).

JTS vs. JTA

These like-sounding acronyms are both related to Java EE transaction management. JTA defines application transaction services as well as the interactions between the application server, the transaction manager and resource managers. On the other hand, JTS (Java Transaction Service) deals with how a transaction manager is actually implemented. A JTS transaction manager supports JTA as its high-level interface and implements the Java mapping of the OMG Object Transaction Service (OTS) specification as its low-level interface.

As an EJB developer, there really is no need for you to deal with JTS.

Container-managed transactions are by far the simplest and most flexible way of managing EJB transactions. We will take a look at them first.

6.2 Container-Managed Transactions (CMT)

In CMT, the container starts, commits and rolls back a transaction on our behalf. Transaction boundaries in declarative transactions are always marked by the start and end of EJB business methods. More precisely, the container starts a JTA transaction before a method is invoked, invokes the method and depending on what happened during the method call, either commits or rolls back the managed transaction. All we have to do is tell the container how to manage the transaction by using either annotations or deployment descriptors and ask it to roll back the transaction when needed. By default, the container assumes that you will be using CMT on all business methods. This section describes CMT in action. We will build Snag-it ordering system using CMT and see usage of @TransactionManagement and @TransactionAttribute. Finally we will learn how to rollback a transaction using methods of EJBContext and when application exception is raised.

6.2.1 Snag-It Ordering Using CMT

Code Listing 6.1 implements the Snag-It ordering scenario as the method of a Stateless Session Bean using CMT. This is fine since the user can order only one item at a time using the Snag-It feature and no state information need be saved between calls to the `OrderManagerBean`. The bean first checks to see if there are any bids on the item, and if there are not, it validates the customer's credit card, charges the customer and removes the item from bidding. To keep the code sample as

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

simple as possible, we have omitted all details that are not directly necessary for our explanation of CMT.

14 Listing 6.1: Implementing Snag-It Using CMT

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
|#1
public class OrderManagerBean {
    @Resource                                                                  |#2
    private SessionContext context;                                            |#2
…
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
|#3
    public void placeSnagItOrder(Item item, Customer customer){
        try {
            if (!bidsExisting(item)){
                validateCredit(customer);
                chargeCustomer(customer, item);
                removeItemFromBidding(item);
            }
        } catch (CreditValidationException cve) {                             |#4
            context.setRollbackOnly();                                         |#4
        } catch (CreditProcessingException cpe){                              |#4
            context.setRollbackOnly();                                         |#4
        } catch (DatabaseException de) {                                      |#4
            context.setRollbackOnly();                                         |#4
        }                                                                      |#4
    }
}
```

(annotation) <#1 Bean Uses CMT>
(annotation) <#2 Injected EJB Context>
(annotation) <#3 Transaction Required for Method>
(annotation) <#4 Rollback Signal on Exception>

We will briefly describe the major features of Listing 6.1 now and explore CMT in detail using the example as a jump-off point in the coming sections. #1 tells the container that it should manage the transactions for this bean. If we do not specify the `TransactionManagement` annotation or the `transaction-type` element in the deployment descriptor, the container assumes that we intend to use CMT. The EJB context is injected into the bean in #2. #3 means that a transaction is required for the `placeSnagItOrder` method and one should be started by the container when needed. If an exception stops us from completing the Snag-It order, we ask the container to roll back the transaction using the injected `EJBContext` Object's `setRollbackOnly` method#4.

Let us first take a closer look at the `TransactionManagement` annotation#1 next on our way to analyzing the code in more detail.

6.2.2 The @TransactionManagement Annotation

The `@TransactionManagement` annotation specifies if CMT or BMT is to be used for a particular bean. In our case, we specify the value `TransactionManagementType.CONTAINER` - meaning the container should manage transactions on the bean's behalf. If we wanted to manage transactions programmatically instead, we would specify `TransactionManagementType.BEAN` for the `TransactionManagement` value. Notably, although we have explicitly included the annotation in our example, if we leave it out, the container will assume CMT anyway. When we explore BMT, it will be more obvious why CMT is the default and most commonly used choice for

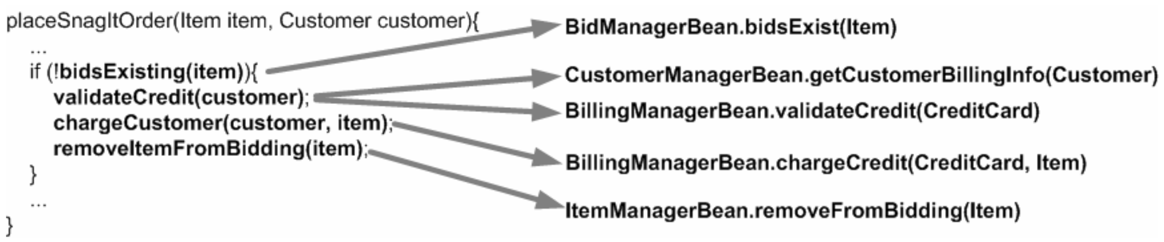
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

transaction management. We will take a look at the second transaction related annotation in Listing 6.1, `TransactionAttribute#3` next.

6.2.3The @TransactionAttribute Annotation

Although the container does most of the heavy lifting in CMT, you still need to tell the container how it should manage transactions. To understand what this means, consider the fact that the transaction that wraps around your bean's method could be started by the container specifically when calling your method or it could be inherited from a client calling your method (otherwise called *joining* a transaction). Let us explore this idea a little more using our example. The `placeSnagItOrder` method in Listing 6.1 calls a number of methods such as `bidsExisting`, `validateCredit`, `chargeCustomer` and `removeItemFromBidding`. As Figure 6.2 depicts, these method calls could simply be forwarded to other Session Bean invocations, such as `BidManagerBean.bidsExist`, `BillingManagerBean.validateCredit`, `BillingManagerBean.chargeCustomer` and `ItemManagerBean.removeFromBidding`.


placeSnagItOrder(Item item, Customer customer){
...
if (!bidsExisting(item)){
validateCredit(customer);
chargeCustomer(customer, item);
removeItemFromBidding(item);
}
...
}
BidManagerBean.bidsExist(Item)
CustomerManagerBean.getCustomerBillingInfo(Customer)
BillingManagerBean.validateCredit(CreditCard)
BillingManagerBean.chargeCredit(CreditCard, Item)
ItemManagerBean.removeFromBidding(Item)

41 **Figure 6.2: The method invocations from the CMT session bean is actually forwarded to other session beans that may be using various transaction attributes.**

42

We already know that the `placeSnagItOrder` method is managed by a transaction. What if all the Session Beans we are invoking are also managed by CMT? Should the container reuse the transaction created for our method to invoke the other methods? Should our existing transaction be independent of the other Session Bean's transactions? What happens if any of the methods cannot support transactions? The `@TransactionAttribute` annotation tells the container how to handle all these situations. The annotation can be applied either to individual CMT bean methods or to the entire bean. If the annotation is applied at the bean level, all business methods in the bean inherit the transaction attribute value specified by it. In Listing 6.1, we specify that the value of the `@TransactionAttribute` annotation for the `placeSnagItOrder` method should be `TransactionAttributeType .REQUIRED#3`. There are six different choices for this annotation defined by enumerated type `TransactionAttributeType`. Table 6.2 summarizes their behavior.

1.12 Table 6.2: Effects of transaction attributes on EJB methods

Transaction Attribute	Caller Transaction Exists?	Effect
Required	No	Container creates a new transaction.
	Yes	Method joins the caller's transaction.
Requires new	No	Container creates a new transaction.
	Yes	Container creates a new transaction and the client's transaction is suspended.
Supports	No	No transaction is used.
	Yes	Method joins the caller's transaction.
Mandatory	No	javax.ejb.EJBTransactionRequiredException is thrown.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

	Yes	Method joins the caller's transaction.
Not supported	No	No transaction is used.
	Yes	The client transaction is suspended and the method is called without a transaction.
Never	No	No transaction is used.
	Yes	javax.ejb.EJBException is thrown.

Let us take a look at what each of these values really mean and where they are applicable.

Required

This is the default and most commonly applicable transaction attribute value. This value means that the EJB method should always be invoked within a transaction. If the method is invoked from a non-transactional client, the container will start a transaction before the method is called and finish it when the method returns. On the other hand, if the caller invokes the method from a transactional context, the method will *join* the existing transaction. In case of transactions propagated from the client, if our method indicates that the transaction should be rolled back, the container will not only roll back the whole transaction, but will throw a `javax.transaction.RollbackException` back to the client. This lets the client know that the transaction it started has been rolled back by another method. Our `placeSnagItOrder` method is most likely invoked from a non-transactional web tier. Hence, the `REQUIRED` value in the `TransactionAttribute` annotation will cause the container to create a brand new transaction for us when the method is executed. If all the other Session Bean methods we invoke from our bean are also marked `REQUIRED`, when we invoke them, they will join the transaction created for us. This is just fine, since we want the entire ordering action to be covered by a single "umbrella" transaction. In general, you should use the `REQUIRED` value if you are modifying any data in your EJB method and are unsure if the client will start a transaction of its own before calling your method.

Requires New

The `REQUIRES_NEW` value means that the container must always create a new transaction to invoke the EJB method. If the client already has a transaction, it is temporarily *suspended* until our method returns. This means that the success or failure of our new transaction has no effect on the existing client transaction. From the client's perspective, its transaction is paused, our method is invoked, our method either commits or rolls back its own transaction and the client's transaction is resumed as soon as our method returns. The `REQUIRES_NEW` attribute has limited uses in the real world. You should use it if you need a transaction but do not want a rollback to affect the client. Vice-versa, you also want this value when you do not want the client's rollback to affect you. Logging is a great example. Even if the parent transaction rolls back, you want to be able to record the failure into your logs. On the other hand, failing to log a minor debugging message should not roll back your entire transaction and the problem should be localized to the logging component.

Mandatory

This really means *requires existing*. That is, the caller must have a transaction before calling an EJB method and the container should never create a transaction on behalf of the client. If the EJB method using the `MANDATORY` attribute is invoked from a non-transactional client, the container throws an `EJBTransactionRequiredException`. This value is also very rarely used. You should use this value if you want to make sure the client fails if you request a rollback. We can make

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

a reasonable case to require a MANDATORY transaction on a Session Bean method that charges the customer. After all, we want to make sure nothing is accidentally given away for free if the client neglects to detect a failure in the method charging the customer and the invoker's transaction can be forcibly rolled back by us when necessary.

Not Supported

If we specify NOT_SUPPORTED to be the transaction attribute, the EJB method cannot be invoked in a transactional context. If a caller with an associated transaction invokes the method, the container will suspend the transaction, invoke the method and then resume the transaction when the method returns. This attribute is typically useful only for an MDB supporting a JMS provider in non-transactional, auto-acknowledge mode. To recap from Chapter 5, in such cases, the message is acknowledged as soon as it is successfully delivered and the MDB has no capability or apparent need to support rolling back message delivery.

Supports

The SUPPORTS attribute essentially means the EJB method will inherit whatever the transactional environment of the caller is. If the caller does not have a transaction, the EJB method will be called without a transaction. On other hand, if the caller is transactional, the EJB method will join the existing transaction and will not cause the exiting transaction to be suspended. This avoids any needless overhead in suspending or resuming the client transaction. This attribute is typically useful for methods that perform read-only operations such as retrieving a record from a database table. In our Snag-It example, the Session Bean method for checking if a bid exists on the item about to be ordered can probably have a SUPPORTS attribute since it modifies no data.

Never

In CMT, NEVER really means NEVER. In other words, this attribute means that the EJB method can never be invoked from a transactional client. If such an attempt is made, a javax.ejb.EJBException is thrown. This is probably the least-used transaction attribute value. It could be used if your method is changing a non-transactional resource (such as a text file) and you want to make sure the client knows about the non-transactional nature of the method.

Transaction Attributes and MDB

As we mentioned in Chapter 4, MDBs do not support all of the six transaction attributes we have discussed. Although you can apply any of the attributes to a Stateful or Stateless Session Bean, MDBs only support the REQUIRED and NOT_SUPPORTED attributes. This goes back to the fact that no client ever invokes MDB methods directly. It is the container that invokes MDB methods when it receives an incoming message. Since there is never an existing client transaction to suspend or join, REQUIRES_NEW, SUPPORTS, MANDATORY make no sense (refer to Table 6.2). NEVER makes no sense either since we do not need that strong a guard against the container. In effect, depending on message acknowledgment on method return, we need only tell the container of two conditions: we need a transaction (REQUIRED) that encapsulates the message listener method; or we do not need transaction support (NOT_SUPPORTED).

So far, we have taken a detailed look at how transactions are created and managed by the container. We know that the successful return of a CMT method causes the container to commit a method or at least not roll it back if it is a joined transaction. We have mentioned the fact that a

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

CMT method can mark an available transaction as rolled back, but have not discussed the actual mechanics. Let us dig into the underpinnings next.

6.2.4 Marking a CMT Transaction for Rollback

If the appropriate business conditions arise, a CMT method can ask the container to roll back a transaction as soon as possible. The important thing to note here is that the transaction is not rolled back immediately, but a flag is set for the container to do the actual rollback when it is time to end the transaction. Let us go back to a snippet of our scenario in Listing 6.1 to see exactly how this is done:

```
@Resource
private SessionContext context;
...
public void placeSnagItOrder(Item item, Customer customer){
    try {
        ...
        validateCredit(customer);
        ...
    } catch (CreditValidationException cve) {
        context.setRollbackOnly();
        ...
```

As the code snippet shows, the `setRollbackOnly` method of the injected `javax.ejb.EJBContext` marks the transaction to be rolled back when we are unable to validate the user's credit card, a `CreditValidationException` is thrown and we cannot allow the order to complete. If you go back and look at the complete listing, we do the same thing in case of other serious problems, such as a database problem or if we have trouble charging the credit card.

To keep things simple, let us assume that the container started a new transaction because the `placeSnagItOrder` method was invoked from a non-transactional web-tier. This means that after the method returns, the container would check to see if it could commit the transaction. Since we set the roll back flag for the underlying transaction through the `setRollbackOnly` method, the container would roll back instead. Because the EJB context in this case is really a proxy to the underlying transaction, you should never call the `setRollbackOnly` method unless you are sure there is an underlying transaction to flag. Typically, you can only be sure of this fact if your method has a `REQUIRED`, `REQUIRED_NEW` or `MANDATORY` transaction attribute. If we are not using CMT or our method is not invoked in a transaction context, calling this method will throw `java.lang.IllegalStateException`.

Another `EJBContext` method you should know about is `getRollbackOnly()`. The method returns a `boolean` telling you if the underlying CMT transaction has already been marked for roll back. If you suspect that this method is used very infrequently, you are right. There is one case in particular when it is very useful to check the status of the transaction you are participating in – before engaging in a very long, resource intensive operation. After all, why expend all that effort for something that is already going to be rolled back? For example, let us assume that ActionBazaar checks a potential Power Seller's creditworthiness before approving an account. Since this calculation involves a large set of data collection and business intelligence algorithms that potentially involve third parties, it is only undertaken if the current transaction has not already been rolled back. The code could look like the following:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
@Resource
private SessionContext context;
... checkCreditWorthiness(Seller seller) { ...
    if (!context.getRollbackOnly()) {
        DataSet data = getDataFromCreditBureauRobberBarons(seller);
        runLongAndConvolutedBusinessAnalysis(seller, data);
        ...
    } ...
```

If the model of catching exceptions just to call the `setRollbackOnly` method seems a little cumbersome, you are in luck. EJB 3.0 makes the job of translating exceptions into transaction rollback almost transparent using the `ApplicationException` paradigm. We will take a look at the role of exception handling in transaction management next.

6.2.5 Transaction and Exception Handling

The subject of transactions and exception handling in EJB 3.0 is intricate and often confusing. However, properly used, exceptions used to manage transactions can be extremely elegant and intuitive.

To see how how exceptions and transaction work together, let us revisit the exception handling code in the `placeSnagItOrder` method:

```
try {
    // Ordering code throwing exceptions.
    if (!bidsExisting(item)){
        validateCredit(customer);
        chargeCustomer(customer, item);
        removeItemFromBidding(item);
    }
} catch (CreditValidationException cve) {
    context.setRollbackOnly();
} catch (CreditProcessingException cpe){
    context.setRollbackOnly();
} catch (DatabaseException de) {
    context.setRollbackOnly();
}
```

As we can see, the `CreditValidationException`, `CreditProcessingException` and `DatabaseException` exceptions being thrown are essentially the equivalent of the managed transaction being rolled back. To avoid this all too common mechanical code, EJB 3.0 introduces the idea of controlling transactional outcome through the `@javax.ejb.ApplicationException` annotation. The best way to see how this works is though example. Listing 6.2 re-imlements the `placeSnagItOrder` method using the `@ApplicationException` mechanism to roll back CMT transactions:

15 Listing 6.2: Using @ApplicationException to roll back CMT transactions

```
public void placeSnagItOrder(Item item, Customer customer)
    throws CreditValidationException, CreditProcessingException,     |#1
        DatabaseException {                                          |#1
    if (!bidsExisting(item)){                                        |#2
        validateCredit(customer);                                    |#2
        chargeCustomer(customer, item);                              |#2
        removeItemFromBidding(item);                                 |#2
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    }                                                                   |#2
}
...
@ApplicationException(rollback=true)                                    |#3
public class CreditValidationException extends Exception {              |#3
...
@ApplicationException(rollback=true)                                    |#3
public class CreditProcessingException extends Exception {              |#3
...
@ApplicationException(rollback=false)                                    |#4
public class DatabaseException extends RuntimeException {               |#4
...
```
(annotation) <#1 Declaring exceptions on the throws clause>
(annotation) <#2 Exceptions thrown from the method body>
(annotation) <#3 ApplicationException Specification>
(annotation) <#4 A RuntimeException is marked as ApplicationException>

The first change from listing 6.1 you will notice is the fact that the `try-catch` blocks have disappeared and have been replaced by a `throws` clause in the method declaration#1. However it's a good idea for you to gracefully handle the application exceptions in the client and generate appropriate error message. The various nested method invocations still throw the three exceptions listed in the `throws` clause #2. The most important thing to note, however, is the three `@ApplicationException` specifications on the custom exceptions. The `@ApplicationException` annotation identifies a Java checked or unchecked exception as an *application exception*.

> In EJB, an application exception is an exception that the client is expected to handle. When thrown, such exceptions are passed directly to the method invoker. By default, all checked exceptions except for `java.rmi.RemoteException` are assumed to be application exceptions. On the other hand, all exceptions that inherit from either `java.rmi.RemoteExceptions` or `java.lang.RuntimeException` are assumed to be *system exceptions* (as you might already know, all exceptions that inherit from `java.lang.RuntimeException` are unchecked). In EJB, system exceptions are not assumed to be expected by the client. When encountered, such exceptions are *not* passed to the client as-is but are wrapped in a `javax.ejb.EJBException` instead.

In code listing 6.2, the `@ApplicationException` annotations on `CreditValidationException` and `CreditProcessingException` does not change this default behavior since both would have been assumed to be application exceptions anyway. However, by default, `DatabaseException` #4 would have been assumed to be a system exception. Applying the `@ApplicationException` annotation to it causes it to be treated as an application exception instead.

More than the `@ApplicationException` annotation itself, the `rollback` element changes default behaviour in profound ways. By default, application exceptions do not cause an automatic CMT transaction roll back since the `rollback` element is defaulted to `false`. However, setting the element to `true` tells the container that it should roll back the transaction before the exception is passed on to the client. In code listing 6.2, this means that whenever a `CreditValidationException`, `CreditProcessingException`, or `DatabaseException` is thrown, the transcation will be rolled back and the client would receive an exception indicating the cause for failure, accomplishing exactly the same thing as the more verbose code in listing 6.1 aims to

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

do. If the container detects a system exception, such as an `ArrayIndexOutOfBounds` or `NullPointerException` that you did not guard for, it will still roll back the CMT transaction. However, in such cases the container will also assume that the Bean is in inconsistent state and will destroy the instance. Because unnecessarily destroying Bean instances is costly, you should never delibretely use system exceptions.

Although the simplified code is very tempting, we recommend you to use application exceptions for CMT rollback carefully. Using the `setRollbackOnly` method, however verbose, removes the guesswork from automated transaction management, especially for junior developers that might have a hard time understanding the intricacies of exception handling in EJB. However, you should not interpret this to mean you should avoid using custom application exceptions in general. In fact, we encourage the usage of this very powerful and intuitive errror handling mechanism widely used in the Java realm.

As we can clearly see, CMT relieves us from all but the most unavoidable details of EJB transaction management. However, for certain circumstances, CMT may not give us the level of control we need. BMT gives us this additional control while still providing a very powerful, high-level API, as we will see next.

Session Synchronization

Although by using CMT you do not have full control over when a transaction is started, committed or rolled back, you can be notified about the transaction's life cycle events. This is done simply by having your CMT Bean implement the `javax.ejb.SessionSynchronization` interface. This interface defines three methods:

`void afterBegin()`: Called right after the container creates a new transaction and before the business method is invoked.
`void beforeCompletion()`: Invoked after a business method returns but right before the container ends a transaction.
`void afterCompletion(boolean committed)`: Called after the transaction finishes. The boolean `committed` flag indicates if a method was committed or rolled back.

Implementing this interface in a Stateful Session Bean can be considered very close to having a poor man's persistence mechanism, because data can be loaded into the bean when the transaction starts and unloaded right before the transaction finishes, while the `afterCompletion` callback can be used to reset default values. However, one can make a valid argument that since Session Beans are supposed to model processes, if it makes sense to cache some data and synchronize with the database as a natural part of a process, then this practice is just fine if not fairly elegant.

Note this facility does not make much sense in a stateless session bean or MDB where data should not be cached anyway; hence, the interface is not supported for those bean types.

6.3 Bean-Managed Transactions (BMT)

The greatest strength of CMT is also its greatest weakness. Using CMT, you are limited to having the transaction boundaries set at the beginning and end of business methods and rely on the container to

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

actually determine when a transaction starts, commits or rolls back. BMT on the other hand, allows you to specify exactly these details programmatically, using semantics very similar to the JDBC transaction model with which you might already be familiar. However, even in this case, the container still helps you out by actually creating the physical transaction as well as taking care of a few low level details. In BMT, you must be much more aware of the underlying JTA transaction API, primarily the `javax.transaction.UserTransaction` interface, which we will introduce shortly. But first, we will re-develop the Snag-It ordering code in BMT so that we can use it as a crutch for the next few sections. You will learn more about the `javax.transaction.UserTransaction` interface and its use. Finally we discuss pro and cons of using BMT over CMT.

6.3.1 Snag-It Ordering Using BMT

Listing 6.3 re-implements the code of Listing 6.1 using BMT. To summarize, it checks if there are any bids on the item ordered, validates the user's credit card, charges the customer and removes the item from bidding. Note that the import statements were are omitted and error handling trivialized to keep the code sample short.

16 Listing 6.3: Implementing Snag-It Using BMT

```
@Stateless)
@TransactionManagement(TransactionManagementType.BEAN)
|#1
public class OrderManagerBean {
    @Resource                                                      |#2
    private UserTransaction userTransaction;                       |#2

    public void placeSnagItOrder(Item item, Customer customer){
        try {
            userTransaction.begin();                               |#3
            if (!bidsExisting(item)){
                validateCredit(customer);
                chargeCustomer(customer, item);
                removeItemFromBidding(item);
            }
            userTransaction.commit();                              |#4
        } catch (CreditValidationException cve) {                  |#5
            userTransaction.rollback();                            |#5
        } catch (CreditProcessingException cpe){                   |#5
            userTransaction.rollback();                            |#5
        } catch (DatabaseException de) {                           |#5
            userTransaction.rollback();                            |#5
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(annotation) <#1 Bean Uses BMT>
(annotation) <#2 Injected Transaction>
(annotation) <#3 Transaction Started>
(annotation) <#4 Transaction Commited>
(annotation) <#5 Transaction Rolled Back on Exception>

Briefly scanning the code, you will note that the `@TransactionManagement` annotation specifies the value TransactionManagementType.`BEAN` as opposed to

Licensed to John Sweitzer <admin@saolailaem.info>

TransactionManagementType.CONTAINER, indicating that we are using BMT this time#1. The TransactionAttribute annotation is missing altogether since it is applicable only for CMT. A UserTransaction, the JTA representation of a bean managed transaction is injected#2 and used explicitly to begin#3, commit#4 or rollback#5 a transaction. The transaction boundary is much smaller than the entire method and includes only calls that really need to be atomic. The sections that follow discuss the code in greater detail next, starting with getting a reference to the javax.transaction.UserTransaction.

6.3.2 Getting a UserTransaction

The UserTransaction interface encapsulates the basic functionality provided by a JAVA EE transaction manager. JTA has a few other interfaces used under different circumstances. We will not cover them, as the great majority of time you will be dealing with UserTransaction. For a full coverage of JTA, check out http://java.sun.com/products/jta/. As you might expect, the UserTransaction interface is too intricate under the hood to be instantiated directly and must be obtained from the container. In Listing 6.3, we have used the simplest way of getting a UserTransaction, namely by injecting it through the @Resource annotation. There are a couple of other ways to do this instead: Using JNDI Lookup or the EJBContext.

JNDI Lookup

The application server binds the UserTransaction to the JNDI name java:comp/UserTransaction. We can look it up directly using JNDI with this code:

```
Context context = new InitialContext();
UserTransaction userTransaction =
  (UserTransaction) context.lookup("java:comp/UserTransaction");
userTransaction.begin();
// Perform transacted tasks.
userTransaction.commit();
```

This method is typically used outside of EJBs — for example, if you need to use a transaction in a helper or a non-managed class in the EJB or web-tier where dependency injection is not supported. If you actually find yourself in this situation, you might want to think long and hard about moving the transactional code to an EJB where you have access to greater abstractions.

EJBContext

We can also get a UserTransaction by invoking the getUserTransaction method of the EJBContext. This is useful if we are using a SessionContext or MessageDrivenContext for some other purpose anyway and a separate injection just to get a transaction instance would be redundant. A very important thing to note here is that we can only use the getUserTransaction method if we are using bean managed transactions. Calling this in a CMT environment will cause the context to throw an IllegalStateException. The following code shows the getUserTransaction method in action:

```
@Resource
private SessionContext context;
...
UserTransaction userTransaction = context.getUserTransaction();
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
userTransaction.begin();
// Perform transacted tasks.
userTransaction.commit();
```

On a related but very relevant note, you cannot use the EJBContext getRollbackOnly and setRollbackOnly methods in BMT and they will throw an IllegalStateException if accessed. Next, let us take a look at how the obtained UserTransaction interface is actually used.

6.3.3 Using UserTransaction

We have already seen the UserTransaction interface's most frequently used methods, namely begin, commit and rollback. The UserTransaction interface has a few other useful methods we should take a look at as well. The definition of the entire interface looks like the following:

```
public interface UserTransaction {
void begin() throws NotSupportedException, SystemException;
void commit() throws RollbackException,
HeuristicMixedException, HeuristicRollbackException, SecurityException,
IllegalStateException, SystemException;

void rollback() throws IllegalStateException, SecurityException,
SystemException;

void setRollbackOnly() throws IllegalStateException,
SystemException;

int getStatus() throws SystemException;
void setTransactionTimeout(int seconds) throws SystemException;
}
```

The begin method creates a new low-level transaction behind the scenes and associates it with the current thread. You might be wondering what would happen if you called the begin method twice before calling rollback or commit. You might think this is possible if you want to create a nested transaction, a paradigm supported by some transactional systems. In reality, the second invocation of begin would throw a NotSupportedException since Java EE does not support nested transactions. The commit and rollback methods, on the other hand, remove the transaction attached to the current thread by begin. While commit sends a 'success' signal to the underlying transaction manager, rollback abandons the current transaction. The setRollbackOnly method on this interface might be slightly counterintuitive as well. After all, why bother marking a transaction rolled back when you can roll it back yourself?

To understand why, consider the fact that we could call a CMT method from our BMT bean that contains a lengthy calculation and checks the transactional flag before proceeding (like out Power Seller credit validation example in section 6.3.4). Since our BMT transaction would be propagated to the CMT method, it might be programmatically simpler, especially in a long method, to mark the transaction rolled back using the setRollbackOnly method instead of writing an involved if-else block avoiding such conditions. The getStatus method is a more robust version of getRollbackOnly in the CMT world. Instead of returning a boolean, this method will return an integer-based status of the current transactions indicating a more fine-tuned set of states a transaction could possibly be in. The javax.transaction.Status interface defines exactly what these states are and we list them in Table 6.3.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

1.13 Table 6.3: The possible values of the javax.transaction.Status interface. These are the status values returned by the UserTransaction.getStatus method.

Status	Description
STATUS_ACTIVE	The transaction is associated and is in an active state.
STATUS_MARKED_ROLLBACK	The associated transaction is marked for rollback possibly due to invocation of the setRollbackOnly method.
STATUS_PREPARED	The associated transaction is in the prepared state because all resources have agreed to commit (refer to the two-phase commit discussion in section 6.1.2).
STATUS_COMMITTED	The associated transaction has been committed.
STATUS_ROLLEDBACK	The associated transaction has been rolled back.
STATUS_UNKNOWN	The status for associated transaction is not known (very clever, don't you agree?).
STATUS_NO_TRANSACTION	There is no associated transaction in the current thread.
STATUS_PREPARING	The associated transaction is preparing to be committed and awaiting response from subordinate resources (refer to the two-phase commit discussion in section 6.1.2).
STATUS_COMMITTING	The transaction is in the process of committing
STATUS_ROLLING_BACK	The transaction is in the process of rolling back

1.14

The `setTransactionTimeout` method sets the time in milliseconds, in which a transaction must finish. The default transaction timeout value is set to different values for different application servers. For example, JBoss has a default transaction timeout value of 300 seconds whereas Oracle Application Server 10g has a default transaction timeout value of 30 seconds. You might want to use this method if you are using a very long-running transaction. Typically, it is better to simply set the application server-wide defaults using vendor specific interfaces, however. At this point, you are probably wondering how to set a transaction timeout when using CMT instead. This is only supported by containers either using an attribute in the vendor specific deployment descriptor or vendor specific annotations.

Comparing Listing 6.1 and 6.3, you might ask if the additional complexity and verbosity associated with BMT is really worth it. Let us explore this issue in detail next.

6.3.4 The Pros and Cons of BMT

CMT is the default transaction type for EJB transactions. In general, BMT should be used sparingly because it is verbose, complex and difficult to maintain. There are some concrete reasons to use BMT, however. BMT transactions need not begin and end in the confines of a single method call. If you are using a Stateful Session Bean and need to maintain a transaction across method calls, BMT is your only option. Be warned, however, this technique is very complicated and error-prone and you might just be better off rewriting your application rather than attempting this. Can you spot a bug in listing 6.4? The last catch-block did not rollback the transaction like all other catch-blocks did. But even that is not enough, what if the code throws an Error (rather than an Exception)? Whichever way you do it, it is error prone and we recommend using CMT instead.

Another argument for BMT is that you can fine-tune your transaction boundaries so that the data held by your code is isolated for the shortest time possible. Our opinion is that this idea indulges in premature optimization and again, you are probably better off refactoring your methods to be smaller and more specific anyway. Another drawback for BMT is the fact that it can never join an existing transaction. Existing transactions are always suspended when calling a BMT method, significantly limiting flexible component reuse.

Believe it or not, this wraps up our discussion of EJB transaction management. It is now time to turn our attention to another critical aspect of enterprise Java development -- security.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

6.4 Exploring EJB Security

Securing enterprise data has always has been a primary application development concern. This is even more true today in the age of sophisticated cyber-world hackers, phishers and identity/data thieves. Consequently, security is a major concern in developing robust Java EE solutions. EJB has a security model that is elegant, flexible, and portable across heterogonous systems.

In the remainder of this Chapter, we explore some basic security concepts such as authentication and authorization, users and groups and investigate discuss the Java EE/EJB security framework and take a look at both declarative and programmatic security in EJB 3.0.

We start with two of the most basic ideas in security: authentication and authorization.

6.4.1 Authentication versus Authorization

Securing an application involves two primary functions: authentication and authorization. Authentication must be done before authorization can be performed, but as we shall see, both are necessary aspects of application security. Let us explore both of these concepts next.

Authentication

Authentication is the process of verifying user identity. By authenticating yourself, you prove that you are who you say you are. In the real world, this is usually accomplished through visual inspection/identity cards, signature/handwriting, fingerprint checks and even DNA tests. In the computer world, the most common method of authentication is by checking username and password. All security is meaningless if someone can log onto a system with a false identity.

Authorization

Authorization is the process of determining whether a particular user has access to a particular resource or task, and it comes into play once a user is authenticated. In an open system, an authenticated user can access any resource. In a realistic security environment, this all-or-nothing approach would be highly ineffective. Therefore, most systems must restrict access to resources based on user identity. Although there might be some resources in a system that are accessible to all, most resources should be accessed only by a limited group of people.

Both authentication and authorization, but particularly authorization, is closely tied to other security concepts, namely *users*, *groups* and *roles*, which we will look at next.

6.4.2 Users, Groups and Roles

To perform efficient and maintainable authorization, it is best if we can organize users into some kind of grouping. Otherwise, each resource must have an associated list of all the users that can access it. In a non-trivial system, this would easily become an administrator's nightmare. To avoid this problem, users are organized into *groups* and groups as a whole are assigned access to resources, making the access list for an individual resource much more manageable.

The concept of *role* is a closely related to the concept *group* but is a bit tricky to understand. For an EJB application, the concept of roles is much more critical than users and groups. To understand the distinction, consider the fact that you might not be building an in-house solution but a packaged Java EE application. As a result, you might not know the exact operational environment your

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

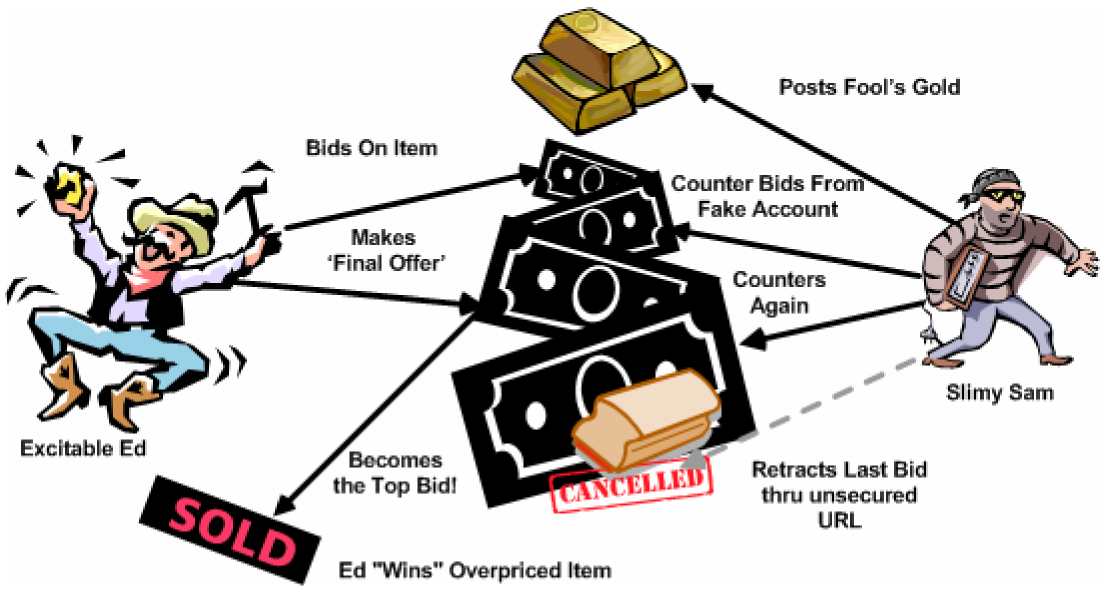
Licensed to John Sweitzer <admin@saolailaem.info>

application might be deployed in once it is purchased by the customer. As a result, it would be impossible for you to code for the specific group names a customer's system administrator will choose. Neither should you care about groups. What you do care about is what *role* a particular user in a group plays for your application. In the customer system, user Joe might belong to the system group called *peons*. Now assume that an ActionBazaar Integrated B2B Enterprise Purchasing System installation is made on the customer's site. Among other things, this type of B2B installation transparently logs in all existing users from the customer system into the ActionBazaar site through a custom desktop shortcut. Once logged in, from ActionBazaar's perspective, Joe could simply be a *buyer* who buys items online on behalf of the B2B customer company. To another small application in the operational environment, user Joe might be an *administrator* who changes system-wide settings. For each deployed application in the operational environment, it is the responsibility of the system administrator to determine what *system group* should be mapped to what *application role*. In the Java EE world, this is typically done through vendor-specific administrative interfaces. As a developer, you simply need to define what roles your application's users have and leave the rest to the administrator. For ActionBazaar, roles can be buyers, sellers, administrators and so on.

Let us solidify our understanding of application security in EJB using an ActionBazaar example next.

6.5.3 A Security Problem in ActionBazaar

At ActionBazaar, customer service representatives (CSR) are allowed to cancel a user's bid under certain circumstances (for example, if the seller discloses something in answer to an email question from the bidder that should have been mentioned on the item description). However, the cancel bid operation does not check if the user is actually a CSR, as long as the user can locate the functionality on the ActionBazaar site, for example by typing in the correct URL.



43 **Figure 6.4: Security breach into ActionBazaar allows hacker to shill bids by posting an item, starting a bidding war from a fake account and then at the last minute canceling the highest fake bid. The end result is that an unsuspecting bidder winds up with an overpriced item.**

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

A clever hacker broke into the ActionBazaar web server logs and figured out the URL used by CSRs to cancel bids. Using this knowledge, he devised an even cleverer “shill bidding” scheme to incite users to overpay for otherwise cheap items. The hacker would post items on sale and use a friend’s account to incite a bidding war with genuine bidders. If at any point genuine bidders gave up bidding and a fake bid becomes the highest bid, the hacker would avoid actually having to pay for the item and losing money in posting fees by canceling his highest fake bid through the stolen URL. No one would be any wiser as the genuine bidders as well as the ActionBazaar system would think the highest bid was canceled for legitimate reasons. The end result would be that an honest bidder would be fooled into overpaying for otherwise cheap items. After a while, ActionBazaar customer service finally catches onto the scheme thanks to a few observant users and makes sure the bid canceling action is authorized for CSRs only. Now if a hacker tries to access the functionality, the system would simply deny access, even if the hacker has a registered ActionBazaar account and accesses the functionality through the URL or otherwise. As we discuss how security is managed by EJB in the next section, you will get a very good idea of what the solution might actually look like.

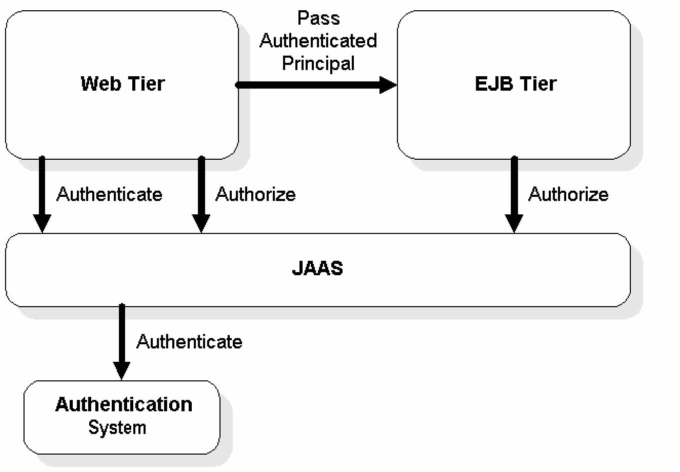
6.4.4 EJB 3.0 and Java EE Security

Java EE security is largely based on the JAAS (Java Authentication and Authorization Service) API. JAAS essentially separates the authentication system from the Java EE application by using a well-defined, pluggable API. In other words, the Java EE application need only know how to talk to the JAAS API. The JAAS API, on the other hand, knows how to talk to underlying authentication systems like LDAP or Microsoft Active Directory using a vendor plug-in. As a result, you can easily swap between authentication systems simply by swapping JAAS plug-ins without changing any code. In addition to authentication, the application server internally uses JAAS to perform authorization for both the web and EJB tiers. When we look at programmatic EJB security management, we will directly deal with JAAS very briefly, namely when we discuss the JAAS `javax.security.Principal` interface. Feel free to explore JAAS at http://java.sun.com/products/jaas/ since our discussion is limited to what is needed for understanding EJB security.

JAAS is designed so that both the authentication and authorization steps can be performed at any Java EE tier, including the Web and EJB tiers. Realistically, however, most Java EE applications are web accessible and share an authentication system across tiers, if not across the application server. JAAS fully leverages this reality and once a user (or entity, to use a fancy security term) is authenticated at any Java EE tier, the authentication context is passed through tiers whenever possible instead of repeating the authentication step. The `Principal` object we already mentioned represents this sharable, validated authentication context. Figure 6.5 depicts this common Java EE security management scenario.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



44 **Figure 6.5: Most Common Java EE Security Management Scenario using JAAS**

As shown in Figure 6.5, a user enters the application through the Web tier. The web tier gathers authentication information from the user and authenticates the supplied credentials using JAAS against an underlying security system. A successful authentication results in a valid user Principal. At this point, the Principal is associated with one or more roles. For each secured Web/EJB tier resource, the application server checks if the principal/role is authorized to access the resource. The Principal is transparently passed from the Web tier to the EJB tier as needed.

A detailed discussion of Web tier authentication and authorization is beyond the scope of this book as is the extremely rare scenario of standalone EJB authentication using JAAS. However, we will give you a basic outline of Web tier security to serve as a starting point for further investigation before diving into authorization management in EJB 3.0.

Web-Tier Authentication and Authorization

The Web-tier Servlet specification (http://java.sun.com/products/servlet/) successfully hides a great many low-level details for both authentication and authorization. As a developer, you simply need to tell the Servlet container what resources you want secured, how they are secured, how authentication credentials are gathered and what roles have access to secured resources. The Servlet container, for the most part, takes care of the rest. Web-tier security is mainly configured using the `login-config` and `security-constraint` elements of the `web.xml` file. Let us take a look at an example of how securing the administrative module of ActionBazaar might look using these elements.

17 Listing 6.3: Sample web.xml elements to secure order cancelling and other ActionBazaar admin functionality.

```
<login-config>
    <auth-method>BASIC</auth-method>                                    |#1
    <realm-name>ActionBazaarRealm</realm-name>                          |#2
</login-config>

...
<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            ActionBazaar Administrative Component
        </web-resource-name>
        <url-pattern>/admin/*</url-pattern>                             |#3
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    </web-resource-collection>
    <auth-constraint>
        <role-name>CSR</role-name>                                        |#4
    </auth-constraint>
</security-constraint>
```
(annotation) <#1 Indicates How Authentication is Done>
(annotation) <#2 Specifies Authentictaion Realm>
(annotation) <#3 Indicates what is Secured>
(annotation) <#4 The Roles Accessing the Secured Resource>

#1 in Listing 6.3 specifies how the web container should gather and validate authentication. In our case, we have chosen the simplest authentication mechanism, `BASIC`. `BASIC` authentication uses a HTTP-header based authentication scheme that usually causes the web-browser to gather username/password information using a built-in prompt. Other popular authentication mechanisms include `FORM` and `CLIENT-CERT`. `FORM` is essentially the same as `BASIC` except for the fact that the prompt used is an HTML form that you create. `CLIENT-CERT`, on the other hand, is an advanced form of authentication that bypasses username/password prompts altogether. In this scheme, the client sends a public key certificate stored in the client browser to the web server using SSL and the server authenticates the contents of the certificate instead of a username/password. The credentials are then validated by JAAS. #2 specifies the realm the container should authenticate against. A *realm* is essentially a container-specific abstraction over a JAAS-driven authentication system. #3 specifies that all URLs that match the pattern `/admin/*` should be secured. Finally, #4 specifies that only validated principals with the CSR role can access the secured pages. In general, this is all there really is to securing a web application using JAAS, unless you choose to use programmatic security, which essentially follows the same pattern used in programmatic EJB security that we will discuss soon.

EJB Authentication and Authorization

At the time of writing, authenticating and accessing EJBs from a standalone client, without any help from the Servlet container is still a daunting task that requires you to thoroughly understand JAAS. In affect, we would have to implement all of the authentication steps that the Servlet container nicely abstracts away from us. Thankfully, this task is not undertaken very often and most application servers provide JAAS login-module available that can be used by applications.

On the other hand, the authorization model in EJB 3.0 is very simple yet powerful. Much like authorization in the Web tier, it centers on the idea of checking whether the authenticated Principal is allowed to access an EJB resource based on the Principal's role. Like transaction management, authentication can be either declarative or programmatic, each of which provides a different level of control over the authentication process. In addition, like the transaction management features discussed in this chapter, security really applies to Session Beans and Message Driven Beans and not the EJB Java Persistence API.

We will first explore declarative security management by coding up our bid-canceling scenario presented in 6.4.3 and then move on to exploring programmatic security management.

6.4.5 Declarative Security

Listing 6.4 applies authentication rules to the `BidManagerBean` that includes the `cancelBid` method our clever hacker used for his shill-bidding scheme. Now, only CSRs are allowed to use this method. Note, we have omitted method implementation since thus really is not relevant to our discussion.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

18 Listing 6.4: Securing Bid Cancellation Using Declarative Security Management

```
@DeclareRoles("BIDDER", "CSR", "ADMIN")                                    |#1
@Stateless
public class BidManagerBean implements BidManager {
    @RolesAllowed("CSR, ADMIN")                                             |#2
    public void cancelBid(Bid bid, Item item) {...}

    @PermitAll                                                              |#3
    public List<Bid> getBids(Item item) {...}
}
```

(annotation) <#1 Roles for the bean declared>
(annotation) <#2 Roles with access to method>
(annotation) <#3 Permits all system roles access to the method>

The listing uses some of the most commonly used security annotations defined by Common Metadata annotations for Java Platform specification (JSR-250), `javax.annotation.security.DeclareRoles`, `javax.annotation.security.RolesAllowed` and `javax.annotation.security.PermitAll`. Two other annotations that we have not used but will discuss are the `javax.annotation.security.DenyAll` and `javax.annotation.security.RunAs`. We will start our analysis of the code and security annotations with `@DeclareRoles` annotation.

Declaring Roles

It is highly recommended, but not required that you declare the security roles to be used in your application, EJB module, EJB or business methods. There are a few ways of declaring roles, one of which is through the `@DeclareRoles` annotation, which we use in Listing 6.4#1. This annotation applies at either the method or the Class level and consists of an array of role names. We are specifying that the `BidManagerBean` uses the roles of Bidders, CSRs and System Administrators. Alternatively, we can specify roles for the entire enterprise application or EJB module through deployment descriptors. The ActionBazaar application could use the roles of Guests, Bidders, Sellers, Power Sellers, CSRs, admins, and so on. If we never declare roles, the container will automatically build a list of roles by inspecting the `@RolesAllowed` annotation that we will talk about next. Remember, when the application is deployed, the local system administrator must map each role to groups defined in the runtime security environment.

Specifying Authenticated Roles

The `@RolesAllowed` annotation is the crux of declarative security management. This annotation can be applied to either an EJB business method or an entire class. When applied to an entire EJB, it tells the container which roles are allowed to access any EJB method. On the other hand, we can use this annotation on a method to specify the authentication list for that particular method. The tremendous flexibility offered by this annotation becomes evident when we consider the fact that we can override class level settings by reapplying the annotation at the method level (for example to restrict access further for certain methods). However, we discourage such usage because at best it is convoluted and at worst it can cause subtle mistakes that are hard to discern. In Listing 6.4, we specify that only CSR and system administrator roles be allowed to cancel bids through the `cancelBid` method#2. The `@PermitAll` and `@DenyAll` annotations are conveniences that perform essentially the same function as this annotation. We will discuss both next.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

@PermitAll and @DenyAll

We can use the `@PermitAll` annotation to mark an EJB class or a method to be invoked by any role. We use this annotation in listing 6.4#3 to instruct the container that any user can retrieve the current bids for a given item. You should use this annotation sparingly, especially at the Class level, as it is possible to inadvertently leave security holes if it is used carelessly. The `@DenyAll` annotation does exactly the opposite of what `@PermitAll` does. That is, when used at either the Class or the method level, it renders functionality inaccessible by any role. You might be wondering why you would ever use this annotation. The annotation makes sense when you consider the fact that your application may be deployed in wide-ranging environments that you did not envision. You can essentially invalidate methods or classes that might be inappropriate for a particular environment without changing code using the `@DenyAll` annotation. Just as with the `@RolesAllowed` annotation, when applied at the method level, these annotations will override bean-level authorization settings. We will now wrap up our discussion of declarative security management by discussing our final annotation, `@RunAs`.

@RunAs

The `@RunAs` annotation comes in handy if we need to dynamically assign a new role to the existing Principal in the scope of an EJB method invocation. We might need to do this, for example if we are invoking another EJB within our method but the other EJB requires a role that is different from the current Principal's role. Depending on the situation, the new "assumed" role might be either more restrictive, lax or neither. For example, the `cancelBid` method in Listing 6.4 might need to invoke a statistics tracking EJB that manages historical records in order to delete the statistical record of the cancelled bid ever taking place. However, the method for deleting a historical record might require an `ADMIN` role. Using the `@RunAs` annotation, we can temporarily assign a `CSR` an `ADMIN` role so that the statistics tracking EJB thinks an admin is invoking the method:

```
@RunAS("ADMIN")
@RolesAllowed("CSR")
public void cancelBid(Bid bid, Item item) {...}
```

You should use this annotation sparingly since like the `@PermitAll` annotation, it can open up security holes you might not have foreseen.

As you can see, declarative security gives you access to a powerful authentication framework while staying mostly out of the way. The flexibility available to you through the relatively small number of relevant annotations should be apparent as well. If you have ever rolled out your own security or authentication system, however, one weakness might have crossed your mind already. The problem is that although you can authenticate a role using declarative security, what if you need to provide security settings specific to individuals, or even simple changes in method behavior based on the current Principal's role? This is where programmatic EJB security steps onto the stage.

6.4.6 Using EJB Programmatic Security

In effect, programmatic security gives us direct access to the Principal as well as convenient means to check the Principal's role in code. Both of these functions are made available through the EJB context. We will begin exploring programmatic security by redeveloping the bid-canceling scenario as a starting point. Listing 6.5 implements the scenario:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

19 Listing 6.5: Securing Bid Cancellation Using Programmatic Security

```
@Stateless
public class BidManagerBean implements BidManager {
    @Resource SessionContext context;                                   |#1
    ...
    public void cancelBid(Bid bid, Item item) {
        if (!context.isCallerInRole("CSR")) {                           |#2
           throw new SecurityException(                                 |#3
               "No permissions to cancel bid");                         |#3
        }
        ...
    }
    ...
}
```

(annotation) <#1 EJB Context Injected>
(annotation) <#2 Authentiation Check>
(annotation) <#3 Exception Thrown on Violation>

#1 in listing 6.5 injects the EJB context. We use the `isCallerInRole` method of the EJBContext to see if the underlying authenticated principal has the CSR role#2. If it does not, we throw a `java.lang.SecurityException` notifying the user about the authentication violation#3. Otherwise, the bid cancellation method is allowed to proceed normally. We will discuss both the security management related methods provided in the EJB context next, namely the `isCallerInRole` and `getCallerPrincipal`.

isCallerInRole and getCallerPrincipal

Believe it or not, programmatic security is made up solely of the two previously mentioned JAAS-related methods. The methods are defined in the `javax.ejb.EJBContext` interface as follows:

```
public interface EJBContext {
    ...
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String roleName);
    ...
}
```

We already saw the `isCallerInRole` method in action; it is fairly self-explanatory. Behind the scenes, the EJB context retrieves the Principal associated with the current thread and checks if any of its roles matches the name you provided. The `getCallerPrincipal` method gives us direct access to the `java.security.Principal` representing the current authentication context. The only method of interest in the `Principal` interface is `getName()`, which returns the name of the Principal. Most of the time, the name of the Principal is the login name of the validated user. This means that just as in the case of a homemade security framework, you could validate the individual user if you needed to. For example, let us assume that we had a change of heart and decided that in addition to the CSRs, bidders can cancel their own bids as long as the cancellation is done within a minute of putting in the bid. We could implement this using the `getCallerPrincipal` method as follows:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
public void cancelBid(Bid bid, Item item) {
    if (!context.isCallerInRole("CSR")
        && !(context.getCallerPrincipal().getName().equals(
            bid.getBidder().getUsername()) && (bid.getTimestamp() >=
                (getCurrentTime() - 60*1000))))) {
        throw new SecurityException(
            "No permissions to cancel bid");
    }
    ...
}
```

Note though, there is no guarantee exactly what the Principal name might return. In some environments, it can return the role name, group name or any other arbitrary String that makes sense for the authentication system. Before you use the `Principal.getName()` method, you should check the documentation of your particular security environment. As you can see, the one great drawback of programmatic security management is the intermixing of security code with business logic as well as the potential hard-coding of role and Principal names. In previous versions of EJB, there was no real way of getting around these shortfalls. However, in EJB 3.0, you can alleviate this problem somewhat using interceptors. Let us see how to accomplish this next.

Using Interceptors for Programmatic Security

As we know, in EJB 3.0 we can setup interceptors that are invoked before and after (around) any EJB business method. This facility is ideal for cross-cutting concerns that should not be duplicated in every method, such as programmatic security (if you are unfamiliar with the term crosscutting, in aspect-oriented-programming or AOP-speak, application requirements that cut across components are called *cross-cutting concerns*). We could re-implement code Listing 6.5 using interceptors instead of hard-coding security in the business method as follows:

20 Listing 6.6: Using Interceptors with Programmatic Security

```
public class SecurityInterceptor {
    @AroundInvoke                                                  |#1
    public Object checkUserRole(InvocationContext context)
        throws Exception {
        if (!context.getEJBContext().isCallerInRole("CSR")) {      |#2
            throw new SecurityException(
                "No permissions to cancel bid");
        }

        return context.proceed();
    }
}

@Stateless
public class BidManagerBean implements BidManager {
    @Interceptors(actionbazaar.security.SecurityInterceptor.class)  |#3
    public void cancelBid(Bid bid, Item item) { ... }
```

(annotation) <#1 Marked for Intercepted Invocation>
(annotation) <#2 EJB Context Accessed from Invocation Context>
(annotation) <#3 Specifying Interceptor for method>

Licensed to John Sweitzer <admin@saolailaem.info>

The `SecurityInterceptor` Class method `checkUserRole` is designated as `AroundInvoke` meaning it would be invoked whenever a method is intercepted#1. In the method, we check to see if the Principal is a CSR#2. If case the role is not right, we throw a `SecurityException`. Our `BidManagerBean`, on the other hand, specifies the `SecurityInterceptor` Class as the interceptor for the `cancelBid` method. Note, although using interceptors helps matters a little bit in terms of removing hard-coding out of business logic, there is no escaping the fact that there is still a lot of hard-coding going on in the interceptors themselves. Moreover, unless you are using a very simple security scheme where most EJB methods have similar authorization rules and you can reuse a small number of interceptors across the application, things could get complicated fast. In affect, you would have to resort to writing ad-hoc interceptors for method-specific authentication combinations (just admin, CSR and admin, everyone, no one, and so on). Contrast this to the relatively simple life of using the declarative security management annotations or deployment descriptors. All in all, declarative security management is the scheme you should stick with, unless you have an absolutely unavoidable reason not to do so.

6.5 Summary

In this Chapter, we discussed the basic theory of transactions, transaction management using CMT and BMT, basic security concepts as well as programmatic and declarative security management. Both transactions and security are crosscutting concerns that ideally should not be interleaved with business logic. The EJB 3.0 take on security and transaction management tries to reflect exactly this belief, fairly successfully in our opinion, while allowing some flexibility.

An important thing to notice is the fact that even if you specify nothing for transaction management in your EJB, the container still assumes default transactional behavior. On the other hand, the container applies no default security settings if you leave it out. The assumption is that at a minimum, an application server would be authenticated and authorized at a level higher than EJB (for example the Web tier). Nevertheless, we highly recommend that you not leave yourself vulnerable by ignoring security at the mission-critical EJB layer where most of your code and data is likely to reside. Security vulnerabilities are insidious and you are better safe than sorry. Most importantly, the security features of EJB 3.0 are so easy to use that there is no reason to risk the worst by ignoring them.

The discussion on security and transactions wrap up our coverage of Session and Message Driven Beans. Neither feature is directly applied to the EJB Persistence API as they were for Entity Beans in EJB 2.1. We will see why this is the case as we explore the Persistence API in the next few chapters.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 7 Implementing Domain Models with EJB 3.0

Most of today's enterprise systems save their data into a relational database of some kind. This is why, persistence, the process of saving and retrieving data from permanent storage, has been a major application development concern for many decades. As a matter of fact, some authoritative sources claim that a great majority of enterprise development efforts concentrate on the problem of persistence.

Arguably, after JDBC, EJB 2.x Entity Beans has been the most significant groundbreaking solution to the problem of persistence in Java. Unfortunately, many of us who developed Entity Beans experienced an API that felt overcomplicated, cumbersome and unpolished. It is pretty fair to say Entity Beans were the most weakly conceived part of EJB 2.x. In the past few years, lightweight persistence solutions like Hibernate and TopLink successfully filled the gap left open by Entity Beans. EJB 3.0 Java Persistence API (JPA) brings the innovative ideas created by these popular solutions into the Java EE standard and leaves behind the Entity Beans paradigm.

Domain modeling is a concept inseparably linked with persistence. In fact, it is often the domain model that is persisted. As a result, it makes good sense to present JPA by breaking things down into four Chapters that might mirror the iterative process of developing the domain model and persistence layer of the ActionBazaar application. We have decided on four convenient development phases: defining, persisting, manipulating and querying the domain model. In this Chapter, we briefly introduce domain modeling, present the ActionBazaar domain model, and implement part of the domain model using EJB 3.0 JPA. In Chapter 8 we explain how entities in our domain model are persisted into a database by using Object-Relational mapping. In Chapter 9, we manipulate the entities using the `EntityManager` API. Finally, in Chapter 10, we query the persisted entities using the EJB 3.0 Query API.

7.1 Domain Modeling and the JPA

Often the first step to developing an enterprise application is creating the domain model, that is, listing the entities in the domain and defining the relationships between them.

In this section we will first give you a primer on domain modeling. Then we will explore the ActionBazaar problem domain and identify actors in a domain model such object, relationships, cardinality, etc. We will provide a brief overview of how domain modeling is supported with EJB 3.0 Java Persistence API and then build a simple domain object as a simple Java class.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

7.1.1 Introducing Domain models

Although domain modeling is often presented as something complex and arcane, the idea behind it is really pretty simple. In affect, a *domain model* is a conceptual image of the problem your system is trying to solve. Very literally, it is made up of the objects in the "system universe" and the relationships or associations between them. As you can guess, an *object* in a domain model need not be a physical object but just a concept used by your system. A *relationship* on the other hand is an imaginary link between objects that "need to know about one another." The critical thing to note is that the domain model describes the objects and how the objects might relate to each other, but not how a system acts on the objects.
We like to think of a domain model as a set of interlocking toy blocks. Each uniquely shaped block in the set is an *object*. The shape of each block determines how they fit with each other. Each such "fit" is a *relationship*. In the end though, you put together the blocks into whatever configuration sparks your imagination. The master plan for putting together the final results of the potential offered by the block set is the *business rules* of the application. The business rules of the application are the implemented by the Session Beans and Message Driven Beans we discussed in the last few Chapters, while the persistence API implements the domain model the business rules act on.

We will not talk about domain modeling much further than what is needed for explaining the concepts we just introduced. However, we encourage you to explore the topic further by checking out the excellent books written on the subject of domain modeling, most notably *Patterns of Enterprise Applications Architecture* by Martin Fowler (Addison-Wesley, 2002).UML Class diagrams are the most popular method of creating the initial domain model. However, we are going to avoid using formal Class Diagrams throughout this Chapter and in the rest of the book. Instead we will use the simplest diagrams possible that might have a very shallow resemblance to UML.

7.1.2 The ActionBazaar Problem Domain

Modeling the entire ActionBazaar domain will introduce complexity that we do not really need in order to explain JPA. To avoid this unnecessary complexity, we are going to develop the core functionality of the ActionBazaar application that is directly related to buying and selling items on bid online[7].

As Figure 7.1 shows, at the heart of it, ActionBazaar centers on the following activities[8]:

30. Sellers posting an item on ActionBazaar.
31. Items being organized into searchable and navigable categories.
32. Bidders placing bids on items.
33. The highest bidder ordering the item won.

[7] Admittedly, this is a slightly unoriginal example. We considered using an example slightly tangential to the central theme of ActionBazaar but decided against it and remained true to the ActionBazaar core concept.
[8] If you are familiar with use cases and the list looks a lot like use cases, they really are.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

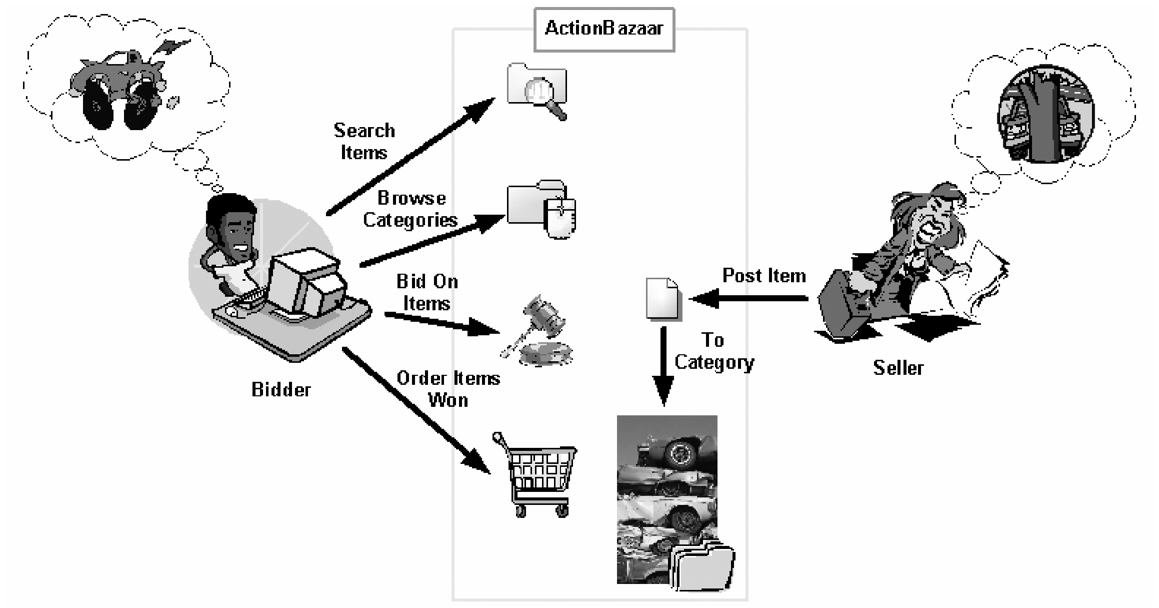


Figure 7.1: The core functionality of ActionBazaar. Sellers post items into searchable and navigable categories. Bidders bid on found items and the highest bidder orders items won.

In our artificially simplistic scenario, we can pick out the domain *objects* simply by scanning the list of activities and looking for nouns: seller, item, category, bidder, bid and order. Our goal is identify the domain objects or entities that we want to persist in the database. In the real world, finding domain objects usually involves hours of work and many iterations of analyzing the business problem. We will make our initial diagram by randomly throwing together our objects into Figure 7.2:

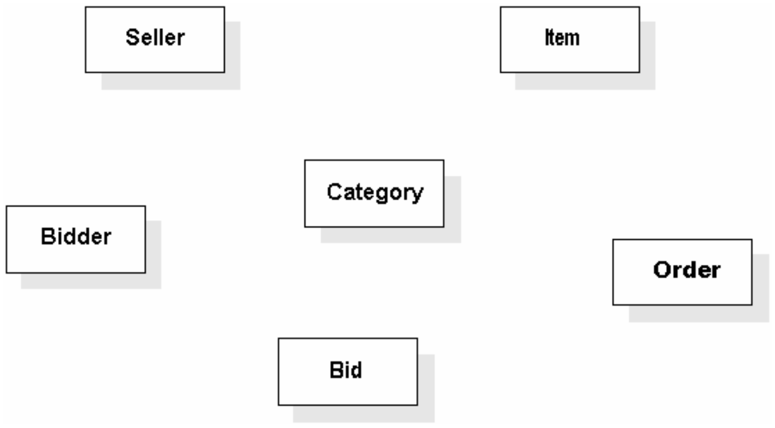


Figure 7.2 Entities are objects that can be persisted in the database. As the first step you identify entities e.g. Entities in the ActionBazaar domain

Putting in the links between objects that should know about each other (these are the infamously complex domain *relationships*) will complete our domain model. We encourage you to spend a little bit of time looking at Figure 7.2 and trying to guess how the objects might be related before peeking at the finished result in Figure 7.3.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

We will not spell out or discuss every relationship in Figure 7.3 since most are pretty intuitive even with the slightly cryptic arrows and numbers. We will explain what is going on with the arrows and numbers in just a little bit when we talk about direction and multiplicity of relationships. For now, all we really need to note is the text describing how objects are related to each other. For example, an item is sold by a seller, the item is in a category, each category has a parent category and a possible set of subcategories, a bidder places a bid on an item and so on. You should also note that although the domain model describes the possibilities for cobbling objects together, it does not actually describe the way the objects are manipulated. For example, although we can see that an order consists of one or more items and is placed by a bidder, we are not really told how or when these relationships are formed. Applying a bit of common sense though, it is easy to figure out that an item won through a winning bid is put into an order placed by the highest bidder. These relationships are probably formed by the business rules after the bidding is over and the winner checks out the item won.

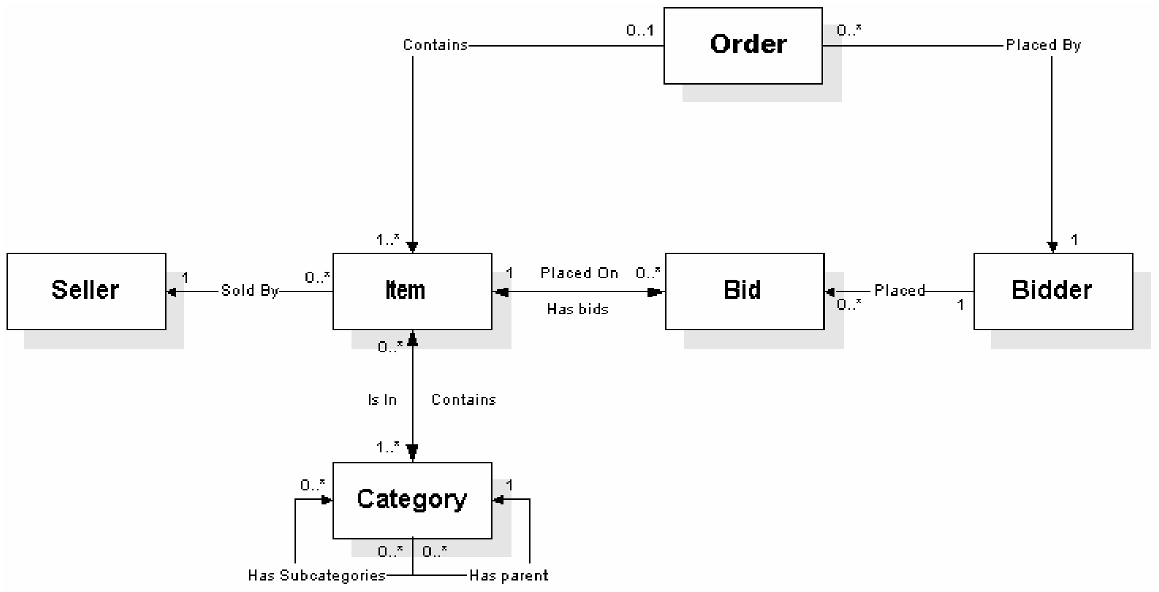


Figure 7.3: ActionBazaar domain model complete with entities and relationships. Entities are related to each other and the relation can either be one-one, one-many, many-one and many-many. Relataionships can either be uni or bi-dierctional.

We will clarify the concepts behind domain model objects, relationships and multiplicity a little more next, before moving onto the JPA.

7.1.3 Domain Model Actors

Domain modeling theory identifies four not three, domain model "actors": objects, relationships,the multiplicity and optionality of relationships. We will now discuss the details that we have left out so far on all three actors.

Objects

For a Java developer perspective, as the name implies, domain objects are very closely related to Java Objects. Like Java Objects, domain objects can have both behavior (*methods* in Java terms) and

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

state (*instance variables* in Java). For example, the category domain object probably has name, creation date and modification date as attributes. Similarly, a category probably also has the behavior of being renamed and the modification date updated. There are likely hundreds of instances of category domain objects in the ActionBazaar such as 'Junkyard Cars for Teenagers', 'Psychedelic Home Décor from the Sixties', "Cheesy Romantic Novels for the Bored Housewife", and so on.

Relations

In Java terms, a relation is manifested as one Object having a reference to another. If the `Item` and `Bid` Objects are related, there is probably a `Bid` instance variable in `Item`, an `Item` instance variable in `Bid` or both. Where the Object reference resides determines the direction of the arrows in Figure 7.3. If `Item` has a reference to `Bid`, the arrow should point from `Item` to `Bid`. As it so happens, in our case `Item` and `Bid` have references to each other (an `Item` *has* `Bids` on it and `Bids` are *placed on* `Items`). Signifying this fact, the arrow connecting `Bid` and `Item` points in both directions in Figure 7.3. This is what is meant by a *bi-directional* relationship or association as opposed to a *uni-directional* association or relation. Typically objects are nouns and associations are verbs such as: *has, is part of, is member of, belongs to*, and so on.

Multiplicity or Cardinality

As you can probably infer from Figure 7.3, not all relationships are one-to-one. That is, there may be more than one object on either side of a relationship. For example, a `Category` can have more than one `Item`. Multiplicity refers to this multifaceted nature of relationships. The multiplicity of a relationship can be:

- One-to-one: Each side of the relationship may have at most only one object. An employee can have only one ID card and an ID card can only be assigned to one employee.
- One-to-many: A particular object instance may be related to at least two instances of another. For example, an `Item` can have more than one `Bid`. Note, taken from the point of view of a `Bid`, the relationship is said to be *many-to-one*. For example, many Bids can be placed by a `Bidder` in Figure 7.3.
- Many-to-many: If both sides of the relationship may have more than one object, the relationship is many-to-many. For example, an `Item` can be in more than one `Category` and a `Category` can have multiple `Items`.

Optionality or Ordinality

Ordinality or optionality of a relationship determines whether an associated entity exists. For example we have bi-directional one-to-one association between `User` and `BillingInfo` and every user need not always have billing information so the relationship is *optional*. However, `BillingInfo` always belongs to a `User` and hence the optionality for BillingInfo-User association is false.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Rich vs. Anemic Domain Models

As we mentioned, domain models are eventually persisted into the database. It might already be obvious that it is very easy to make the domain model objects look exactly like database tables. As a matter of fact, this is exactly why data modeling is often synonymous to domain modeling and DBAs are often the domain experts. In this mode of thinking, domain objects contain attributes (mapping to database table columns) but no behavior. This type of model is referred to as the *anemic model.*

A *rich domain model* on the other hand, encapsulates both object attributes and behavior and utilizes objected oriented design such as inheritance, polymorphism and encapsulation.

Note an anemic domain model may not necessarily be a very bad thing for some applications. For one, it is very painless to map objects to the database. As a rule of thumb, the richer the domain model is, the harder it is to map it to the database, particularly while using inheritance.

Having established the basic concepts of domain modeling, we can now start discussing how the domain model is persisted using the EJB 3.0 Java Persistence API and actually start implementing our domain model.

7. 1.4 The EJB 3.0 Java Persistence API

In contrast to EJB 2.x Entity Beans, the EJB 3.0 Java Persistence API (JPA) is a metadata driven POJO technology. That is, to save data held in Java Objects into a database, our Objects are not required to implement an interface, extend a class or fit into a framework pattern. In fact, persisted objects need not contain a single inline statement of JPA . All we have to do is code our domain model as plain Java Objects and use annotations or the XML to give the persistence provider the following information:

1. What our domain objects are (for example using the `@Entity` and `@Embedded` notations).
2. How to uniquely identify a persisted domain object (for example using the `@Id` annotation).
3. What the relations between objects are (for example using the `@OneToOne`, `@OneToMany`, `@ManyToMany` annotations).
4. How the domain object is mapped to database tables (for example using various Object-Relational mapping annotations like `@Table`, `@Column`, `@JoinColumn`, etc).

As we can see, although Object-Relational mapping using the JPA (or any other O/R frameworks like Hibernate) is a great improvement over Entity Beans or JDBC, automated persistence is still an inherently a complex activity. The large number of persistence-related annotations and wide array of possible arrangements is a result of this fact. To make things as digestible as possible, we will only cover steps 1-3 in this chapter, leaving step 4 to Chapter 8. Moreover, we will stray from our pattern of presenting then analyzing a complete example since the wide breadth of the persistence API would not yield to the pattern nicely. Instead, we are going to explore the persistence API by visiting each step in our list using specific cases from the ActionBazaar example, analyzing features and intricacies

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

on the way. Not straying from previous Chapters, however, we will still focus on using annotations, leaving the description of deployment descriptor equivalents for a brief discussion in chapter 11. We will implement our first step next by coding a domain object from ActionBazaar.

SQL-Centric Persistence: Spring JDBCTemplate and iBATIS

Like many of us, if you are very conformable with SQL, JDBC and like the control and flexibility offered by do-it-yourself, hands-on approach, O/R in its full-blown black magic, automated form may not be for you. As a matter of fact, O/R tools like Hibernate and EJB 3.0 Java persistence API (JPA) might seem like overkill or an unnecessary learning curve to overcome, even despite the long-term benefits offered by a higher-level API.

If this is the case, you should give tools like Spring JDBCTemplate and iBATIS a very close look. Both of these tools do an excellent job abstracting out really low-level, verbose JDBC mechanics while keeping the SQL/Database-centric feel of persistence intact.

However, you should give O/R Frameworks and JPA a fair chance. You just might find that it makes your life a lot easier/OO-centric, freeing you to use your neuron-cycles to solve business problems instead.

7.1.5 Domain Objects as Java Classes

We now get our feet wet by examining some code for the JPA. We pick a representatively complex domain object, `Category`, and see how it might look like in Java code. The `Category` class in Listing 7.1 is a simple POJO class that is a domain object built using Java. This is a candidate for becoming an entity and to be persisted to the database. As we mentioned earlier, the `Category` domain object may have the category name and modification date as attributes. In addition, there are a number of instance variables in the POJO Class that express domain relationships instead of simple attributes of a category. The `id` attribute also does more than simply serving as a data-holder for business logic and identifies an instance the Category object. You will learn about identity in the next section.

Listing 7.1: Category domain object in Java

```
package ejb3inaction.actionbazaar.model;
import java.sql.Date;

public class Category {                                              |#1
    protected Long id;                                               |#2

    protected String name;                                           |#3
    protected Date modificationDate;                                 |#3

    protected Set<Item> items;                                       |#4
    protected Category parentCategory;                               |#4
    protected Set<Category> subCategories;                           |#4
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    public Category() {}                                                    |#5

    public Long getId() {                                                   |#6
       return this.id;                                                      |#6
    }                                                                       |#6
                                                                            |#6
    public void setId(Long id) {                                            |#6
        this.id = id;                                                       |#6
    }                                                                       |#6
                                                                            |#6
    public String getName() {                                               |#6
       return this.name;                                                    |#6
    }                                                                       |#6
                                                                            |#6
    public void setName(String name) {                                      |#6
        this.name = name;                                                   |#6
    }                                                                       |#6
                                                                            |#6
    public Date getModificationDate() {                                     |#6
       return this.modificationDate;                                        |#6
    }                                                                       |#6
                                                                            |#6
    public void setModificationDate(Date modificationDate) {                |#6
        this.modificationDate = modificationDate;                           |#6
    }                                                                       |#6
                                                                            |#6
    public Set<Item> getItems() {                                           |#6
       return this.items;                                                   |#6
    }                                                                       |#6
                                                                            |#6
    public void setItems(Set<Item> items) {                                 |#6
        this.items = items;                                                 |#6
    }                                                                       |#6
                                                                            |#6
    public Set<Category> getSubCategories() {                               |#6
       return this.subCategories;                                           |#6
    }                                                                       |#6
                                                                            |#6
    public void setSubCategories(Set<Category> subCategories) {             |#6
        this.subCategories = subCategories;                                 |#6
    }                                                                       |#6
                                                                            |#6
    public Category getParentCategory() {                                   |#6
       return this.parentCategory;                                          |#6
    }                                                                       |#6
                                                                            |#6
    public void setParentCategory(Category parentCategory) {                |#6
        this.parentCategory = parentCategory;                               |#6
    }                                                                       |#6
}
```
(annotation) <#1 Plain Java Object>
(annotation) <#2 Instance Variable Uniquely Identifying Object>
(annotation) <#3 Object Attribute Instance Variable>
(annotation) <#4 Instance Variables for Relations>
(annotation) <#5 Empty Constructor>
(annotation) <#6 Getters and Setters For Each Instance Variable>

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The `Category` POJO has a number of protected instance fields#3 #4, each with corresponding setters and getters that conform to JavaBeans naming conventions#6. In case you are unfamiliar with them, JavaBeans rules state that all instance variables should be non-public and made accessible via methods that follow the *getXX* and *setXX* pattern used in Listing 7.1, where *XX* is the name of the property (instance variable). Other than `name` and `modificationDate`, all the other properties have a specific role in domain modeling and persistence. The `id` field is used to store a unique number used to identify the category#2. The `items` property stores all the items stored under a category and represents a many-to-many relationship between items and categories. The `parentCategory` property represents a self-referential many-to-one relationship between parent and child categories. Finally, the `subCategories` property maintains a many-to-many relationship between parent categories and subcategories.

The `Category` Class as it stands in Listing 7.1 is a perfectly acceptable Java implementation of a domain object. The problem is that the EJB 3.0 persistence provider has no way of distinguishing the fact that the `Category` Class is a domain object instead of just another random Java Object used for business logic, presentation, or some other purpose. Moreover, note that the properties representing relationships do not make direction or multiplicity clear. Lastly, the persistence provider also needs to be told about the special purpose of the `id` property. We will start solving some of these problems by using JPA annotation next, starting with identifying the `Category` Class as a domain object.

7.2 Implementing Domain Objects with JPA

In the previous few sections you learnt about domain modeling concepts and identified part of the ActionBazaar domain model. Also we briefly introduced some commonly used metadata annotations supported by JPA. In this section we will see some of the JPA annotations in action as we implement part of the domain model using EJB 3.0 JPA. We will start with @Entity:using annotation that converts a POJO to an entity. Then we will learn about field and property-based persistence and entity identity. Finally we will discover embedded objects.

7.2.1 The @Entity Annotation

The `@Entity` annotation marks a POJO as a domain object that can be uniquely identified. You may think of the annotation as the persistence counterpart of the `@Stateless`, `@Stateful` and `@MessageDriven` annotations. We would mark the `Category` Class as an Entity as follows:

```
@Entity
public class Category {
     ...
     public Category() { ... }
     public Category(String name) { ... }
     ...
}
```

As the code snippet demonstrates, all non-abstract entities must have either a public or protected no-argument constructor. The constructor is used to create a new entity instance by using *new* operation as follows:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
Category category = new Category();
```

One of the coolest features of JPA is that since Entities are POJOs, they support a full range of OO inheritance features, with a few persistence related nuances thrown in. You can have an Entity extend either another Entity or even a non-Entity Class. For example, it would be good design to extend both the `Seller` and `Bidder` domain object Classes from a common `User` Class as depicted in Figure 7.4.

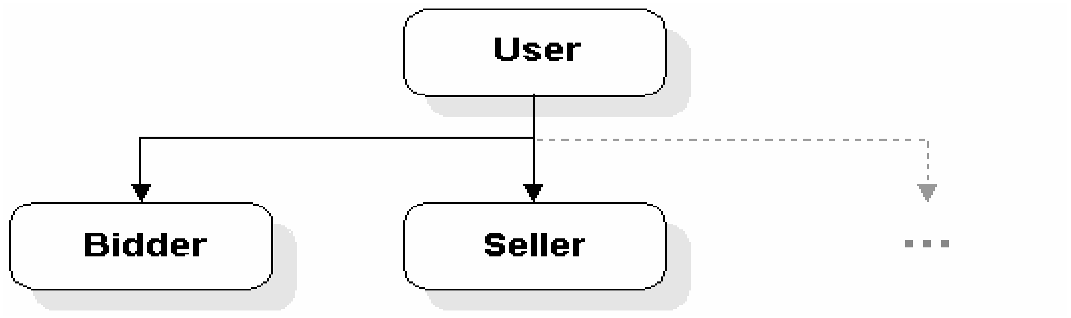


Figure 7.4 Inheritance Support with Entities. Bidder, Seller entities extend the User entity class

As the code snippet that follows shows, this Class could store information common to all users like the user ID, username and email address.

```
@Entity
public class User {                                                    |#1
    ...
    String userId;
    String username;
    String email;
    ...
}

@Entity
public class Seller extends User { ...                                 |#2


@Entity
public class Bidder extends User { ...                                 |#2
```
(annotation) <#1 Parent Entity Class>
(annotation) <#2 Entity Subclasses>

Because the parent `User` class is declared an Entity, all the inherited fields like `username` and `email` are persisted when either the `Seller` or `Bidder` Entities are saved. A slightly counter-intuitive nuance you should note is that this would not be the case if the `User` Class were not an Entity itself. Rather the value of the inherited properties would be discarded when either `Seller` or `Bidder` are persisted. The preceding code snippet also demonstrates an interesting weakness—the `User` class could be persisted on its own, which is not necessarily desirable or appropriate application behavior. One way to avoid this problem is to declare the `User` Class abstract, since abstract Entities are allowed but cannot be directly instantiated or saved. In any case, this is probably

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

better OO design anyway. Since JPA supports entity inheritance the relationship between entities and queries may be polymorphic. We will discuss about handling polymorphic queries in Chapter 10.

Obviously the ultimate goal of persistence is to save the properties of the Entity into the database (such as name and modification date for the `Category` Entity in Listing 7.1). However, things are not as simple as they seem and there are a few twists about Entity data persistence you need to have a good grasp of.

7.2.2 Persisting Entity Data

An entity being a persistent object has some state that is stored into the database. In this section we will discuss access types, defining a transient field and datatypes supported by JPA.

Field vs Property based Persistence

An entity maintains its state by using either fields or properties (via setter and getter methods). Although JavaBeans Object property-naming conventions have been widely used in the Java platform for a good number of years, some developers consider these conventions to be overkill and would rather access instance variables directly. The good news is that JPA supports this paradigm (whether it should is an open question. We will express our viewpoint a few paragraphs later). Defining O-R mapping using fields or instance variables of entity is known as *field-based accesss* whereas using O-R mapping with properties is known as *property-based access*.

If you want to use *field-based access*, you can declare all your POJO persisted data fields public or protected and ask the persistence provider to ignore getters/setters altogether. You would only have to provide some indication on at least one of the instance variables that they are to be used for persistence directly. You can do this by using the @Id annotation that we will discuss next that apply to either a property on a field . Depending on your inclination, this transparent flexibility may or may not seem a little counter-intuitive. In the early releases of the EJB 3.0 specification the `@Entity` annotation had an element named `accessType` to explicitly specify the persistence data storage type to be either `FIELD` or `PROPERTY`. You will learn that O-R mapping using XMLprovides an element named `access` to specify the access type. However, many developers did not like this element and wanted additional flexibility in having the JPA provider dynamically determine the access type based on Entity field usage patterns.

In the snippet that follows, shows you what field-based persistence might look like:

```
@Entity
public class Category {
    @Id
    public Long id;

    public String name;
    public Date modificationDate;


    public Category() {}
}
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

In the code snippet, the persistence provider would infer that the id, name, and modificationDate public fields should be persisted since the @Id annotation is used on the id field. The annotations would have been applied to getters if we did not intend to use fields for persistence instead of properties.

Note that annotations used with a setter method is ignored by the persistence provider for property-based access.

One caveat in choosing between field and property-based persistence is that both are one-way-streets; you cannot mix and match access types in the same Entity or in any Entity in the POJO hierarchy. Field-based persistence is a one-way-street in another important way: you give up the OO benefits of encapsulation/data-hiding that you get from getters and setters if you expose the persistence fields to be directly manipulated by clients. Even if you used field-based access we recommend that you make the fields private and expose the fields to be modified by getter/setter method.

For example, property setters are often used in non-trivial applications to validate the new data being set or standardize POJO data in some fashion. In our example, we could automatically convert Category names to uppercase in the setName method:

```
public void setName(String name) {
    this.name = name.toUpperCase();
}
```

In general, we highly recommend that you use field-based access with accessor methods or property-based access. It is much easier to have it and not need it than to find out that you need it later on and have to engage in a large-scale, painful refactoring effort in the face of deadlines.

By default, the persistence provider saves all Entity fields or properties that have JavaBeans style public or protected setters and getters (for example, getName, setName in Listing 7.1). In addition, persisted setters and getter cannot be declared final, as the Persistence provider may need to override them.

Defining a transient field

If necessary, you can stop an Entity property from being persisted by marking the getter with the @Transient annotation. A transient field is typically useful for caching some data that you do not want to save in the database. For example, the Category Entity could have a property named activeUserCount that stores the number of active users currently browsing items under the directory or a generatedName field that is generated by concatenating category id and name. However, saving this runtime information into the database would not make much sense. We could avoid saving the property by using @Transient as follows:

```
@Entity
public class Category {
    ...
    @Transient
    protected Long activeUserCount;
    transient public String generatedName

    ...
        public Long getActiveUserCount() {
        return activeUserCount;
    }
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    public void setActiveUserCount(Long activeUserCount) {
        this.activeUserCount = activeUserCount;
    }
    ...
}
```

We could have achieved the same effect by using the `@Transient` tag on the getter when using property-based access.

Note that defining a field with `transient` modifier as we have used with `generatedName` has the same effect as `@Transient` annotation:

Persistent Data types

Before we move on from the topic of persisted POJO data and start discussing identity and relations, we need to discuss exactly what field data can be persisted. Ultimately persisted fields/properties wind up in a relational database table and have to go through an extremely powerful, high-level API to get there. Because of this fact, there are some restrictions on what data types can be used in a persisted field/property. In general, these restrictions are not very limiting, but you should be aware of them nonetheless. Table 7.1 lists the data types that can be used in a persistent field/property:

1.15 Table 7.1: Data types allowable for a persisted field/property

Types	**Examples**
Java primitives	Int, double, long
Java primitives wrappers	java.lang.Integer, java.lang.Double
String type	java.lang.String
Java API Serializable types	java.math.BigInteger, java.sql.Date
User defined Serializable types	Class that implements java.io.Serializable
Array types	byte[], char[]
Enumerated type	{SELLER, BIDDER, CSR, ADMIN}
Collection of Entity types	Set<Category>
Embeddable class	Classes that are defined @Embeddable

We will discuss the `@Embeddable` annotation in section 7.5.3 after we discuss Entity Identities. For now, think of an Embeddable Object as a custom data type for an Entity that encapsulates persisted data.

We have already touched on the issue of identity when we talked about uniquely identifying the `Category` domain object through the `id` property. We will now take up the issue of Entity Identity in greater detail.

7.2.3 Specifying Entity Identity

Every Entity of the domain model must be uniquely identifiable. This requirement partly comes from the fact that at some point Entities must be persisted into a uniquely identifiable row in a database table (or set of rows in multiple tables). If you are familiar with the concept of database table primary keys, this should come as no surprise. Without primary keys, you would never be able

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

to uniquely identify and retrieve the data you put into a record since you would not know which row it went into after performing the save! The concept of being able to distinguish different instances of the same object holding a different set of data is not completely alien to object oriented programming either. Consider the `equals` method in `java.lang.Object`, meant to be overridden by subclasses as necessary. This method is the OO equivalent of comparing the primary keys of two distinct database records. In most cases, the `equals` method is implemented by comparing the data that uniquely identifies instances of the same object from one another. In the case of the `Category` Object, you might imagine that the `equals` method would look like this:

```
public boolean equals (Object other) {
    if (other instanceof Category) {
        return this.name.equals(((Category)other).name)
    } else {
        return false;
    }
}
```

In this case, we would be assuming that the `name` instance variable uniquely identifies a `Category`. The `name` field therefore is the *identity* for the `Category` Object. In Listing 7.1, however, we choose the `id` field as the identity for `Category`. This choice will be more obvious when we talk about mapping the `Category` Object into a database table. As we will see, in effect, we choose this instance variable because we get it free from the database as a unique `Category` identifier and it is less resource intensive than comparing the `java.lang.String` `name` field since it is a numeric `java.lang.Long`. There are several ways of telling the persistence provider where the identity of an Entity is stored. Starting with the simplest and ending with the most complex, these are:

1. Using the `@Id` Annotation.
2. Using the `@IdClass` Annotation.
3. Using the `@EmbeddedId` Annotation.

We will now look at each of these mechanisms next.

The @Id Annotation

Using the `javax.persistence.Id` annotations is the simplest way of telling the persistence provider where the Entity identity is stored. The `@Id` annotation marks a field or property as identity for an Entity. Since we are using property based persistence for the `Category` Entity, we could let the API know that we are using the `id` property as the identity by applying the `@Id` annotation to the `getId` method as in the following code snippet. In case of field-based persistence, the `@Id` annotation would have been applied directly to an instance variable instead.

```
@Entity
public class Category {
    ...
    protected Long id;
    ...

    @Id
    public Long getId() {
       return this.id;
    }

    public void setId(Long id) {
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
        this.id = id;
    }
    ...
}
```

Because the identity we specify will end up in a database primary key column, there are limitations to what data types an identity might have. EJB 3.0 supports primitives, primitive wrappers and Serializable types like `java.lang.String`, `java.util.Date`, and `java.sql.Date` as identities. In addition, when choosing numeric types you should avoid types such as `float`, `Float`, `double`, etc because of the indeterminate nature of type precision. For example, let us assume that we are using float data as the identity and specify 103.789 and 103.787 as the identity values for two separate Entitiy instances. If the database rounds these values to two-digit decimal precision before storing the record, both of these values would map to 103.79 and we would have a primary key violation!

An important thing to note is that using `@Id` annotation on its own works only for identities with just one field or property. In reality, you will often have to use more than one property or field (known as composite key) to uniquely identify an Entity. For sake of illustration, let us assume that we changed our minds and decided that a `Category` is uniquely identified by its name and creation date. There are two ways we can accomplish this: either by using the `@IdClass` or `@EmbeddedId` annotations.

The IdClass Annotation

In effect, the `@IdClass` annotation enables us to use more than one `@Id` annotation in a sensible way. This is the basic problem with using more than one `@Id` field or property in an Entity class: it is not obvious how to compare two instances in an automated fashion. This is especially true since in cases where composite keys are necessary, one or more of the fields that constitute the primary key are often relation or association fields. For example, although this is not the case for us, the `Bid` domain object might have an identity consisting of the `item` to bid on, the `bidder` as well as a bid amount.

```
public class Bid {
    private Item item;
    private Bidder bidder;
    private Double amount;
    ...
}
```

In the preceding snippet, both the `item` and `bidder` instance variables represent relation references to other entities. It might be that neither of the references is simple enough to compare instances using the `equals` method, as it would be for a `java.lang.String` or `java.lang.Long`. This is where a designated `IdClass` comes in. The best way to understand how this works is though an example. For simplicity, we will return to our `Category` object with the name and creation date identity. This is how the solution might look like:

Listing 7.2: Specifying Category Identity Using IdClass

```
public class CategoryPK implements Serializable {
    String name;                                                    |#1
    Date createDate;                                                |#1

    public CategoryPK() {}                                          |#2

    public boolean equals(Object other) {                           |#3
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
        if (other instanceof CategoryPK) {
            final CategoryPK otherCategoryPK = (CategoryPK)other;
            return (otherCategory.name.equals(name) &&
                otherCategoryPK.createDate.equals(createDate));
        }

        return false;
    }

    public int hashCode() {                                         |#4
        return super.hashCode();
    }
}

@Entity
@IdClass(CategoryPK.class)                                          |#5
public class Category {
    public Category() {}

    @Id                                                             |#6
    protected String name;                                          |#6

    @Id                                                             |#6
    protected Date createDate;                                      |#6
    ...
}
```

(annotation) <#1 Stored Indentity Fields>
(annotation) <#2 Empty Constructor>
(annotation) <#3 equals method comparing Identity>
(annotation) <#4 Hashcode Implementation>
(annotation) <#5 IdClass Specification>
(annotation) <#6 Identity Fields>

As shown in Listing 7.2, the `CategoryPK` Class is designated as the `IdClass` for `Category`#6. The `Category` Class has two Identity fields marked by the `@Id` annotation, `name` and `creationDate`#6. These two Identity fields are mirrored in the `CategoryPK` Class#1. The `equals` method implemented in `CategoryPK` compares the two mirrored Identity fields to determine if two given identities are equal#3. The magic here is that at runtime, the persistence provider determines if two `Category` Objects are equal by copying the marked `@Id` fields into the corresponding fields of the `CategoryPK` Object and using `CategoryPK.equals`. Note that any IdClass must be Serializable and must provide a valid `hashCode` implementation#4. In effect, all that is happening here is that we are specifying exactly how to compare multiple identity fields using an external `IdClass`#5. The disadvantage to using `@IdClass` is the slight redundancy and associated maintainability problems in repeating the definition of identity fields in the Entity and the `IdClass`. In our case the `name` and `createDate` fields are defined in both the `Category` and `CategoryPK` Classes. However, the `IdClass` approach keeps your domain model clutter free, especially as opposed to the slightly awkward object model proposed by the third approach, which uses the `@EmbeddedId` annotation.

The @EmbeddedId Annotation

Using the `@EmbeddedId` annotation is like moving the `IdClass` right into your Entity and using the identity fields nested inside it to store Entity data. Take a look at what we mean in the following code snippet that rewrites Listing 7.2 using `@EmbededId`:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

21 Listing 7.3: Specifying Category Identity Using EmbeddedId

```
@Embeddable                                                         |#1
public class CategoryPK {
    String name;
    Date createDate;

    public CategoryPK() {}

    public boolean equals(Object other) {                           |#2
        if (other instanceof CategoryPK) {
            final CategoryPK otherCategoryPK = (CategoryPK)other;
            return (otherCategory.name.equals(name) &&
                otherCategoryPK.createDate.equals(createDate));
        }

        return false;
    }

    public int hashCode() {                                         |#2
        return super.hashCode();
    }
}

@Entity
public class Category {
    public Category() {}

    @EmbeddedId                                                     |#3
    protected CategoryPK categoryPK;
    ...
}
```

(annotation) <#1 Embeddable Identity Class>
(annotation) <#2 Still Includes Equals and HashCode>
(annotation) <#3 Marking as Identity>

In Listing 7.3, notice that the identity fields, `name` and `createDate`, are absent altogether from the `Category` Class. Instead an "Embeddable" Object instance, `categoryPK`, is designated as the identity using the `@EmbeddedId` annotation#3. The `CategoryPK` Object itself is almost identical to the `IdClass` used in Listing 7.1 and contains the `name` and `createDate` fields. We still need to implement the `equals` and `hashCode` methods#2. The only difference is that the `@Embedded` Object need not be `Serializable`. In effect, the Object designated as `@EmbeddedId` is expected to be a simple data holder encapsulating only the identity fields. Note, the `@Id` annotation is missing altogether since it is redundant. As a matter of fact, you are not allowed to use `Id` or `IdClass` in conjunction with `EmbeddedId`. As you can see, although this approach saves typing, it is a little awkward to justify in terms of Object modeling (even the variable name, `categoryPK`, is more reminiscent of relational databases than OO). It is a little unwieldy too. Imagine having to write `category.catetogyPK.name` to use the name field for any other purpose than as a primary key, as opposed to using `category.name`. However, which method you choose is ultimately a matter of personal taste.

Unless you really need a composite primary key because you are stuck with a legacy database we do not recommend using it and instead recommend a simple generated key (also known as surrogate key) that we will discuss in Chapter 8.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

One concept is critical: identities can only be defined *once* in an entire Entity hierarchy. Having had a sufficient introduction to the idea of Entities and identities, we are now ready to explore the `@Embeddable` annotation in greater detail.

7.2.4 The @Embeddable Annotation

Let us step back a second from the idea of identities into the world of pure OO domain modeling. Are all domain objects always identifiable on their own? How about Objects that are simply used as convenient data holders/groupings inside other Objects? An easy example would be an `Address` Object used inside a `User` Object as an elegant OO alternative to listing street address, city, zip, etc directly as fields of the `User` object. It would be overkill for the `Address` Object to have an identity since it is not likely to be used outside a `User` Object. This is exactly the kind of scenario the `@Embeddable` annotation was designed for. The `@Embeddable` annotation is used to designate persistent Objects that need not have an identity of their own. This is because `Embeddable` Objects are identified by the Entity Objects they are nested inside and never persisted or accessed on their own. Put another way, `Embeddable` Objects share their identities with the enclosing Entity. An extreme case of this is the `@EmbeddedId` situation where the embeddable object *is* the identity. Let us take a look at a code snippet showing the user/address example to get a better understanding of the most commonly used `Embeddable` Object semantic patterns:

Listing 7.4: Using Embedded Objects

```
@Embeddable                                                         |#1
public class Address {
    protected String streetLine1;
    protected String streetLine2;
    protected String city;
    protected String state;
    protected String zipCode;
    protected String country;
    ...
}

@Entity
public class User {
    @Id
    protected Long id;                                              |#2
    protected String username;
    protected String firstName;
    protected String lastName;
    @Embedded
    protected Address address;                                      |#3
    protected String email;
    protected String phone;
    ...
}
```
(annotation) <#1 Embeddable Address>
(annotation) <#2 Shared Identity>
(annotation) <#3 Embedded Address>

In Listing 7.4, the embeddable `Address` Object#1 is embedded inside a `User` Entity#3 and shares the identity marked with the `@Id` annotation#2. It is illegal for an `@Embeddable` object to have an identity. Also, the EJB 3.0 API does not support nested embedded Objects. In most cases, embedded objects are stored in the same database record as the entity and are only materialized in the OO world. We will show you how this is done in Chapter 8.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Our discussion of Embedded Objects rounds out the coverage of domain Objects. We will take a look at domain object relationships next.

7.3 Entity Relationships

In section 7.2.1, we explored the concepts of domain relationships, direction and multiplicity. As review, we will summarize those concepts here before diving into the details of how to specify domain relationships using the JPA. As you might have noted in our domain object code samples, a relationship essentially means that one Entity holds an object reference to another. For example, the `Bid` Object holds a reference to the `Item` Object the bid was placed on. Hence, there is a relationship between the `Bid` and `Item` domain objects. Recall that relationships can be either unidirectional or bidirectional. The relationship between `Bidder` and `Bid` in Figure 7.3 is unidirectional, since the `Bidder` object has a reference to `Bid`, but the `Bid` object has no reference to the `Bidder`. The `Bid-Item` relationship, on the other hand, is bidirectional, meaning both the `Bidder` and `Item` Objects have references to each other. Relationships can be one-to-one, one-to-many, many-to-one or many-to-many. Each of these relationship types is expressed in the JPA through an annotation. Table 7.2 lists the relationship annotations we will discuss in the following sections.

Table 7.2: Domain Relation types and corresponding annotations

Type of relationship	**Annotation**
One-to-one	@OneToOne
One-to-many	@OneToMany
Many-to-one	@ManyToOne
Many-to-many	@ManyToMany

We explore each annotation using examples next.

7.3.1 @OneToOne

The `@OneToOne` annotation is used to mark uni- and bi-directional one-to-one relationships. Although in most systems, one-to-one relationships are rare, they make perfect sense for domain modeling. In fact, our ActionBazaar example in Figure 7.3 has no one-to-one relationships. However, we can imagine that the `User` domain object parent to both `Seller` and `Bidder` has a one-to-one relationship with a `BillingInfo` object. The `BillingInfo` object might contain billing data on a user's credit card, bank account, and so on. Let's start by seeing what a unidirectional relationship would look like as depicted in figure 7.5.

Unidirectional One-to-One

For the time, being, let us assume that the `User` object has a reference to the `BillingInfo`, but not vice-versa. That is, the relationship is uni-directional.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

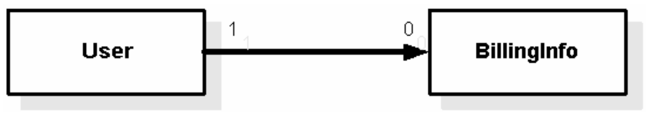


Figure 7.5: One-to-One relationship between User and BillingInfo entities. A User may have atmost one instance of BillingInfo object and BillingInfo object cannot exist without a User.

The following code illustrates this relationship:

Listing 7.5: Unidirectional OneToOne Relationship

```
@Entity
public class User {
    @Id
    protected String userId;
    protected String email;
    @OneToOne                                                    |#1
    protected BillingInfo billingInfo;
}

@Entity
public class BillingInfo {
    @Id
    protected Long billingId;
    protected String creditCardType;
    protected String creditCardNumber;
    protected String nameOnCreditCard;
    protected Date creditCardExpiration;
    protected String bankAccountNumber;
    protected String bankName;
    protected String routingNumber;
}
```

(annotation) <#1 One-to-one relationship between User and BillingInfo>

In Listing 7.5, the `User` Class holds a `BillingInfo` reference in the persisted `billingInfo` field. Since the `billingInfo` variable holds only one instance of the `BillingInfo` Class, the relationship is one-to-one. The `@OneToOne` annotation indicates that the persistence provider should maintain this relationship in the database#1. Let us take a closer look at the definition of the `@OneToOne` annotation to understand its features better:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

First, note that this annotation can be applied to either fields or properties since the `Target` is specified to be `METHOD, FIELD`. We are using field-based persistence for the examples to keep things simple. The `targetEntity` element tells the persistence provider what the related Entity Class is. In most cases, this is redundant since the container can infer the Class from the Class of the field or the return type of the property getter and setter. However, you can specify it explicitly anyway if you prefer. We will see a case in which this element is indispensable when we explore one-to-many relations. The `cascade` and `fetch` parameters are best discussed after we introduce

Licensed to John Sweitzer <admin@saolailaem.info>

Object-relational mapping in the next Chapter. For now, suffice it to say that `cascade` controls what happens to related data when the relation is altered or deleted and `fetch` specifies when and how the related fields are populated from database tables.

Listing 7.6 shows an example of how the `@OneToOne` annotation might be applied to a property instead of a field.

22 Listing 7.6: Property Based Unidirectional OneToOne Relationship

```
@Entity
public class User {
    private Long userId;
    private String email;
    private BillingInfo billing;
    ...
    @OneToOne                                                  |#1
    public BillingInfo getBilling() {
        this.billing;
    }

    public void setBilling(BillingInfo billing) {
        this.billing = billing;
    }
}

@Entity
public class BillingInfo {
    private Long billingId;
    private String creditCardType;
    ...
}
```
(annotation) <#1 One-to-one relationship between User and BillingInfo Using Properties>

The `optional` element tells the persistence provider if the related Object must always be present. By default, this is set to `true`, which means that a corresponding related object need not exist for the Entity to exist. In our case, every user need not always have billing information (for example if the user just signed up) so the relationship is *optional* and the `billing` field can sometimes be `null`. If the `optional` parameter was set to `false`, the Entity cannot exist if the relationship or association does not hold. In other words, no `User` without `BillingInfo` could ever exist. We will see the `mappedBy` parameter in action in the next section when we discuss bi-directional associations.

Bidirectional One-to-One

The real point of having domain relationships between Entities is to be able to reach one Entity from another. In our previous example, we can easily reach the billing information through the `billingInfo` reference when we have an instance of a `User`. In some cases, you need to be able to access related Entities from either side of the relationship (admittedly, this is rare for one-to-one relationships). For example, the ActionBazaar application may periodically check for credit card expiration dates and notify users of imminently expiring credit cards. As a result, the application should be able to access user information from a given `BillingInfo` Entity and the `User-BillingInfo` relationship should really be bidirectional. In effect, bidirectional one-to-one relationships are implemented using two `@OneToOne` annotations pointing to each other on either

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

side of the bidirectional relation. Let us see how this works in Listing 7.7 by refactoring the code from Listing 7.5:

23 Listing 7.7: Bidirectional OneToOne Relationship

```
@Entity
public class User {
    @Id
    protected String userId;
    protected String email;
    @OneToOne                                                         |#1
    protected BillingInfo billingInfo;
}

@Entity
public class BillingInfo {
    @Id
    protected Long billingId;
    protected String creditCardType;
    protected String creditCardNumber;
    protected String nameOnCreditCard;
    protected Date creditCardExpiration;
    protected String bankAccountNumber;
    protected String bankName;
    protected String routingNumber;
    @OneToOne(mappedBy="billingInfo", optional="false");               #2
    protected User user;
}
```

(annotation) <#1 One-to-One Relationship Between User and BillingInfo>
(annotation) <#1 Reciprocal Relationship to User>

In listing 7.7, the `User` class still has a relation to the `BillingInfo` Class through the `billingInfo` variable#1. However, in this case the relationship is bidirectional because the `BillingInfo` class also has a reference to the `User` Class through the `user` field#2. The `@OneToOne` annotation on the `user` field has two more interesting things going on. The first is the `mappedBy="billingInfo"` specification#2. This is telling the container that the "owning" side of the relationship exists in the `User` Class's `billingInfo` instance variable. The concept of a *relationship owner* is really not originated from domain modeling. It exists as a convenience to define the database mapping for a relationship only once instead of repeating the same mapping for both directions of a relationship. We will see this concept in action in Chapter 8 when we describe O-R mapping. For now, you should simply note the role of the `mappedBy` attribute. The second interesting feature of the `@OneToOne` annotation on the `user` field is that the `optional` parameter is set to `false` this time. This means that a `BillingInfo` Object cannot exist without a related `User` object. After all, why bother storing credit card or bank account information that is not related to an existing user?

7.3.2 @OneToMany and @ManyToOne

As you might have gathered from the ActionBazaar domain model in Figure 7.3, one-to-many and many-to-one relations are the most common in enterprise systems. In this type of relationship, one Entity will have two or more references of another. In the Java world, this usually means that an Entity has a collection-type field such as `java.util.Set` or `java.util.List` storing multiple

Licensed to John Sweitzer <admin@saolailaem.info>

instances of another Entity. Also, if the association between two Entities is bi-directional, one side of the association is one-to-many and the opposite side of association is many-to-one.

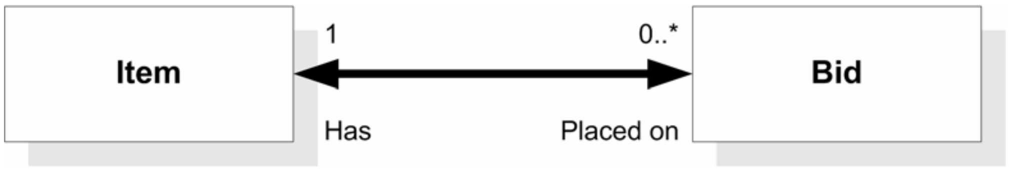


Figure 7.6: Every Item has one or more Bids where more than one Bids may be placed on an Item. Hence relationship between Item and Bid is one-to-many where relationship between Bid-Item is many-to-one.

In Figure 7.6, the relationship between `Bid` and `Item` is one-to-many from the perspective of the `Item` object, while it is many-to-one from the perspective of the `Bid`. Similar to the one-to-one case, we can mark the owning side of the relationship by using the `mappedBy` column on the Entity that is not the owner of the relationship. We will analyze these relationships further by actually coding the `Bid-Item` relationship in Listing 7.8:

Listing 7.8: One-Many Bidirectional Relationship

```
@Entity
public class Item {
    @Id
    protected Long itemId;
    protected String title;
    protected String description;
    protected Date postdate;
    ...
    @OneToMany(mappedBy="item")                                           |#1
    protected Set<Bid> bids;
    ...
}

@Entity
public class Bid {
    @Id
    protected Long bidId;
    protected Double amount;
    protected Date timestamp;
    ...
    @ManyToOne                                                            |#2
    protected Item item;
    ...
}
```

(annotation) <#1 One-to-Many Relationship>
(annotation) <#2 Corresponding Many-to-One Relationship>

[[JC: add some intervening text before the head]]

One-to-Many Relationship

Listing 7.8 shows that the `Item` domain object has a `Set` of `Bid` objects that it has references to. To signify this domain relationship, the `bids` field is marked with a `@OneToMany` annotation#1. There are a few nuances about the `@OneToMany` annotation we should talk about. In order to explore them, let us take a quick look at the definition of the annotation:

Licensed to John Sweitzer <admin@saolailaem.info>

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

If you notice, this is literally identical to the definition of the `@OneToOne` annotation, including the `mappedBy` element. As a matter of fact, the only element we need to discuss further is `targetEntity`. Remember that this element is used to specify the class of the related Entity if it is not immediately obvious. In the `@OneToMany` annotation used in listing 7.8, this parameter is omitted since we are using Java generics to specify the fact that the `bids` variable stores a `Set` of `Bid` Objects:

```
@OneToMany(mappedBy="item")
```

protected Set<Bid> bids;

Imagine, however, what would happen if we did not use generics on the `Set`. In this case, it would be impossible for the persistence provider to determine what Entity the `Item` object has a relation to. This is exactly the situation the `targetEntity` parameter is designed for. We would use it to specify the Entity at the other end of the `OneToMany` relation as follows:

```
@OneToMany(targetEntity=Bid.class,mappedBy="item")
protected Set bids;
```

[[JC: add some intervening text before the head]]

Many-to-One as owning-side of relationship

You should also note the `mappedBy="item"` value on the `@OneToMany` annotation specifying the owning side of the bidirectional relation to be the `items` field of the `Bid` Entity.

Because the relationship is bidirectional, The `Bid` domain object has a reference to an `Item` through the `item` variable#2. The `@ManyToOne` annotation on the `item` variable tells the persistence provider that more than one `Bid` Entity could hold references to the same `Item` instance. For bi-directional one-to-many relationships, `ManyToOne` is always the owning side of the relationship. Because of this fact, the `mappedBy` element does not exist in the definition of the `@ManyToOne` annotation:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

Other than this minor difference, all the other elements of the `@ManyToOne` annotation have the same purpose and functionality as the elements in the `@OneToOne` and `@OneToMany` annotations.

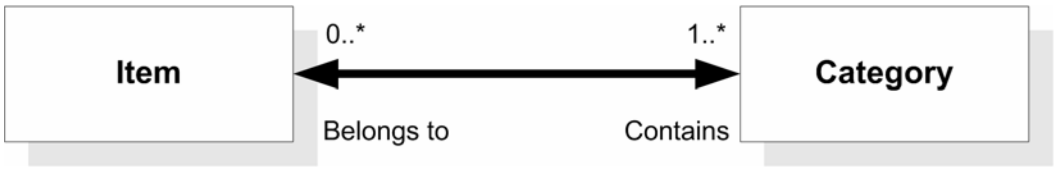
The last type of domain relationship is many-to-many, which we will discuss next.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

7.3.3 @ManyToMany

While not as common as one-many relations, many-to-many relations occur quite frequently in enterprise applications. In this type of relationship, both sides of the relationship might have multiple references to related Entities. In our ActionBazaar example, the relationship between Categoriy and Item is many-to-many as depicted in figure 7.7.



47 **Figure 7.7: The relationship beween Category and Item is many-to-many because every category may have one or more items whereas each item may belong to more than one categories.**

That is, a category can contain multiple items and an item can belong to multiple categories. For example, a category named "Sixties Fashion" could contain items like "Bellbottom Pants" and "Platform Shoes". "Bellbottom Pants" and "Platform Shoes" could also be listed under "Uncomfortable and Outdated Clothing." Although many-to-many relations can be unidirectional, they are often bidirectional because of their cross-connecting, mutually independent nature. Not too surprisingly, a bidirectional many-to-many relation is often represented by `@ManyToMany` annotations on opposite sides of the relation. Like the one-to-one and one-many relationships we can identify the owning-side of the relationship by specifying `mappedBy` on the "subordinate" entity and may have to use the `targetEntity` attribute if not using Java Generics.

The definition for `@ManyToMany` is identical to `OneToMany` and holds no special intricacies beyond those already discussed:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

To round off our discussion of many-to-many relations let us take a look at Listing 7.9 to see how the `Item-Category` relation might look like:

Listing 7.9 : Many-Many Relationship Between Category and Items

```
@Entity
public class Category {
    @Id
    protected Long categoryId;
    protected String name;
    ...
    @ManyToMany                                                  |#1
    protected Set<Item> items;
    ...
}

@Entity
public class Item {
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @Id
    protected Long itemId;
    protected String title;
    ...
    @ManyToMany(mappedBy="items")                                      |#2
    protected Set<Category> categories;
    ...
}
```
(annotation) <#1 Owning Many-to-Many Relationship>
(annotation) <#2 Subordinate Many-to-Many Relationship>

In Listing 7.9, the `Category` object's `items` variable is marked by the `@ManyToMany` annotation and is the owning side of the bidirectional association. In contrast, the `Item` object's `categories` variable signifies the subordinate bidirectional many-to-many association. As in the case of one-to-many relationships, the `@ManyToMany` annotation is missing the `optional` attribute. This is because an empty `Set` or `List` implicitly means an optional relation, meaning that the Entity can exist even if no associations do.

As a handy reference, we will summarize the various elements available in the @OneToOne, @OneToMany, @ManyToOne and @ManyToMany annotations in Table 7.3:

Table 7.3: Elements available in the @OneToOne, @OneToMany, @ManyToOne and @ManyToMany annotations

Element	**@OneToOne**	**@OneToMany**	**@ManyToOne**	**@ManyToMany**
TargetEntity	☑	☑	☑	☑
Cascade	☑	☑	☑	☑
Fetch	☑	☑	☑	☑
Optional	☑	☒	☑	☒
MappedBy	☑	☑	☒	☑

RIP - Container Managed Relations

If you have used EJB 2.x, you might be familiar with the Container-Managed-Relationship feature of Entity Beans with bidirectional relationships. This feature monitored changes on either side of the relationship and updated the other side automatically. CMR is not supported in this version because Entities can possibly be used outside of containers. However, mimicking this feature really is not too hard using a few extra lines of code. Let us take the User-BillingInfo one-to-one relation for example. The code for changing the BillingInfo object for a User and making sure both sides of the relationship are still accurate would look like the following:

```
user.setBilling(billing);
billing.setUser(user);
```

Believe it or not, we are now at the end of the Chapter and have finished discussing domain modeling using the JPA.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

7.4 Summary

In this chapter, we have discussed basic domain modeling concepts, Entities, relationships and how to define them using the JPA. The lightweight API makes creating rich, elegant object-oriented domain models a simple matter of applying annotations or deployment descriptor settings to plain Java Objects. The even greater departure from the heavyweight, framework code-laden approach of EJB 2.x is the fact that the new persistence API can be separated altogether from the container, as we will see in the coming chapters.

It is interesting to note that the API does not directly control relationship multiplicity. In the case of one-to-one and many-to-one relations, the `optional` annotation element somewhat specifies the multiplicity of the relationship. However, in the case of one-to-many and many-to-many relations, the API does not enforce multiplicity at all. In this case, it is the responsibility of the programmer to control the size of the collection holding Entity references (`java.util.Set` objects in our examples) and hence control multiplicity.

In Chapter 8 we will move onto the next step to building the ActionBazaar persistence layer and show you how to map the Entities and relationships we created to the database using the Object-relational mapping.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 8 Object-Relationship Mapping using EJB 3 JPA

In the previous chapter, we used EJB 3.0 JPA features to create a POJO domain model that supported a full range of OO features including inheritance. Namely, we identified Entities, Embedded Objects and the relationships between them using EJB 3.0 annotations. In this chapter, we learn how to persist our domain model into a relational database using Object-Relational mapping, which is the basis for JPA. In effect, Object-Relational mapping specifies how sets of Java Objects, including references between them are mapped to rows and columns in database tables. The first part of this chapter very briefly discusses the difference between Object-Oriented and relational world also known as "impedance mismatch". Later sections of the chapter explore the O-R (Object-Relational) mapping features of the EJB 3.0 JPA.

If you are a seasoned Enterprise developer, you are probably fairly comfortable with relational databases. If this is not the case then refer to Appendix XX for some primer on some relatively obscure relational database concepts like normalization and sequence columns that you must understand to get a clear understanding of the intricacies of O-R mapping.

We will warm up to the discussion by taking a look at the basic motivation behind O-R mapping, the so called "impedance mismatch." Then we will begin our analysis by mapping domain objects, move on to mapping relations and finally map inheritance using the inheritance strategies supported by JPA.

8.1 The Impedance Mismatch

The term impedance mismatch refers to the differences in the OO and relational paradigms and difficulties in application development that arises from these differences. The persistence layer where the domain model resides is where the impedance mismatch is usually the most apparent. The root of the problem lies in the differing fundamental objectives of both technologies.

Recall the fact that when a Java object holds a reference to another, the actual referred object is not copied over into the referring object. In other words, Java accesses Objects by reference and not by value. For example, two different `Item` Objects containing the same `category` instance variable value really point to the same `Category` Object in the JVM. This fact frees us from space efficiency concerns in implementing domain models with a high degree of conceptual abstraction. If this were not the case, we would probably store the identity of the referred `Category` Object (perhaps in an

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

`int` variable) inside the `Item` and materialize the link when necessary. This is in fact almost exactly what is done in the relational world.

The JVM also gives us the luxury of inheritance and polymorphism (by means that are very similar to the Object reference feature) that does not exist in the relational world. Lastly, as we mentioned in the previous chapter, a rich domain model object includes behavior (methods) in addition to attributes (data in instance variables). Databases tables on the other hand, inherently encapsulate only rows, columns and constraints, not business logic. These differences mean that the relational and OO model of the same conceptual problem look very different, especially for an appropriately normalized database created by an experienced DBA. Table 8.1 summarizes some of the overt mismatches between the object and relational worlds.

Table 8.1: The impedance mismatch: Obvious differences between the Object and Relational Worlds

OO Model (Java)	**Relational Model**
Object, Classes	Table, Rows
Attributes, Properties	Columns
Identity	Primary Key
Relationship/Reference to Other Entity	Foreign key
Inheritance/Polymorphism	Not supported
Methods	Indirect parallel to SQL logic, Stored Procedures
Code is portable	Not necessarily potable depending on vendor

1.16

In the following sections, we will crystallize the Object-Relational mismatch a little more by looking at a few corner cases while saving a persistence layer domain model into the database. We will discuss problems in mapping objects to database tables and provide a brief overview of Object-Relational Mapping. .

8.1.1 Mapping Objects to Databases

The most basic persistence layer for a Java application could consist of saving and retrieving domain objects using the JDBC API directly. To flush out the particularly rough spots in the object-relational mismatch, we will assume automated O-R mapping does not exist and we are following the direct JDBC route to persistence. We will see later that the EJB 3.0 Persistence API irons out these rough spots through simple configuration. Scott Ambler has written very good material that discusses the problem of mapping Objects to a relational database in much greater detail than we have the scope for. You can review the material at http://www.ambysoft.com/essays/mappingObjects.html.

One-To-One Mapping

As we discussed in the previous chapter, one-to-one relationship between Entities although are very rare in applications, makes a great deal of sense in the domain-modeling world. For example, The `User` and `BillingInfo` objects represent two logically separate concepts in the real world (we assumed) that are bound by a one-to-one relationship. Moreover, we also know that it does not make very much sense for a `BillingInfo` Object to exist without an associated `User`. The relation could be unidirectional from `User` to `BillingInfo`. The Figure 8.1 shows how this relationship might be materialized:

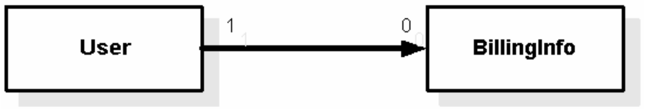


Figure 8.1: A unidirectional one-to-one relationship between User and BillingInfo.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The code in listing 8.1 implements this relationship:

24 Listing 8.1: One-to-One Relationship Between User and BillingInfo

```
public class User {
    protected String userId;
    protected String email;
    protected BillingInfo billing;                                        |#1
}

public class BillingInfo {
    protected String creditCardType;
    protected String creditCardNumber;
    protected String nameOnCreditCard;
    protected Date creditCardExpiration;
    protected String bankAccountNumber;
    protected String bankName;
    protected String routingNumber;
}
```

(annotation) <#1 Object Reference for One-to-One Relation>

From an OO perspective, it would make sense for the database tables storing this data to mirror the Java implementation in code listing 8.1. In this scheme, two different tables, `USERS` and `BILLING_INFO` would have to be created, with the `billing` object reference in the `User` object#1 being translated into a foreign key to the `BILLING_INFO` table's key in the `USERS` table (perhaps called BILLING_ID). The problem is that this scheme does not make complete sense in the relational world. As a matter of fact, since the objects are merely expressing a one-to-one relationship, normalization would dictate that the `USERS` and `BILLING_INFO` tables be merged into one. This would eliminate the almost pointless `BILLING_INFO` table and the redundant foreign key in the `USER` table. The extended `USER` table could look like the following:

```
USER_ID                 NOT NULL, PRIMARY KEY  NUMBER
EMAIL                   NOT NULL               VARCHAR2(255)
CREDIT_CARD_TYPE                               VARCHAR2(255)
CREDIT_CARD_NUMBER                             VARCHAR2(255)
NAME_ON_CREDIT_CARD                            VARCHAR2(255)
CREDIT_CARD_EXPIRATION                         DATE
BANK_ACCOUNT_NUMBER                            VARCHAR2(255)
BANK_NAME                                      VARCHAR2(255)
ROUTING_NUMBER                                 VARCHAR2(255)
```

In effect, our persistence layer mapping code would have to resolve this difference by pulling field data out of both the `User` and related `BillingInfo` tables and storing it into the columns of the combined `USERS` table. A bad approach, but an all too common one, would be to compromise your domain model to make it fit the relational data model (get rid of the separate `BillingInfo` Object). While this would certainly make the mapping code simper, you would lose out on a sensible domain model. In addition, you would write awkward code for the parts of your application that deal only with `the `BillingInfo` object and not the `User` object. If you remember our discussion in Chapter 7 then you probably realize that BillingInfo may make sense as an embedded object since you do not want to have a separate identity, and want to store the data in the USERS table.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

One-To-Many Relationships

The relational primary-key/foreign-key mechanism is ideally suited for a parent-child one-to-many relationship between tables. Let us take the probable relationship between the ITEMS and BIDS tables for example. The tables are likely to look like those shown in Listing 8.2:

25

26 **Listing 8.2: One-to-Many Relationship Between ITEMS and BIDS Tables**

```
ITEMS TABLE
ITEM_ID          NOT NULL, PRIMARY KEY    NUMBER
TITLE            NOT NULL                 VARCHAR2(255)
DESCRIPTION      NOT NULL                 CLOB
INITIAL_PRICE    NOT NULL                 NUMBER
BID_START_DATE   NOT NULL                 TIMESTAMP
BID_END_DATE     NOT NULL                 TIMESTAMP
ITEM_SELLER_ID   NOT NULL,                NUMBER
                 FOREIGN KEY (USERS(USER_ID))

BIDS TABLE
BID_ID           NOT NULL, PRIMARY KEY    NUMBER
AMOUNT           NOT NULL                 NUMBER
BID_DATE         NOT NULL                 TIMESTAMP
BID_BIDDER_ID    NOT NULL,                NUMBER
                 FOREIGN KEY (USER(USER_ID))
BID_ITEM_ID      NOT NULL,                NUMBER                         |#1
                 FOREIGN KEY (ITEMS(ITEM_ID))
```

(annotation) <#1 Foreign Key Signifying One-To-Many Relation>

The `ITEM_ID` foreign key into the `ITEMS` table from the `BID` table means that multiple `BIDS` table rows can refer to the same record in `ITEMS` table. This implements a many-to-one relation going from the `BIDS` table to the `ITEMS` table and it is very simple to retrieve an item given a bid record. On the other hand, retrieval from ITEMS to BIDS will require a little more effort in looking for BIDS rows that match a given ITEM_ID key. As we mentioned in the previous chapter, however, the relationship between the `Item` and `Bid` domain objects are one-many *bidirectional*. This means that the `Item` Object has a reference to a set of `Bid` Objects while the `Bid` object holds a reference to an `Item` Object. As a Java developer, you might have expected the `ITEMS` table to contain some kind of reference to the `BIDS` table in addition to the `ITEM_ID` foreign key in the `BIDS` table. The problem is that such a table structure simply does not make sense in the relational world. Instead, our O-R mapping layer must translate the parent-child unidirectional database relation into a bidirectional one-to-many relation in the OO world by using a lookup scheme instead of simple, directional references.

Many-To-Many

Many-to-many relations are fairly common in enterprise development. In our ActionBazaar domain model presented in Chapter 7, the relationship between the `Item` and `Category` domain objects is many-to-many. That is, an item can belong in multiple categories while a category can contain more than one item. This is fairly easy to implement in the OO world with a set of references on either side of the relationship. In the database world on the other hand, the only way to implement a relation is through a foreign key, which is inherently one-to-many. As a result, the only real way to implement many-to-many relations is by breaking them down into two one-to-many

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

relations. Let us see how this works by taking a look at the database table representation of the item-category relation in Listing 8.3.:

Listing 8.3: Many-to-Many Relationship Between ITEMS and CATEGORIES Tables

```
ITEMS TABLE
ITEM_ID          NOT NULL, PRIMARY KEY    NUMBER                         |#1
TITLE            NOT NULL                 VARCHAR2(255)
...

CATEGORIES TABLE
CATEGORY_ID        NOT NULL, PRIMARY KEY    NUMBER                       |#2
NAME               NOT NULL                 VARCHAR2(255)
...

CATEGORY ITEMS TABLE                                                     |#3
ITEM_ID          NOT NULL, PRIMARY KEY    NUMBER                         |#4
                 FOREIGN KEY(ITEMS(ITEM_ID))
CATEGORY_ID      NOT NULL, PRIMARY KEY    NUMBER                         |#5
                 FOREIGN KEY(CATEGORIES(CATEGORY_ID))
```

(annotation) <#1 ItemsT able Primary Key>
(annotation) <#2 Category Table Primary Key>
(annotation) <#3 Association Table Implementing Many-to-Many Relation>
(annotation) <#4 Items Table Foreign Key>
(annotation) <#5 Category Table Foreign Key>

The `CATEGORY_ITEMS` table is called an *association* or *intersection table* and accomplishes a pretty neat trick. The only two columns it contains are foreign key references to the `ITEMS` and `CATEGORIES` tables (ironically the two foreign keys combined are the primary key for the table). In effect, it makes it possible to match up arbitrary rows of the two related tables, making it possible to implement many-to-many relations. Since neither related table contains a foreign key, relationship direction is completely irrelevant. To get to the records on the other side of the relation from either side, we must perform a join in the O-R mapping layer involving the association layer. For example, to get all the items under a category, we must retrieve the `CATEGORY_ID`, join the `CATEGORY_ITEMS` table with the `ITEMS` table and retrieve all item data for rows that match the `CATEGORY_ID` foreign key. Saving the relation into the database would involve saving rows into the `CATEGORIES` and `ITEM` tables as well as the `CATEGORY_ITEMS` table. Clearly, the many-to-many relationships are modeled very differently in the relational world than they are in the OO world.

Inheritance

Unlike the three previous cases, one-to-one, one-to-many, and many-to-many, inheritance is probably the most severe case of the object-relational mismatch. Inheritance therefore calls for solutions that are not elegant fits to relational theory at all,.The OO concept of inheritance has no direct equivalent in the relational world. However, there are a few creative ways that O-R solutions bridge this gap, including: storing each object in the inheritance hierarchy in completely separated tables, mapping all classes into a single table, or storing superclass/subclasses in related tables. Because none of these strategies is simplistic, we will defer a detailed discussion to the O-R mapping section later in the chapter.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

8.1.2 Introducing O/R Mapping

In the most general sense, the term *Object-Relational Mapping* means any process that can save an Object (in our case a Java Object) into a relational database. As we mentioned, for all intents and purposes you could write home-brewed JDBC code to do that. In the realm of automated persistence, O-R Mapping really means using primarily configuration metadata to tell an extremely high level API which tables a set of Java Objects are going to be saved into. It is not all that far from the truth that this involves the promisingly simple act of figuring out what table row an Object instance should be saved into and what field/property data belongs in what column. In EJB 3.0, the configuration metadata obviously consists either of annotations or deployment descriptor elements. As our impedance mismatch discussion points out, there are a few wrinkles in the idealistic view of automated persistence. Because of the inherent complexity of the problem, EJB 3.0 cannot make the solution absolutely effortless, but it goes a long way in making it less painful. In the next section, we start our discussion of EJB 3.0 O-R Mapping by covering the simplistic case of saving an Entity without regards to domain relations or inheritance.

O-R Mapping Portability s in EJB 2.x

One of the greatest weaknesses of EJB 2.x CMP (Container Managed Persistence) Entity beans was that it never standardized the process of Object-Relational Mapping. Instead, mapping strategies were left up to the individual vendors, whose approaches varied widely. As a result, porting Entity beans from one application server to another more or less meant redoing O-R mapping all over again. This meant that the portability that EJB 2.x promised meant little more than empty words.

EJB 3.0 firmly standardizes O-R mapping and gets us much closer to then elusive goal of portability. As a matter of fact, this is likely to be very accomplishable as long as you are careful to steer clear of application server specific features.

Other than smoothing out the impedance problems by applying generalized strategies behind the scenes, there are a few other benefits to using O-R. Even disregarding the edge cases discussed above, if you have spent any time writing application persistence layers using JDBC, you know that substantial work is required. This is largely because of the repetitive "plumbing" code of JDBC and the large volume of complicated hand-written SQL involved. As we will soon see, using O-R Mapping frees us from this burden and the task of persistence largely becomes an exercise in simple configuration. The fact that the EJB3 Persistence provider generates JDBC and SQL code on your behalf has another very nice effect. Because the persistence provider is capable of automatically generating code optimized to your database platform from your database-neutral configuration data, switching databases becomes a snap. Accomplishing the same using hand-written SQL is careful, tedious work at best and impossible at worst. Database portability is one of the most appealing features of the EJB 3.0 Java Persistence API that fits nicely with the Java philosophy but has been elusive for some time.

Now that we've looked at the reasons for O/R mapping, let's see how EJB 3.0 implements it.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

8.2 Mapping Entities

This section explores some of the fundamental features of EJB 3.0 O-R mapping by taking a look at the implementation of the ActionBazaar `User` Entity. You will learn use of several O-R mapping annotations such as @Table, @Column, @Enumerated, @Lob, @Temporal and @Embeddable.

If you remember from our discussion in Chapter 7, `User` is the superclass to both the `Seller` and `Bidder` domain objects. To keep this example simple we will ignore the inheritance and use a peristence field to identify the user type in the table USERS. The `User` Entity contains fields that are common to all user types in ActionBazaar such as the user ID, username (used for login and authentication), first name, last name, user type (bidder, seller and admin, etc), user-uploaded picture and user account creation date. All fields are mapped and persisted into the database for the `User` Entity in code listing 8.4. We have used field-based persistence for the Entity to keep the code sample short.

Listing 8.4: Mapping an Entity

```
@Entity                                                                  |#1
@Table(name="USERS")                                                      |#2
@SecondaryTable(name="USER_PICTURE",                                     |#3
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User implements Serializable {

    @Id                                                                  |#4
    @Column(name="USER_ID", nullable=false)                              |#5
    protected Long userId;

    @Column(name="USER_NAME", nullable=false)                            |#5
    protected String username;

    @Column(name="FIRST_NAME", nullable=false, length=1)                 |#5
    protected String firstName;

    @Column(name="LAST_NAME", nullable=false)                            |#5
    protected String lastName;

    @Enumerated(EnumType.ORDINAL)                                        |#6
    @Column(name="USER_TYPE", nullable=false)
    protected UserType userType;

    @Column(name="PICTURE", table="USER_PICTURE")
    @Lob                                                                 |#7
    @Basic(fetch=FetchType.LAZY)                                         |#8
    protected byte[] picture;

    @Column(name="CREATION_DATE", nullable=false)
    @Temporal(TemporalType.DATE)                                         |#9
    protected Date creationDate;

    @Embedded                                                           |#10
    protected Address address;

    public User() {}                                                    |#11
}

@Embeddable                                                             |#12
public class Address implements Serializable {
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @Column(name="STREET", nullable=false)
    protected String street;

    @Column(name="CITY", nullable=false)
    protected String city;

    @Column(name="STATE", nullable=false)
    protected String state;

    @Column(name="ZIP_CODE", nullable=false)
    protected String zipCode;

    @Column(name="COUNTRY", nullable=false)
    protected String country;
}
```

(annotation) <#1 Entity Reference>
(annotation) <#2 Table Mapping>
(annotation) <#3 A Join Table>
(annotation) <#4 Id field>
(annotation) <#5 Field Column Mappings>
(annotation) <#6 Enumerated Column>
(annotation) <#7 Blob Field>
(annotation) <#8 Lazy Loading>
(annotation) <#9 Temporal Field>
(annotation) <#10 Embedded field>
(annotation) <#11 Default Constructor>
(annotation) <#12 Embeddable Class>

Briefly scanning Listing 8.4, we see that the `User` Entity is mapped to the `USERS` table#2 joined with the `USER_PICTURE` table using the `USER_ID` primary key#3. Each of the fields is mapped to a database column using the `@Column` annotation#5. We deliberately made the Listing feature-rich and quite a few interesting things are going on with the columns. The `userType` field is restricted to be an ordinal enumeration#6. The `picture` field is marked as a binary large object (BLOB)#7 that is lazily loaded#8. The `creationDate` field is marked as a temporal date#9. All in all, the `@Table`, `@SecondaryTable`, `@Column`, `@Enumerated`, `@Lob`, `@Basic`, `@Temporal`, `@Embedded` and `@Embeddable` O-R mapping annotations are used. We will start our analysis of O-R with the `@Table` annotation.

Annotations vs. XML in O-R Mapping

The difficulty of choosing between annotations and XML deployment descriptors manifests itself most strikingly in the arena of EJB 3.0 O-R Mapping. XML descriptors are verbose, hard to manage and most developers find them to be a sour-point for Java EE. While O-R mapping with Annotations make life simpler you should keep in mind that you are hard-coding your database schema in your code similar to using JDBC. This means that the slightest schema change will result in a recompilation and redeployment cycle as opposed to simple configuration. If you have a stable database design that rarely changes or you are comfortable using JDBC DAO then there is no issue here. But if you have an environment where database schema is less stable (subject to change more often) you would probably be better off using descriptors. Luckily, you can use XML descriptors to override O-R mapping annotations after deploying to a production environment. As a result, changing your mind in response to the reality on the ground many not be a very big deal.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

DP: Yes, similar8.2.1 Specifying the table

@Table specifies the table containing the columns to which the Entity is mapped. In code listing 8.4, the @Table annotation makes the USERS table's columns available for O-R Mapping. In fact, by default, all the persistent data for the Entity is mapped to the table specified by the annotation's name parameter. As you can see from the annotation's definition below, it contains a few other parameters:

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```

If the name parameter is omitted, the table name is assumed to be the same as the name of the Entity. Note, in our case, this would have been just fine, if we are mapping to the USERS table. We limit discussion on the catalog and schema parameters since they are hardly ever used. In affect, they allow you to fully qualify the mapped table[9]. For example, we could have explicitly specified that the USERS table belongs in the ACTIONBAZAAR schema like so:

```
@Table(name="USERS", schema="ACTIONBAZAAR")
public class User
```

By default, it is assumed that the table belongs in the schema of the data source used. We will learn how to specify a data source for a persistence module in Chapter 11 when we discuss Entity packaging. The uniqueConstraints parameter really is not used that often either. It specifies unique constraints on table columns and is only used when table auto-creation is enabled. The following is an example:

```
@Table(name="CATEGORIES",
       uniqueConstraints=
           {@UniqueConstraint(columnNames={"CATEGORY_ID"})})
```

If it did not exist and auto-generation is enabled, the code puts a unique constraint on the CATEGORY_ID column of the CATEGORIES table when it is created during deployment time. The uniqueConstraints parameter supports specifying constraints on more than one column. It is important to keep mind, however, that EJB 3.0 implementations are not mandated to support generation of tables and it is a bad idea to use automatic table generation beyond simple development databases. The @Table annotation itself is optional. If omitted, the Entity is assumed to be mapped to a table in the default schema with the same name as the Entity Class. Most Entities will typically be mapped to a single table. The User Object happens to be mapped to two tables, as you might

[9] We already discussed what a schema is. For all intensive purposes, you can think of a *catalog* to be a "meta-schema" or a higher-level abstraction for organizing schemas. Often, a database will only have one common system catalog.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

have guessed from the `@SecondaryTable` annotation used in listing 8.4#3. We will comer back to this later after we take a look at mapping Entity data using the `@Column` annotation.

8.2.2 Mapping the columns

The `@Column` annotation maps a persisted field or property to a table column. All of the fields used in listing 8.3 are annotated with `@Column`. For example, the `userId` field is mapped to the `USER_ID` column:

```
@Column(name="USER_ID")
protected Long userId;
```

It is assumed that the `USER_ID` column belongs to the `USERS` table specified by the `@Table` annotation. Most often, this is as simple as your `@Column` annotation will look. At best, you might need to explicitly specify which table the persisted column belongs to as we do for the `picture` field in listing 8.4:

```
@Column(name="PICTURE", table="USER_PICTURE")
...
protected byte[] picture;
```

As you can see from the definition in Listing 8.5, a number of other parameters exist for the annotation. The `insertable#3` and `updatable#4` parameters are used to control persistence behavior.

Listing 8.5: The @Column Annotation

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;                                    |#1
    boolean nullable() default true;                                   |#2
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;                                          |#3
    int precision() default 0;                                         |#4
    int scale() default 0;                                             |#8
}
```

(annotation) <#1 Specifies Unique Constraint>
(annotation) <#2 Specifies if Column Allows Nulls>
(annotation) <#3 Length of Column>
(annotation) <#4 Decimal Precision of Column>
(annotation) <#5 Decimal Scale of Column>

If the `insertable` parameter is set to false, the field or property will not be included in the `INSERT` statement generated by the Persistence provider to create a new record corresponding to the Entity. Likewise, setting the `updatable` parameter to false excludes the field or property from being updated when the Entity is saved. These two parameters are usually helpful in dealing with read-only data, like primary keys generated by the database. They could be applied to the userId field as follows:

```
@Column(name="USER_ID", insertable=false, updatable=false)
protected Long userId;
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

When a User Entity is first created in the database, the Persistence provider does not include the USER_ID as part of the generated INSERT statement. Instead, we could be populating the USER_ID column through an INSERT-induced trigger on the database server side. Similarly, since it does not make very much sense to update a generated key, it is not included in the UPDATE statement for the Entity either. The rest of the parameters of the @Column annotation are only used for automatic table generation and specify column creation data. The nullable parameter specifies if the column supports null values#2, the unique parameter#1 indicates if the column has a unique constraint, the length parameter#3 specifies the size of the database column, the precision parameter#4 specifies the precision of a decimal field, the scale parameter#5 specifies the scale of a decimal column. Finally, the columnDefinition parameter allows you to specify the exact SQL to create the column. We will not cover these parameters much further than this basic information since we do not encourage automatic table creation. Note the @Column annotation is optional. If omitted, a persistent field or property is saved to the table column matching the field or property name. For example, a property specified by the getName and setName methods will be saved into the NAME column of the table for the Entity. Next, we will take a look at a few more annotations applied to Entity data, starting with @Enumerated.

8.2.3 Using @Enumerated

Languages like C and Pascal have had enumerated data types for decades. Enumerations were finally introduced in Java 5.0. In case you are unfamiliar with them, we will start with the basics. In listing 8.4, the user type field has a type of UserType. UserType is a Java enumeration that is defined as follows:

```
public enum UserType {SELLER, BIDDER, CSR, ADMIN};
```

This effectively means that any data type defined as UserType (like our persistent field in the User Object) can only have the four values listed in the enumeration. Like an array, each element of the enumeration is associated with an index called the *ordinal*. For example the UserType.SELLER value has an ordinal of 0, the UserType.BIDDER value has an ordinal of 1 and so on. The problem is determining how to store the value of enumerated data into the column. The Java Persistence API supports two options through the @Enumerated annotation. In our case, we specify that the ordinal value should be saved into the database:

```
@Enumerated(EnumType.ORDINAL)
...
protected UserType userType;
```

This means that if the value of the field is set to UserType.SELLER, the value 0 will be stored into the database. Alternatively, you can specify that the Enumeration value name should be stored as a String:

```
@Enumerated(EnumType.STRING)
...
protected UserType userType;
```

In this case a UserType.ADMIN value would be saved into the database as "ADMIN". By default an Enumerated field or property is saved as an ordinal. This would be the case if the @Enumerated annotation is omitted altogether or no parameter to the annotation is specified.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

8.2.4 *Mapping CLOBs and BLOBs*

An extremely powerful feature of relational databases is the ability to store very large data as BLOB (Binary Large Object) and CLOB (Character Large Object) types. These correspond to the JDBC `javax.sql.Blob` and `javax.sql.Clob` Objects. The `@Lob` annotation designates a property of field as a CLOB or BLOB. For example we designate the picture field as a BLOB in listing 8.4:

```
@Lob
@Basic(fetch=FetchType.LAZY)
protected byte[] picture;
```

Whether a field or property designated `@Lob` is a CLOB or BLOB is determined by its type. If the data is of type `char[]` or `String`, the persistence provider maps the data to a `CLOB` column. Otherwise, the column is mapped as a `BLOB`. An extremely useful annotation to use in conjunction with `@Lob` is `@Basic`.@Basic can be marked on any attribute with direct-to-field mapping. Just as we have done for the `picture` field, the `@Basic(fetch=FetchType.LAZY)` specification causes the BLOB or CLOB data to be loaded from the database only when it is first accessed. This is a great feature since LOB data is usually very memory intensive and should only be loaded if needed. Unfortunately lazy loading of LOB types is left as optional for vendors by the EJB 3.0 specification and there is no guarantee that the column will actually be lazily loaded.

8.2.5 *Mapping temporal types*

Most databases support a few different temporal data types with different granularity levels corresponding to DATE (storing day, month and year), TIME (storing just time and not day, month or year) and TIMESTAMP (storing time, day, month and year). The `@Temporal` annotation specifies which of these data types we want to map a `java.util.Date` or `java.util.Calendar` persistent data type to. In listing 8.3, we save the `creationDate` field into the database as a DATE:

```
@Temporal(TemporalType.DATE)
protected Date creationDate;
```

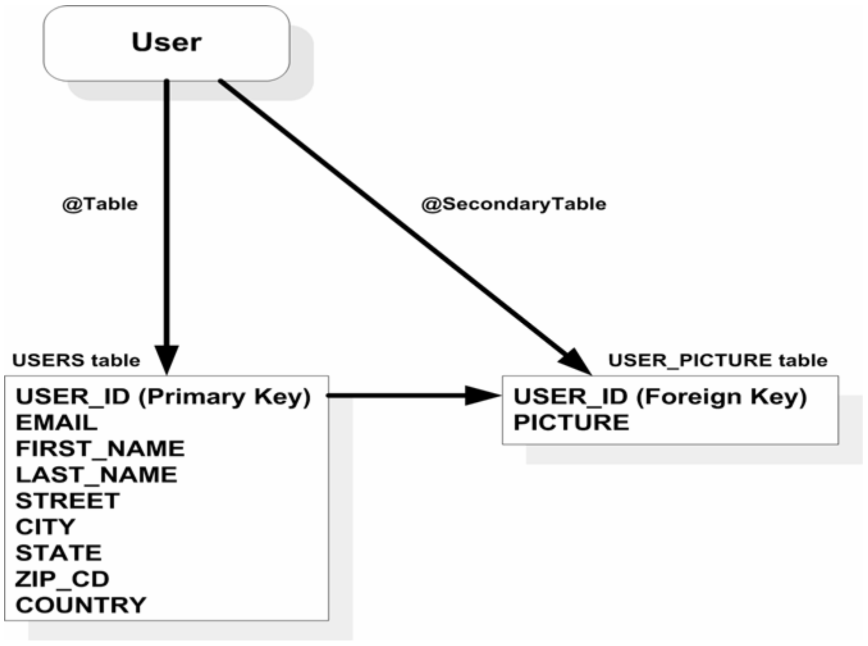
Note this explicit mapping is redundant while using the `java.sql.Date, java.sql.Time` or `java.sql.Timestamp` Java types as opposed to `java.util.Date` or `java.util.Calendar`. If we do not specify a parameter for `@Temporal` annotation or omit it altogether the Persistence provider will assume the data type mapping to be TIMESTAMP (the smallest possible data granularity).

8.2.6 *Mapping an entity to multiple tables*

This is not often the case for non-legacy databases, but sometimes an Entity's data must come from two different tables. In fact, in some rare situations, this is a very sensible strategy. For example, the `User` Entity in Listing 8.4 is stored across the `USERS` and `USER_PICTURE` tables as depicted in Figure 8.2.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



49 **Figure 8.2: An entity can be mapped to more than one table e.g. User entity spans more than one table i.e USERS and USER_PICTURE. The primary table is mapped using @Table and secondary table is mapped using @Secondary table. The primary and secondary tables must have the same primary key.**

This makes excellent sense because the `USER_PICTURE` table stores large binary images that could significantly slow down queries using the table but is rarely used. Isolating the binary images into a separate table in conjunction with the *lazy loading* technique discussed in section 8.2.4 to deal with the `picture` field mapped to the `USER_PICTURE` table can result in a significant boost in application performance. The `@SecondaryTable` annotation enables us to derive Entity data from more than one table and is defined as follows:

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Notice that other than the `pkJoinColumns` element, the definition of the annotation is identical to the definition of the `@Table` annotation. This element is the key to how annotation works. To see what we mean, examine the following code implementing the `User` Entity mapped to two tables:

```
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURE",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User implements Serializable {
..}
```

Obviously the two tables in `@Table` and `@SecondaryTable` are related somehow and are joined to materialize the Entity. This kind of relationship is implemented by creating a foreign key in

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

the secondary table referencing the primary key in the first table. In this case, the foreign key also happens to be the primary key of the secondary table. To be precise, `USER_ID` is the primary key of `USER_PICTURE` table and it references the primary key of the `USERS` table. The `pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID")` specification assumes exactly this relation. The `name` element points to the `USER_ID` foreign key in the `USER_PICTURE` secondary table. The persistence provider is left to figure out what the primary key of the USERS table is, which also happens to be named `USER_ID`. Using the detected primary key, the provider performs a join when necessary in order to materialize the data for the `User` Entity. In the extremely unlikely case that an Entity consists of tables from more than two columns, we may use the `@SecondaryTables` annotation more than once for the same Entity. We will not cover this case here but encourage you to explore it if needed.

Before we conclude the section on mapping Entities and talk about mapping EJB 3.0 relations, we are going to discuss very vital feature of JPA, primary key generation.

8.2.7 Generating primary keys

When we identify a column or set of columns as *primary key*, we essentially ask the database to enforce uniqueness. Primary keys that consist of business data are called *natural keys*. The classic example of this is `SSN` as the primary key for an `EMPLOYEE` table. `CATEGORY_ID` or `EMPLOYEE_ID`, on the other hand, are examples of *surrogate keys*. Essentially, surrogate keys are columns created explicitly to function as primary keys. Surrogate keys are popular and we highly recommend them, especially over compound keys.

There are three popular ways of generating primary key values: identities, sequences and tables. Fortunately, all three strategies are supported via the `@GeneratedValue` annotation and switching is as simple as changing configuration. We will start our analysis with the simplest case, using identities.

Identity Columns as Generators

Many databases such as Microsoft SQLServer support identity column and we can use an identity constraint to manage the primary key for the `User` entity as follows:

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="USER_ID")
protected Long userId;
```

The code above assumes that an identity constraint exists on the `USERS.USER_ID` column. Note that while the value for the identity field may not be available available before the entity data is saved in the database while using IDENTITY as generator type because the typically it is generated while a record is committed.

The two other strategies, SEQUENCE and TABLE, both require the use of an externally defined generator. Namely, a `SequenceGenerator` or `TableGenerator` must be created and set for the `GeneratedValue`. We will see how this works by first taking a look at the sequence generation strategy.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Database squences as Generator

In order to use sequence generators, we must define a sequence in the database first. The following is a sample sequence for the `USER_ID` column in an Oracle database:

```
CREATE SEQUENCE user_sequence START WITH 1 INCREMENT BY 10;
```

We are now ready to create a sequence generator in EJB 3.0:

```
@SequenceGenerator(name="USER_SEQUENCE_GENERATOR",
    sequenceName="USER_SEQUENCE",
            initialValue=1, allocationSize=10)
```

The `@SequenceGenerator` annotation creates a sequence generator named `USER_SEQUENCE_GENERATOR` referencing the Oracle sequence we created and matching its setup. Naming the sequence is critical since it is referred to by the `@GeneratedValue` annotation. The `initialValue` element is pretty self-explanatory: `allocationSize` specifies how much the sequence is incremented by each time a value is generated. The default values for `initialValue` and `allocationSize` are 0 and 50 respectively. It's handy that the sequence generator need not be created in the same Entity that it is used. As a matter of fact, any generator is shared among all Entities in the persistence module and hence each generator must be uniquely named in a persistence module. Finally, we can reimplement the generated key for the `USER_ID` column as follows:

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
    generator="USER_SEQUENCE_GENERATOR")
@Column(name="USER_ID")
protected Long userId;
```

Sequence Tables as Generators

Using table generators is just as simple as with a sequence generator, the first step is to create a table to use for generating values. The table used must follow a general format like the following one created for Oracle:

```
CREATE TABLE sequence_generator_table
    (sequence_name VARCHAR2(80) NOT NULL,
     sequence_value NUMBER(15) NOT NULL,
     PRIMARY KEY (sequence_name));
```

The `sequence_name` column is meant to store the name of a sequence while the `sequence_value` column is meant to store the current value of the sequence. The next step is to prepare the table for use by inserting the initial value manually as follows:

```
INSERT INTO
    sequence_generator_table (sequence_name, sequence_value)
VALUES ('USER_SEQUENCE', 1);
```

In a sense, these two steps combined are the equivalent to creating the Oracle sequence in the second strategy. Despite the obvious complexity of this approach, one upside is that the same

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

sequence table can be used for multiple sequences in the application. We are now prepared to create the `TableGenerator` utilizing the table:

```
@TableGenerator (name="USER_TABLE_GENERATOR",
    table="SEQUENCE_GENERATOR_TABLE",
    pkColumnName="SEQUENCE_NAME",
    valueColumnName="SEQUENCE_VALUE",
    pkColumnValue="USER_SEQUENCE")
```

If you need to, you can specify the values for `initialValue` and `allocationSize` as well. Finally, we can use the table generator for `USER_ID` key generation:

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
    generator="USER_TABLE_GENERATOR")
@Column(name="USER_ID")
protected Long userId;
```

Default Primary key generation strategy

The last option for key generation is to let the PROVIDER decide the best strategy for the underlying database by using the AUTO specification as follows:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="USER_ID")
protected Long userId;
```

This is a perfect match for automatic table creation because the underlying database objects required will be created by the JPA provider.

Probably you would assume that if Oracle were the underlying database, the persistence provider probably would choose SEQUENCE as the strategy; if SQL Server were the underlying database, IDENTITY would probably be chosen on your behalf. However this may not be true and we recommend that you check documentation of your persistence provider. For example TopLink Essentials uses table generator as the default auto generator for all databases.

Also you should note that although generated values are often used for surrogate keys, you could use the feature for any persistence field or property. Before we move onto discussing Entity relations, we will tackle the most complicated case of mapping basic Entity data next—mapping Embeddable Objects.

Standardization of Key Generation

Just as EJB 3.0 standardizes O-R Mapping, it standardizes key generation as well. This is a large leap from EJB 2.x, where you had to resort to various Sequence Generator patterns to accomplish the same for CMP Entity Beans. Using sequences, identity constraints or tables was a significant effort, a far cry from simple configuration, not to mention non-portable.

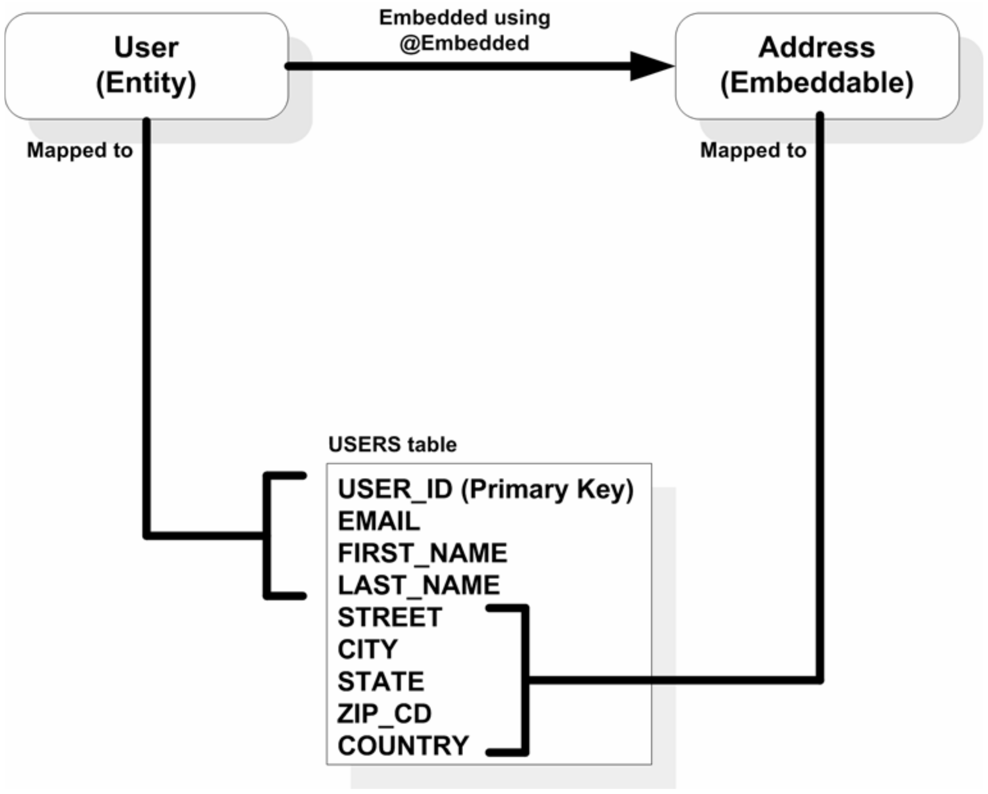
8.2.8 Mapping Embeddable Classes

When discussing Embeddable Objects in Chapter 7, we mentioned that an Embeddable object acts primarily as a convenient data holder for Entities and has no identity of its own. Rather, it shares

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

the identity of the Entity Class it belongs to. We will now discuss how Embeddable Objects are mapped to the database.



50 **Figure 8.3: Embeddable object act as convenient dataholder for entities and have no identity of its own. Address is an embeddable object that is stored as a part of User entity that is mapped to the USERS table.**

In Listing 8.4, we include the `Address` Embedded Object introduced in Chapter 7 in the `User` Entity as a data field. The relevant parts of Listing 8.4 are repeated in Listing 8.6 for easy reference.

29 Listing 8.6: Using Embeddable Objects

```
@Table(name="USERS")                                           |#1
...
public class User implements Serializable {

    @Id                                                        |#2
    @Column(name="USER_ID", nullable=false)
    protected Long userId;
    ...
    @Embedded                                                  |#3
    protected Address address;
    ...
}
...
@Embeddable                                                    |#4
public class Address implements Serializable {
    @Column(name="STREET", nullable=false)                     |#5
    protected String street;
    ...
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @Column(name="ZIP_CODE", nullable=false)                         |#5
    protected String zipCode;
    ...
}
```
(annotation) <#1 The table storing both Entity and Embeddable Object>
(annotation) <#2 Shared identity>
(annotation) <#3 Embeded field>
(annotation) <#4 Embeddable Object>
(annotation) <#5 Embeddable Object field mappings>

The first interesting feature of Listing 8.6 you should notice is that unlike the `User` Entity, the Embeddable `Address` Object is missing a `@Table` annotation. This is because EJB 3.0 does not allow Embedded Objects to be mapped to a table different from the enclosing Entity, at least not directly through the `@Table` annotation. Instead, the `Address` Object's data is stored in the `USERS` table that stores the enclosing Entity#1. Hence, the `@Column` mappings applied to the fields of the `Address` Object#5 really refer to the columns in the `USER` table. For example, the `street` field is mapped to the `USERS.STREET` column, the `zipCode` field is mapped to the `USERS.ZIP_CODE` column and so on. This also means that the `Address` data is stored in the same database row as the `User` data and both objects share the `USER_ID` identity column#2. Other than this minor nuance, all data mapping annotations used in Entities are available for you in Embedded Objects and behave in exactly the same manner. In general, this is the norm and Embedded Objects are often stored in the same table as the enclosing Entity. However, if this does not suit you, it is possible to store the Embeddable Object's data in a separate table and use the `@SecondaryTable` annotation to retrieve its data into the main Entity using a join. We will not detail this solution, as it is fairly easy to figure out and will leave it for you to experiment with instead.

Sharing Embeddable Classes Between Entities

One of the most useful features of Embeddable Classes is that they can be shared between Entities. Using our Address Object example, the Address Object could be embedded inside a BillingInfo object to store billing addresses while still being used by the User Entity. The important nuance to keep in mind is that the Embeddable Class definition is shared in the OO world and *not* the actual data in the underlying relational tables. As we noted, the Embedded data is materialized using the table mapping of the Entity Class. However, this means that all the Embeddable data must be mapped to both the USERS and BILLING_INFO tables. In other words, both tables must contain some mappable street, city, zip, etc columns.
An interesting wrinkle to consider is the fact that the same embedded data could be mapped to columns with different names in two separate tables. For example, the 'state' data column in BILLING_INFO could be called STATE_CODE instead of STATE. Since the Column annotation in Address maps to a column named STATE, how will this column be resolved? The solution to the answer is overriding the column mapping in the enclosing Entity using the AttributeOverride annotation as follows:
```
@Embedded
@AttributeOverrides({@AttributeOverride(
    name="state",
    column=@Column(name="STATE_CODE"))})
protected Address address;
```
In effect, the AttributeOverride annotation is telling the providerto resolve the Embedded field "state" to the STATE_CODE table for the enclosing Entity.

Licensed to John Sweitzer <admin@saolailaem.info>

We have now finished looking at all the annotations required for mapping Entities except for mapping OO Inheritance. We will move onto looking at EJB 3.0 features for mapping Entity relations next.

8.3 Mapping Entity Relations

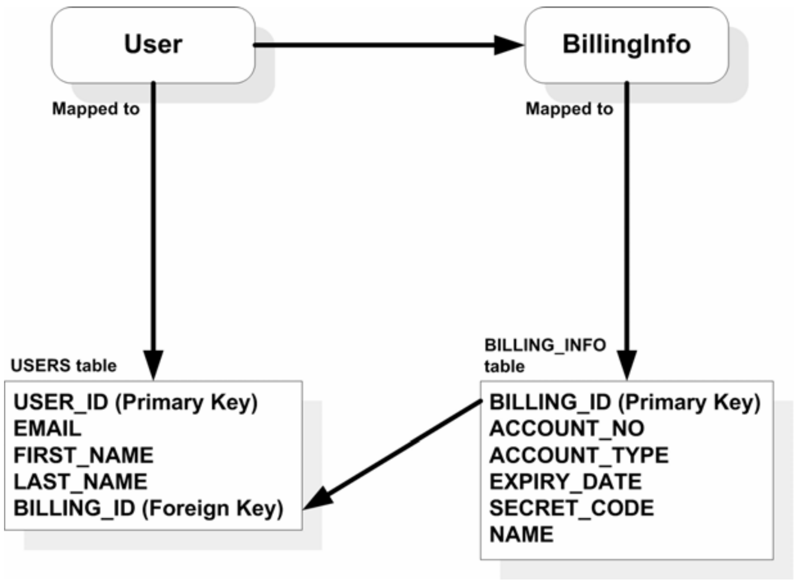
In the last chapter we explored implementing domain relations between entities. In section 8.1.1 we briefly discussed the problems translating relations from the OO world to the database world. In this section, we will see how these solutions are actually implemented in EJB 3.0 using annotations, starting with one-one relations. You will learn mapping of all types of relations one-to-one, one-to-many and many-to-one and many-to-many.

8.3.1 Mapping One-to-One Relations

As we know, one-to-one relations are mapped using primary/foreign key associations. It should be pretty obvious that a parent-child relation usually exists between the Entities of a one-to-one relationship. For example, in the `User-BillingInfo` relationship mentioned earlier, the `User` Entity could be characterized as the parent. Depending on where the foreign key resides, the relationship could be implemented in two different ways: either using the `@JoinColumn` or the `@PrimaryKeyJoinColumn` annotation.

Using @JoinColumn

If the underlying table for the referencing Entity is the one containing the foreign key to the table to which referenced "child" Entity is mapped to, we may map the relation using the `@JoinColumn` annotation.



51 **Figure 8.4: User has a one-one unidirectional relationship with BillingInfo. User and BillingInfo entities are mapped to USERS and BILLING_INFO tables respectively and USERS table has a foreign key reference to BILLING_INFO table. Such associations are mapped using @JoinColumn.**

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

In our User-BillingInfo example as depicted in figure 8.4, the USERS table contains a foreign key named USER_BILLING_ID that refers to the BILLING_INFO table's BILLING_ID primary key. This relation would be mapped shown in listing 8.7.

30 Listing 8.7: Mapping OneToOne Uni-directional Relationship Using @JoinColumn

```
@Entity
@Table(name="USERS")
public class User {
    @Id
    @Column(name="USER_ID")
    protected String userId;
    ...
    @OneToOne
    @JoinColumn(name="USER_BILLING_ID",                                      |#1
        referencedColumnName="BILLING_ID", updatable=false)                  |#1
    protected BillingInfo billingInfo;
}

@Entity
@Table(name="BILLING_INFO")
public class BillingInfo {
    @Id
    @Column(name="BILLING_ID")                                               |#2
    protected Long billingId;
    ...
}
```

(annotation) <#1 Parent Foreign Key Manifestation>
(annotation) <#2 Child Table Primary Key>

The @JoinColumn annotation's name element refers to the name of the foreign key in the USERS table#1. If this parameter is omitted, it is assumed to follow this form:

<relationship field/property name>_<name of referenced primary key column>

In our example, the foreign key name would be assumed to be BILLINGINFO_BILLING_ID in the USER table. The referencedColumnName element specifies the name of the primary key or a unique key the foreign key refers to. If we do not specify the referencedColumnName value it is assumed to be the column containing the identity of the referenced entity. Incidentally, this would have been fine in our case as BILLING_ID is the primary key for BILLING_INFO table.

Like the @Column annotation, the @JoinColumn annotation contains the updatable, insertable, table, unique elements. The elements serve the same purposes as the elements of the @Column annotation. In our case, updatable is set to false which means that the persistence provider would not update the foreign key even if the billingInfo reference were changed. If we have more than one column in the foreign key we can use JoinColumns annotation instead. We will not cover this annotation since this situation is very unlikely if not bad design.

If you have a bi-directional OneToOne relationship then the entity in inverse side of the relationship will contain the mappedBy element as we discussed in the Chapter 7. Let us assume that User and BillingInfo entities have a bidirectional relationship and hence you must modify the BillingInfo entity to have the OneToOne relation definition pointing to User entity as follows:

```
@Entity
public class BillingInfo {
   @OneToOne(mappedBy="billingInfo")
   protected User user;
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
..
}
```

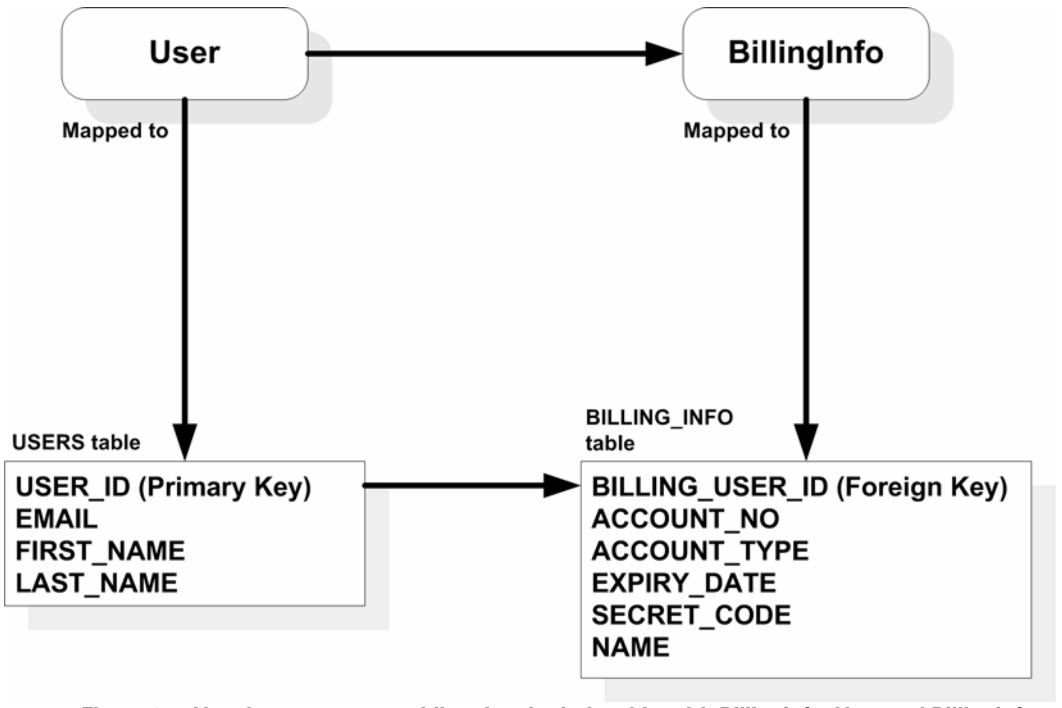
If you look at above code you will realize that `mappedBy` element identifies the name of the association field in the owning side of the relationship. In a bidirectional relationship owning-side of the relationship is the entity that stores the relationship in its underlying table. In our example `USERS` table stores the relationship in the `USER_BILLING_ID` field and hence is the relationship owner. The `OneToOne` relationship in `BillingInfo` has `mappedBy` element specified as `billingInfo` that is the relationship field defined in the `User` entity that has definition for JoinColumn.

Note that you do not have to define `@JoinColumn` in the entities of both side of OneToOne relation.

Next we will discuss how you define the OneToOne relationship when the foreign key is in the table to which child entity is mapped.

Using @PrimaryKeyJoinColumn

In the more likely case that the foreign key reference exists in the table to which the referenced Entity is mapped to, the `@PrimaryKeyJoinColumn` would be used instead.



52 **Figure 8.5: User has a one-one unidirectional relationship with BillingInfo. User and BillingInfo entities are mapped to USERS and BILLING_INFO tables respectively and BILLING_INFO and USERS table share the same primary key i.e. primary key of BILLING_INFO table is also a foreign key referencing the primray key of USERS table. Such associations are mapped using @PrimaryKeyJoinColumn.**

Typically, the `@PrimaryKeyJoinColumn` is used in one-to-one relationships when both the referenced and referencing tables share the primary key of the referencing table. In our example as

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

shown in Figure 8.5, the BILLING_INFO table would contain a foreign key reference named BILLING_USER_ID pointing to the USER_ID primary key of the USERS table. In addition, BILLING_USER_ID would be the primary key of the BILLING_INFO table. The relationship would be implemented as in listing 8.8.:

Listing 8.8: Mapping OneToOne Relationship Using @PrimaryKeyJoinColumn

```
@Entity
@Table(name="USERS")
public class User {
    @Id
    @Column(name="USER_ID")
    protected Long userId;
    ...
    @OneToOne
    @PrimaryKeyJoinColumn(name="USER_ID",                          |#1
        referencedColumnName="BILLING_USER_ID")                    |#1
    protected BillingInfo billingInfo;
}

@Entity
@Table(name="BILLING_INFO")
public class BillingInfo {
    @Id
    @Column(name="BILLING_USER_ID")
    protected Long userId;
    ...
}
```
(annotation) <#1 Parent Primary Key Join>

The @PrimaryKeyJoinColumn annotation's name element refers to the primary key column of the table storing the current Entity. On the other hand, the referencedColumnName element refers to the foreign key in the table holding the referenced Entity. In our case, the foreign key is the BILLING_INFO table's BILLING_USER_ID column and it points to the USERS.USER_ID primary key. If the names of both the primary key and foreign key columns are the same, you may omit the referencedColumnName element since this is what the JPA provider will assume by default. In our example, if we rename the foreign key in the BILLING_INFO table from BILLING_USER_ID to USER_ID to match the name of the primary key in the USERS table, we may omit the referencedColumnName value so that the provider can default it correctly.

If you have a composite primary key in the parent table (rare if you are using surrogate keys) you should use the @PrimaryKeyJoinColumns annotation instead. We encourage you to explore this annotation on your own if you need it.

We will see how to map one-many relationships next.

8.3.2 One-To-Many and Many-To-One

As we mentioned in the previous chapter, one-to-many and many-to-one relations are the most common in enterprise systems and are implemented using the @OneToMany and @ManyToOne annotations. For example, the Item-Bid relation is the ActionBazaar system is one-many, since an Item holds references to a collection of Bids placed on it and a Bid holds a reference to the Item it was placed on. The beauty of EJB 3.0 persistence mapping is that the same two annotations we used for mapping one-to-one relations are also used for one-many relations. This is because both relation

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

types are implemented as primary-key/foreign-key association in the underlying database. Let us see how this is done by implementing the `Item-Bid` relation as in Listing 8.9.

Listing 8.9: One-Many Bidirectional Relationship Mapping

```
@Entity
@Table(name="ITEMS")
public class Item {
    @Id
    @Column(name="ITEM_ID")
    protected Long itemId;
    ...
    @OneToMany(mappedBy="item")
                                            |#1
                                       |#1
    protected Set<Bid> bids;
    ...
}

@Entity
@Table(name="BIDS")
public class Bid {
    @Id
    @Column(name="BID_ID")
    protected Long bidId;
    ...
    @ManyToOne
    @JoinColumn(name="BID_ITEM_ID",                                            |#2
        referencedColumnName="ITEM_ID")                                       |#2
    protected Item item;
    ...
}
```

(annotation) <#1 One-to-Many Relationship Mapping>
(annotation) <#2 Many-to-One Relationship Mapping>

Since multiple instances of `BIDS` records would refer to the same `ITEM` record, the `BIDS` table will hold a foreign key reference to the primary key of the `ITEMS` table. In our example this foreign key is `BID_ITEM_ID` and it refers to the `ITEM_ID` primary key of the `ITEMS` table. This database relation between the tables is shown in Figure 8.6. In Listing 8.9 the many-one relation is expressed in O-R mapping using `@JoinColumn` annotations. In effect, a `@JoinColumn` annotation's job is to specify a primary/foreign key relationship in the underlying data model. Note that the exact `@JoinColumn` specification could have been repeated for both the `Bid.item` and `Item.bids` persistent fields on either side of the relation. In ManyToOne (#1), the `name` element specifies the foreign key, `BID_ITEM_ID` while the `referencedColumnName` element specifies the primary key, `ITEM_ID`. From the `Item` Entity's perspective, this means the persistence provider would figure out what `Bid` instances to put in the `bids Set` by retrieving the matching BIDS_ITEM_ID in the BIDS table.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

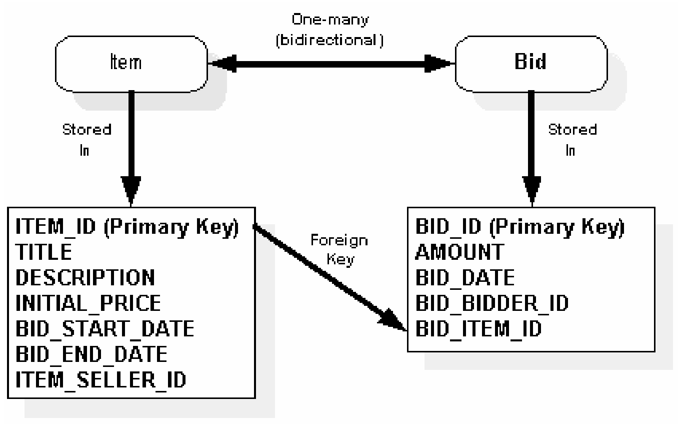


Figure 8.6: The one-many relation between the Item and Bid tables is materialized through a foreign key in the Bid table referring to the primary key of the Item table.

After performing the join, the JPA provider will see what `BIDS` records are retrieved by the join and populate them into the `bids Set`. Similarly, when it is time to materialize the `item` reference in the `Bid` Entity, the persistence provider would populate the `@JoinColumn`-defined join with the available `BID_ITEM_ID` foreign key, retrieve the matched `ITEMS` record and put it into the `item` field.

Instead of repeating the same `@JoinColumn` annotation, we have used the `mappedBy` element (#2) we mentioned but did not detail in Chapter 7. Note that persistence provider will generate deployment-time errors if you specify @JoinColumn on both side of a bidirectional one-many relationship. We can specify this element in the `OneToMany` element on the `Item.bids` variable as follows:

```
public class Item {
    ...
   @OneToMany(mappedBy="item")
   protected Set<Bid> bids;
   ...
}
```

The `mappedBy` element is essentially pointing to the previously the relationship field `Bid.item` with `@JoinColumn` definition . In a bidirectional one-many relationship the owner of relationship is the Entity side that stores the foreign key i.e. the many-side of the relationship.

The persistence provider will know to look it up appropriately when resolving `Bid` Entities. In general, you have to use the `mappedBy` element wherever bi-directional relationship is required. If you do not specify `mappedBy` element with the `@OneToOne` annotation the persistence provider will treat the relationship as a uni-directional relationship. Obviously, this would not be possible if the `OneToMany` relationship reference were unidirectional since there would be no owner of the relation to refer to.

Unfortunately JPA does not support uni-directional `One-To-Many` relationship using foreign key on the target table and you cannot use the following mapping if you have a uni-directional `OneToMany` relationship between Item and Bid:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
@OneToMany(cascade=CascadeType.ALL)
@JoinColumn(name="ITEM_ID", referencedColumnName="BID_ITEM_ID")
   protected Set<Bid> bids;
```

Although many persistence provider will support the above mapping this is not standardized and you have to use a join or intersection table using `@JoinTable` annotation similar to `ManyToMany` relationship that we discuss in 8.3.3. Uni-directional OneToMany are relationship are scarce and we will leave this you to explore on your own. We would rather recommend that you convert your relationship to a bi-directional relation thus avoding complexities in maintaining another table.

Also, the `ManyToOne` annotation does not support the `mappedBy` element since it is always the side of the relation that holds the foreign key and the inverse side can never be the “relationship owner”.

A final point to remember is that foreign keys can refer back to the primary key of the same table it resides in. There is nothing stopping a `@JoinColumn` annotation from specifying such a relation. For example, the many-to-one relationship between subcategories and parent categories could be expressed as shown in Listing 8.10:

31 Listing 8.10: ManyToOne Self-Referencing Relationship Mapping

```
@Entity
@Table(name="CATEGORIES")
public class Category implements Serializable {
    @Id
    @Column(name=“CATEGORY_ID”)
    protected Long categoryId;
    ...
    @ManyToOne
    @JoinColumn(name="PARENT_ID",                                            |#1
        referencedColumnName=”CATEGORY_ID")                                  |#1
    Category parentCategory;     ...
}
```

(annotation) <#1 Self-Referencing ManyToOne Relationship Mapping>

In Listing 8.10, the `Category` Entity refers to its parent through the `PARENT_ID` foreign key pointing to the primary key value of another record in the `CATEGORY` table. Since multiple sub-categories can exist under a single parent, the `@JoinColumn` annotation specifies a many-one relationship. Recall that the `@JoinColumns` annotation used in case of the rare case of composite keys for both the one-one and one-many cases.

We will conclude the section on mapping domain relations next by dealing with the most complex relationship mapping, many-to-many.

8.3.3 Many-to-Many

As we mentioned in section 8.1.1, a many-to-many relationship in the database world is implemented by breaking it down into two one-many relationships stored in an association or join table. In other words, an association or join table allows us to indirectly match-up primary keys from either side of the relationship by storing arbitrary pairs of foreign keys in a row. This scheme is depicted in Figure 8.7.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

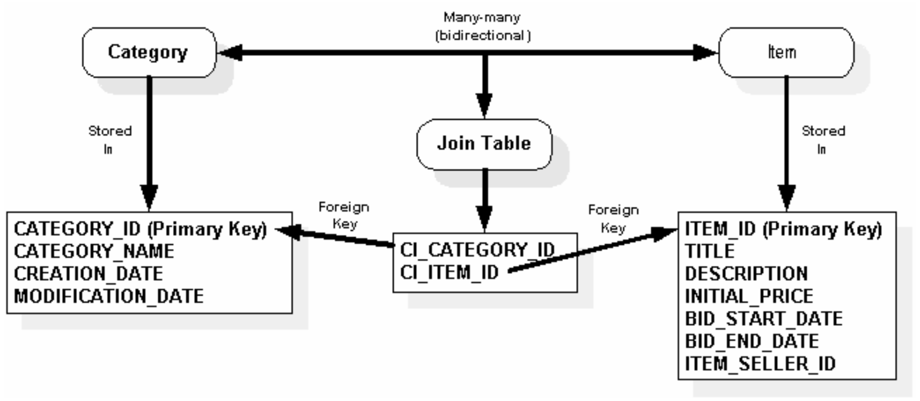


Figure 8.7: Many-to-many relationships are modeled in the database world using join tables. A join table essentially pairs foreign keys pointing to primary keys on either side of the relationship.

The `@JoinTable` mapping in EJB 3.0 models this technique. To see how this is done, let us take a look at the code in Listing 8.11 for mapping the many-many relationship between the `Item` and `Category` Entities. Recall that a `Category` can contain multiple `Items` while an `Item` can belong to multiple `Categories`.

Listing 8.11: Many-to-Many Relationship Mapping

```
@Entity
@Table(name="CATEGORIES")
public class Category implements Serializable {
    @Id
    @Column(name="CATEGORY_ID")
    protected Long categoryId;

    @ManyToMany
    @JoinTable(name="CATEGORY_ITEMS",                         |#1
        joinColumns=                                          |#1
            @JoinColumn(name="CI_CATEGORY_ID",                |#1
                referencedColumnName="CATEGORY_ID"),          |#1
        inverseJoinColumns=                                   |#1
            @JoinColumn(name="CI_ITEM_ID",                    |#1
                referencedColumnName="ITEM_ID"))              |#1
    protected Set<Item> items;
    ...
}

@Entity
@Table(name="ITEMS")
public class Item implements Serializable {
    @Id
    @Column(name="ITEM_ID")
    protected Long itemId;
    ...
    @ManyToMany(mappedBy="items")                             |#2
    protected Set<Category> categories;
    ...
}
```

(annotation) <#1 Owning Many-Many Relationship Mapping>
(annotation) <#2 Subordinate Many-Many Relationship Mapping>

The `@JoinTable` annotation's `name` element specifies the association or join table, which is named `CATEGORY_ITEMS` in our case. The `CATEGORY_ITEMS` table contains only two columns,

Licensed to John Sweitzer <admin@saolailaem.info>

CI_CATEGORY_ID and CI_ITEMS_ID. The CI_CATEGORY_ID column is a foreign key reference to the primary key of the CATEGORIES table, while the CI_ITEM_ID column is a foreign key reference to the primary key of the ITEMS table. The joinColumns and inverseJoinColumns elements represent exactly this fact. Each of the two elements describes a join condition on either side of the many-many relationship. The joinColumns element describes the "owning" relationship between the Category and Item Entities and the inverseJoinColumns element describes the "subordinate" relationship between them. Note the distinction of the "owning" side of the relationship is purely arbitrary.

Just as we used the mappedBy element to reduce redundant mapping for one-many relations, we are using the mappedBy element on the Item.categories field#2 to point to the @JoinTable definition in Category.items. We can specify more than one join column with joinColumns annotation if we have more than one column that constitutes the foreign key (again this is an unlikely situation that should be avoided in a clean design). From the perspective of the Category Entity, the peristence provider will determine what Item Entities go in the items collection by setting the available CATEGORY_ID primary key against the combined joins defined in the @JoinTable annotation, figuring out what CI_ITEM_ID foreign keys match and retrieving the matching records from the ITEM table. The flow of logic is essentially reversed for populating Item.categories. While saving the relationship into the database, the peristence provider might need to update all three of the ITEMS, CATEGORIES and CATEGORY_ITEMS tables as necessary. For a typical change in relation data, the ITEMS and CATEGORIES tables will remain unchanged while the foreign key references in the CATEGORY_ITEMS table will change. This might be the case if we move an item from one category to the other for example. Because of the inherent complexity of many-to-many mappings, the mappedBy element of the @ManyToMany annotation really shines in terms of reducing redundancy.

If you a unidirectional many-many relationship then the only difference is that the inverse side of the relation does not contain the mappedBy element.

We have now finished discussing Entity relation mapping and will discuss mapping OO inheritance next before concluding the chapter.

8.4 Mapping Inheritance

In section 8.1.1 we mentioned the difficulties in mapping OO inheritance into relational databases. We also alluded to the three strategies used to solve this problem: putting all Classes in the OO hierarchy in the same table, using joined tables for the super- and sub-Classes or using completely separate tables for each Class. We will explore how each strategy is actually implemented in this section.

Recall from section 8.2.7 that we could easily utilize different strategies for generating sequences more or less by changing configuration parameters for the @GeneratedValue annotation. The @Inheritance annotation used to map OO inheritance tries to follow the same philosophy of isolating strategy-specific settings into configuration. We will explore inheritance mapping using the three strategies offered through the @Inheritance annotation by implementing a familiar example.

As we mentioned before, the ActionBazaar system has several different user types including sellers and bidders. We have also introduced the idea of creating a User superclass common to all user

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

types. In this scheme of things, the `User` Entity would hold data and behavior common to all users while subclasses like `Bidder` and `Seller` would hold data and behavior specific to user types. A simplified class diagram for this OO hierarchy could look like Figure 8.8:

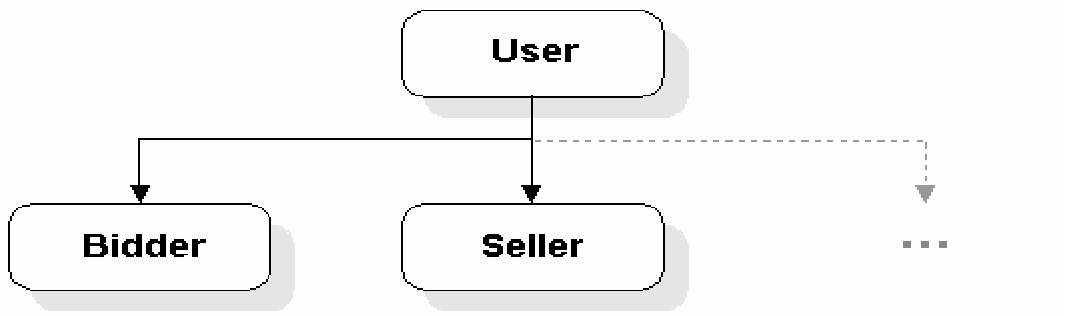


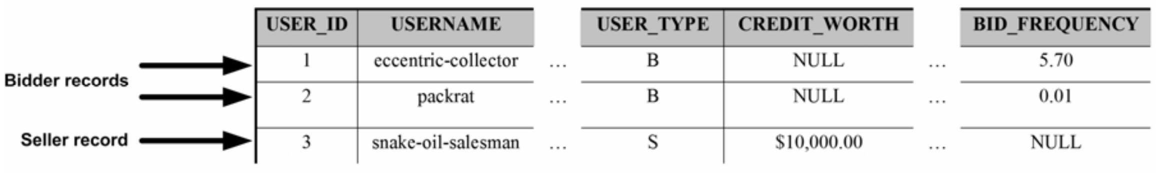
Figure 8.8: The ActionBazaar user hierarchy. Each user type like Bidder and Seller inherit from the common User superclass. The empty arrow signifies there may be some other subclasses of User class.

53

In this section we will discuss how the entire entities in the entity hierarchy in figure 8.8 can be mapped to database tables using different type of inheritance mapping strategies supported by JPA: Single table, Joined subclass and Table per class. Finally we will learn about polymorphic relationships. We will see how this hierarchy can be mapped using the `SINGLE_TABLE` strategy next.

8.4.1 Single Table Strategy

In this strategy all classes in the inheritance hierarchy are mapped to a single table. This means that the single table will contain a superset of all data stored in the class hierarchy. Different Objects in the OO hierarchy are identified using a special column called a *discriminator* column. In effect, the discriminator column contains a value unique to the object type in a given row. The best way of understanding this scheme is by seeing it implemented. For the ActionBazaar schema, let us assume that all user types including Bidders and sellers are mapped into the USERS table. Figure 8.9 shows how the table might look like:


Bidder records
Seller record

USER_ID	USERNAME		USER_TYPE	CREDIT_WORTH		BID_FREQUENCY
1	eccentric-collector	…	B	NULL	…	5.70
2	packrat	…	B	NULL	…	0.01
3	snake-oil-salesman	…	S	$10,000.00	…	NULL

Figure 8.9: Storing all ActionBazaar user types using a single table

As figure 8.9 depicts, the `USERS` table contains data common to all users (such as `USER_ID` and `USERNAME`), Bidder-specific data (such as `BID_FREQUENCY`) and seller-specific data (such as `CREDIT_WORTH`). Record 1 and 2 contain Bidder records while record 3 contains a seller record. This is indicated by the 'B' and 'S' values in the `USER_TYPE` column. We can imagine that the USER_TYPE discriminator column can contain values corresponding to other user types, such as 'A' for admin or 'C' for CSR. You might imagine that the Peristence provider maps each user type to the table by storing persistent data in to relevant mapped columns, setting the `USER_TYPE` value

Licensed to John Sweitzer <admin@saolailaem.info>

correctly and leaving the rest of the values NULL. The next step to understanding the SINGLE_TABLE strategy might be to analyze the actual mapping implementation. The code in Listing 8.12 shows the mapping for the User, Bidder and Seller Entities:

32 Listing 8.12: Inheritance Mapping Using Single Table

```
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)                    |#1
@DiscriminatorColumn(name="USER_TYPE",                                 |#2
    discriminatorType=DiscriminatorType.STRING, length=1)              |#2
public abstract class User ...

@Entity
@DiscriminatorValue(value="S")                                         |#3
public class Seller extends User ...

@Entity
@DiscriminatorValue(value="B")                                         |#4
public class Bidder extends User
```

(annotation) <#1 Inheritance Strategy>
(annotation) <#2 Discriminator Column>
(annotation) <#3 Seller Discriminator>
(annotation) <#4 Bidder Discriminator >

The inheritance strategy and discriminator column has to be specified on the root Entity of the OO hierarchy. In listing 8.12, we specify the strategy to be InheritanceType.SINGLE_TABLE in the @Inheritance annotation on the User Entity#1. The @Table annotation on the User Entity specifies the name of the single table used for inheritance mapping, USERS. The @DiscriminatorColumn annotation#2 specifies the details of the discriminator column. The name element specifies the name of the discriminator, USER_TYPE. The discriminatorType element specifies the data type of the discriminator column, which happens to be String and the length element specifies the size of the column, 1. Both subclasses of User, Bidder and Seller specify a discriminator value using the @DiscriminatorValue annotation. The Seller Class specifies its discriminator value to be 'S'#3. This means that when the peristence provider saves a Seller Object into the USERS table, it will set the value of the USER_TYPE column to be 'S'. Similarly, Seller Entities would only be reconstituted from rows with a discriminator value of 'S'. Likewise, the Bidder subclass specifies its discriminator value to be 'B'. Every subclass of User is expected to specify an appropriate discriminator value not conflicting with other sub-classes. If we do not specify a discriminator value for a subclass, the value is assumed to be the name of the subclass (such as "Seller" for the Seller Entity).

SINGLE_TABLE is the default inheritance strategy for EJB 3.0. Although this strategy is very simple to use, it has one great disadvantage that might be apparent from Figure 8.5. It does not really fully utilize the power of the relational database in terms of using primary/foreign keys and results in a large number of NULL column values.

To understand why, consider the seller record (record number 3) in Figure 8.9. The BID_FREQUENCY value is set to NULL for this record since it is not a Bidder record and the Seller Entity does not map this column. Conversely, none of the Bidder records ever populate the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

CREDIT_WORTH column. It is not that hard to imagine the amount of redundant NULL-valued columns in the USER table if there are a significant numbers of users (such as a few thousand).

For the very same reason, the strategy also limits the ability to enforce data integrity constrains. For example, if you want to enforce a column constraint that BID_FREQUNCY cannot be NULL for a Bidder, you would not be able to enforce the constraint since the same column will contain seller records for which the value may be NULL. Typically, such constraints are enforced through alternative mechanisms such as database triggers for the SINGLE table strategy. The second inheritance strategy we are now going to take a look at, JOINED, avoids these pitfalls and fully utilizes database relations.

8.4.2 Joined Table Strategy

The joined table inheritance strategy uses one-to-one relations to model OO inheritance. In effect, the joined table strategy means creating separate tables for each Entity in the OO hierarchy and relating direct descendants in the hierarchy with one-to-one relations. To understand it better, let us take a look at how the data in figure 8.10 might look using this strategy:

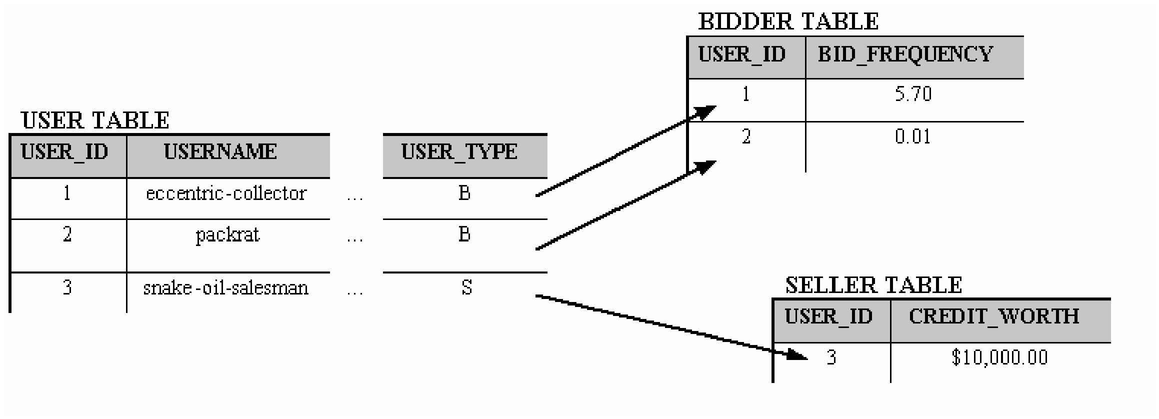


Figure 8.10: Modeling inheritance using joined tables. Each Entity in the OO hierarchy corresponds to a separate table and parent-child relationships are modeled using one-to-one mapping.

In the joined table strategy, the parent of the hierarchy contains only columns common to its children. In our example, the USERS table contains columns common to all ActionBazaar user types (such as USERNAME). The child table in the hierarchy contains columns specific to the Entity sub-type. In our case, both the BIDDERS and SELLERS tables contain columns specific to the Bidder and Seller Entities respectively (for example, the SELLERS table contains the CREDIT_WORTH column). The parent-child OO hierarchy chain is implemented using one-to-one relations. For example, the USERS and SELLERS tables are related through the USER_ID foreign key in the SELLERS table pointing to the primary key of the USERS table. A similar relation exists between the BIDDER and USERS tables. The discriminator column in the USERS table is still used, primarily as way of easily differentiating data types in the hierarchy. We are now ready to take a look at Listing 8.13 to see how the mapping strategy is actually implemented in EJB 3.0:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Listing 8.13: Inheritance Mapping Using Joined Table

```
@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.JOINED)                              |#1
@DiscriminatorColumn(name=”USER_TYPE”,
    discriminatorType=STRING, length=1)
public abstract class User ...

@Entity
@Table(name="SELLERS")
@DiscriminatorValue(value="S")
@PrimaryKeyJoinColumn(name="USER_ID")                                      |#2
public class Seller extends User ...

@Entity
@Table(name="BIDDERS")
@DiscriminatorValue(value="B")
@PrimaryKeyJoinColumn(name="USER_ID")                                      |#3
public class Seller extends User ...
```

(annotation) <#1 Inheritance Strategy>
(annotation) <#2 Primary Key Join>
(annotation) <#3 Primary Key Join>

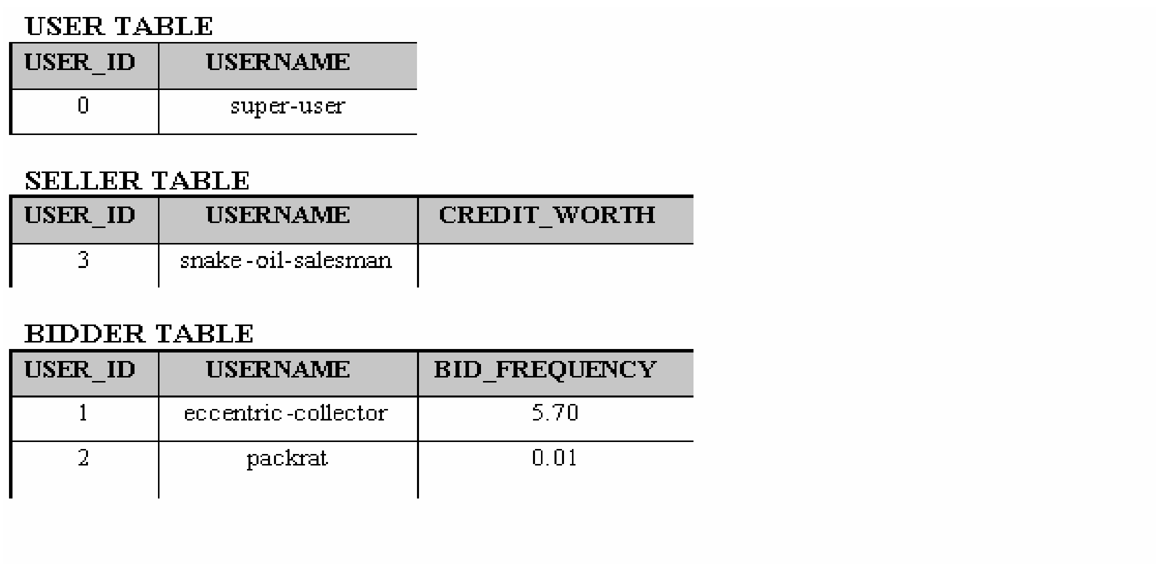
Listing 8.13 uses the `@DiscriminatorColumn` and `@DiscriminatorValue` annotations in exactly the same way as the `SINGLE_TABLE` strategy. The `@Inheritance` annotation's strategy element is specified to be `JOINED`, however#1. In addition, the one-to-one relationships between parent and child tables are implemented through the `@PrimaryKeyJoinColumn` annotations in both the Seller and Bidder Entities. In both cases, the `name` element specifies the `USER_ID` foreign key. Joined Subclass strategy is probably the best mapping strategy from design perspective. From performance perspective it is worse than the Single Table Per hierarchy strategy because it requires joining of multiple tables for polymorphic queries.

8.4.3 Table Per Class Strategy

Table-per-Class is probably the simplest inheritance strategy for a layman to understand. However, this inheritance strategy is the worst from both a relational and an OO standpoint. In this strategy, both the superclass and subclasses are stored in their own table and no relationship exists between any of the tables. To see how this works, take a look the tables in Figure 8.11.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>


USER TABLE

USER_ID	USERNAME
0	super-user

SELLER TABLE

USER_ID	USERNAME	CREDIT_WORTH
3	snake-oil-salesman	

BIDDER TABLE

USER_ID	USERNAME	BID_FREQUENCY
1	eccentric-collector	5.70
2	packrat	0.01

Figure 8.11: Table-per-Class inheritance strategy. Super- and sub-Classes are stored in their own, entirely unrelated tables.

As the Figure depicts, Entity data are stored in their own tables even if they are inherited from the superclass. This is true even for the `USER_ID` primary key. As a result, primary keys in all tables *must* be mutually exclusive for this scheme to work. In addition, inherited columns are duplicated across tables, such as the `USERNAME` column. Using this inheritance strategy, we define the strategy in the superclass and map tables for all the Classes. Listing 8.14 shows how the code might look.

Listing 8.14: Inheritance Mapping Using Table Per Class
```
@Entity
@Table(name="USERS")                                             |#1
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
|#2
public class User {
...
@Entity
@Table(name="SELLERS")                                           |#1
public class Seller extends User {
...
@Entity
@Table(name="BIDDERS")
|#1
public class Bidder extends User {
```
(annotation) <#1 Table Mappings>
(annotation) <#2 Inheritance Strategy >

As we can see, the inheritance strategy, `TABLE_PER_CLASS` is specified in the superclass Entity, `User`. However all of the concrete Entities in the OO hierarchy are mapped to separate tables and each have a `@Table` specification. The greatest demerit for this mapping type is that it does not have good support for polymorphic relations or queries because each subclass is mapped to its own table.

The limitation is that when we want to retrieve entities over the persistence provider will use SQL UNION or retrieve each entity with separate SQL for each subclass in the hierarchy.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Besides being awkward, this strategy is the hardest for an EJB 3.0 provider to implement reliably. As a result, implementing this strategy has been made optional for the provider by the EJB 3.0 specification. We recommend that you avoid this strategy altogether.

This finishes our analysis of the three strategies for mapping OO inheritance. Choosing the right strategy is not as straightforward as you might think. Table 8.2 compares each of mapping strategies.

Table 8.2: EJB 3.0 JPA supports three different inheritance strategies single table per class hirerarchy, joined subclass and table per class. Table per class is optional and worst of these strategies.

	Single Table per class hierarchy	**Joined Subclass Strategy**	**Table per class**
Table Support	One table for all classes in the entity hierarchy - Mandatory columns may be nullable - Table grows when more subclasses gets added	One for parent class and each subclass has a separate table to store polymorphic properties Most normalized tables	One table for each concrete class in the entity hierarchy
Use Discriminator Column	Yes	Yes	No
SQL Generated for retrieval of entity hierarchy	Simple SELECT	SELECT clause joining multiple tables	One SELECT for each sub class or UNION of SELECT
SQL for Insert and Update	Single INSERT or UPDATE for all entities in the hierarchy	Multiple INSERT, UPDATE. One for root class and one for each for involved subclass	One INSERT UPDATE for every subclass
Polymorphic relation	Good	Good	Poor
Polymorphic queries	Good	Good	Poor
Support in EJB 3.0 JPA	Required	Required	Optional

The single table strategy is relatively simple and is fairly performance friendly since it avoids joins under the hood. Even inserts and updates in the single table strategy perform better when compared to the joined table strategy. This is because in the joined table strategy, both the parent and child tables need to be modified for a given Entity subclass. However, as we mentioned, the single table strategy results in a large number of NULL-valued columns. Moreover, adding a new subclass essentially means updating the unified table each time.

In the joined table strategy, adding a subclass means adding a new child table as opposed to altering the parent table (which may or may not be easier to do). By and large, we recommend the joined table strategy since it best utilizes the strengths of the relational database including using normalization techniques to avoid redundancy. In our opinion, the performance cost associated with joins is relatively insignificant, especially with surrogate keys and proper database indexing.

The table per Class strategy is probably the worst choice of the three. It is relatively counterintuitive and uses almost no relational database features. It almost magnifies the object relational impedance instead of attempting to bridge it. The most important reason to avoid this strategy, however, is that EJB 3.0 makes it optional for a provider to implement it. As a result, choosing this strategy might make your solution non-portable across implementations.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Beside these inheritance strategies EJB 3.0 JPA allows an entity to inherit from a non-entity class. Such a class is annotated with @MappedSuperClass annotation. Like an embeddable object, a mapped super class does not have an associated table. We leave details about MappedSuperClass as an exercise for you.

Eclipse O-R Mapping Tool

As we mentioned before, EJB 3.0 makes O-R mapping a lot easier, but not quite painless. This is largely because of the inherent complexity of O-R Mapping and the large number of possible combinations to handle. The good news is that an Eclipse-based EJB 3.0 Mapping tool code-named Dali is underway (http://www.eclipse.org/dali/). The project is lead by Oracle and supported by JBoss and BEA . It will be a part of Eclipse web tool project (WTP) and will support creating and editing EJB 3.0 Object-Relational Mappings using either annotations or XML. Also two commercial products Oracle JDeveloper and BEA Workshop support development of EJB 3.0 applications.

8.4.4 Mapping Polymorphic Relations

In chapter 7 we discussed that JPA fully supports inheritance and polymorphism. Now that we are complete with our discussions on mapping relationships and inheritance you must be excited to know about polymorphic associations and wondering how do you map polymorphic associations. A relation between two entities is said to be polymorphic when the actual relation may refer to instances of subclass of associated entity. Let us assume that there is a bidirectional many-to-one relationship between ContactInfo and User entities. We discussed earlier that User is an abstract entity and entities such as Bidder, Seller, etc inherit from User class. When we retrieve the association from ContactInfo entity User being abstract an instance of its subclass either Bidder or Seller will be retrieved. The greatest benefit of JPA is that you do not have to do any extra work for mapping polymorphic association and you get it for free. You just define the relationship mapping between super class and associated entity and the association becomes polymorphic.

8.5 Summary

In this chapter, we covered basic database concepts, introduced the Object-Relational Impedance problem. We reviewed the O-R mapping annotations such as `@Table, @Column` mapped some of the Entities into database tables.

We reviewed different types of primary key generation strategies and reviewed mapping of composite primary keys.

We reviewed different types of Relations presented in the previous chapter and used JPA annotations such as @JoinColumn and @PrimaryKeyColumn to map those into database tables. You learnt that @ManyToMany and uni-directional @OneToMany relations require association tables. Hope the limitation to support unidirectional relationship using a target foreign key constraint will be addressed in a future release.

We showed you the robust OO Inheritance mapping features supported by the JPA as well and compared their relative merits and demerits. Of three of the inheritance mapping strategies Joined Subclass is the best for design perspective.

You should note, however, that we avoided some complexities while presenting this chapter. First, we used field-based persistence in all of our code samples to keep them as short and simple as

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

possible. Secondly, we only mentioned the most commonly used elements for the annotations used in the chapter. We felt justified in doing so as most of the annotation elements we avoided are rarely used. We do encourage you to check out the full definition of all the annotations in this chapter available online at http://java.sun.com/products/persistence/javadoc-1_0-fr/javax/persistence/package-tree.html.

In the next chapter, we are going to see how to actually manipulate the Entity and Relations we mapped in this chapter using the EntityManager API.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Chapter 9 Manipulating entities with EntityManager API

In Chapter 7 we learned how to develop the application domain model using JPA. In Chapter 8 we saw how domain objects and relations are mapped to the database. While the ORM annotations we discussed in Chapter 8 indicate how an entity is persisted, the annotations don't do the actual persisting. The actual persistence is done by applications using the EntityManager interface, the topic of this chapter.

To use an analogy, the domain model annotated with ORM configuration is kind of like a children's toy that needs assembly. The domain model consists the parts of the toy. The ORM configuration is the assembly instructions. While the assembly instructions tell you how the toy parts are put together, you do the actual assembly. The `EntityManager` is the toy assembler of the persistence world. The `EntityManager` figures out how to persist the domain by looking at the ORM configuration. More specifically, the `EntityManager` performs Create, Read, Update and Delete (CRUD) operations on domain objects. The *Read* part includes extremely robust search and retrieval of domain objects from the database. This chapter covers each of the CRUD operations that the `EntityManager` provides in detail, with the exception of the *search* part of search and retrieval. In addition to simple primary key based domain object retrieval, JPA provides `SQL SELECT` like search capabilities through the EJB 3.0 query API. The query API is so extensive and powerful that we will dedicate the entirety of Chapter 10 to it while briefly touching on it in this one.

Before we dive down into the persistence operations we will learn about EntityManager interface, lifecycle of entities, persistence context and how to obtain an instance of EntityManager. We will discuss about entity lifecycle callback listeners before concluding with best practices.

Before we get into too much code, we are going to gently introduce the `EntityManager` and briefly cover some concepts useful in understanding the nuances behind this critical part of JPA.

9.1 Introducing the EntityManager

EntityManager API is probably most important and interesting part of Java Persistence API. It manages the lifecycle of entities. In this section you will learn about EntityManager and methods in the EntityManager interface. We will uncover the entity lifecycle and then learn about persistence contexts and their types.

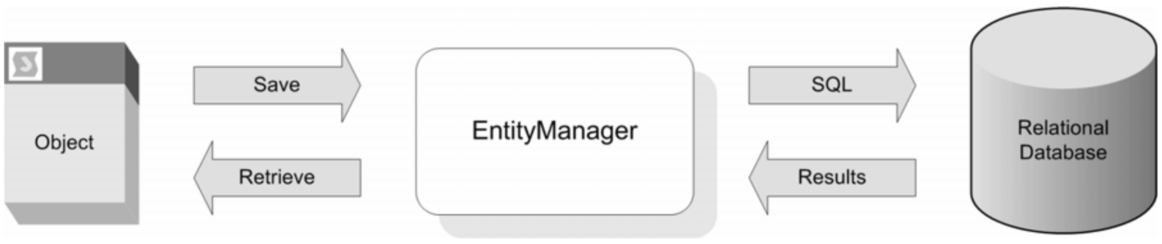
9.1.1 The EntityManager Interface

In a sense, the `EntityManager` is the bridge between the OO and relational worlds as depicted in Figure 9.1. When we ask that a domain Entity be created, the `EntityManager` translates the Entity into a new database record. When we ask that an Entity be updated, it tracks down the

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

relational data that correspond to the Entity and updates the data. Likewise the `EntityManager` removes the relational data when we ask that an Entity be deleted. From the other side of the translation bridge, when we ask for an Entity "saved" in the database, the `EntityManager` creates the Entity object, populates it with relational data and "returns" the Entity back to the OO world.



54 **Figure 9.1: EntityManager acts as the bridge between OO and relational world. It interprets the O-R mapping specified for an entity and saves in the database**

Besides providing these explicit SQL-like CRUD operations, the `EntityManager` also tries to quietly keep Entities synched with the database automatically as long as they are *within the `EntityManager`'s reach* (this behind-the-scenes synchronization is what we mean when we talk about "managed" Entities in the next Section) The `EntityManager` is easily the most important interface in JPA and is responsible for most of the ORM black magic in the API.

Despite all this under-the-hood power, the `EntityManager` is a very small, simple and intuitive interface, especially compared to the mapping steps we already discussed in the previous Chapter and the query API to be discussed in the next Chapter. In fact, once we go over some basic concepts in the next few sections, the interface might seem almost trivial. You might already agree if you took a quick look at Table 9.1. The table lists some of the most commonly used methods defined in the `EntityManager` interface.

1.17 Table 9.1: The EntityManager is used to perform CRUD operations. The following are the most commonly used methods of the EntityManager interface.

Method Signature	**Description**
public void persist(Object entity);	Saves (persists) an Entity into the database.
public <T> T merge(T entity);	Merges an Entity to the EntityManager's persistence context and returns the merged Entity.
public void remove(Object entity);	Removes an Entity from the database.
public <T> T find(Class<T> entityClass, Object primaryKey);	Find an entity instance by its primary key.
public void flush();	Synchronize the state of Entities in the EntityManager's persistence context with database.
public void setFlushMode(FlushModeType flushMode);	Change the flush mode of the EntityManager's persistent context. The flush mode may either be AUTO or COMMIT. The default flush mode is AUTO, meaning that the EntityManager tries to automatically synch the Entities with the database.
public FlushModeType getFlushMode();	Retrieve the current flush mode.
public void refresh(Object entity);	Refresh (reset) the Entity from the database.
public Query createQuery(String jpqlString);	Create a dynamic query using a JPQL statement.
public Query createNamedQuery(String name);	Create a query instance based on a named query on the Entity instance.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

public Query createNativeQuery(String sqlString);
 	
public Query createNativeQuery(String sqlString, Class result Class);
 	
public Query createNativeQuery(String sqlString, String resultSetMapping);
	Create a dynamic query using a native SQL statement.

public void close();	Close an EntityManager.
public boolean isOpen();	Check whether an EntityManager is open.
public EntityTransaction getTransaction();	Retrieve a transaction object that can be used to manually start or end a transaction.
public void joinTransaction();	Ask an EntityManager to join an existing JTA transaction.

Don't worry too much if the methods are not immediately obvious. Except for the methods related to the query API (`createQuery`, `createNamedQuery` and `createNativeQuery`), we will discuss them in detail in the coming sections. The few `EntityManager` interface methods that we did not cover above are rarely ever used, so we won't spend time in the limited scope of this book talking about them. Once you've read and understood the material in this Chapter though, we encourage you to explore them on your own. The EJB 3.0 Java Persistence API final specification is available at http://jcp.org/en/jsr/detail?id=220.

The JPA Entity: A Right Set of Tradeoffs

Nothing in life is free…

As we mentioned and will explore further this Chapter onward, the reworked EJB 3.0 JPA Entity model brings a whole host of things to the table: simplicity, OO support, unit testability, and so on. However, the JPA Entity loses a few features that were available in the EJB 2.x model because of its separation from the container. Because the EntityManager and not the container manages Entities, they cannot directly use container services such as dependency injection, defining method level transaction and security attributes, remoteability, and so on.

However, the truth of the matter is most layered applications designed using the EJB 2.x CMP "Entity Bean" model never really utilized container services directly anyway. This is because Entity Beans were almost always used through a Session Bean façade. This means that Entity Beans "piggybacked" over container services configured at the Session Bean level. The same is true for JPA Entities. This means that in real terms, the JPA model loses very little functionality.

Incidentally, losing the magic word "bean" means that JPA Entities are no longer EJB components managed by the container. So if you catch someone calling JPA Entities "beans", feel free to gently correct them!

Even though JPA is not container-centric like Session Beans or MDBs, Entities still have a life cycle. This is because Entities are "managed" by JPA in the sense that the API keeps track of them under the hood and even automatically synchronizes Entity state with the database when possible. We'll explore exactly how the Entity life cycle looks like in the following section.

9.1.2 The Life Cycle of an Entity

An Entity has a pretty simple life cycle. Making sense of the Entity life cycle and remembering it is easy once you understand a simple concept–the `EntityManager` knows nothing about a POJO regardless of how it is annotated, until you tell the manager to start treating the POJO like a JPA Entity. This is the exact opposite of POJOs annotated to be Session Beans or Message Driven Beans, which are loaded and managed by the container as soon as the application starts. Moreover, the default behavior of the `EntityManager` is to manage an Entity for as short of a time as possible.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Again, this is the opposite of container-managed beans, which remain managed until the application is shut down.

An Entity that the `EntityManager` is keeping track of is considered *attached* or *managed*. On the other hand, when an `EntityManager` stops managing an Entity, the Entity is said to be *detached*. Lastly, an Entity that was never managed at any point is called *transient* or *new*.

Figure 9.2 summarizes all of the Entity life cycle.

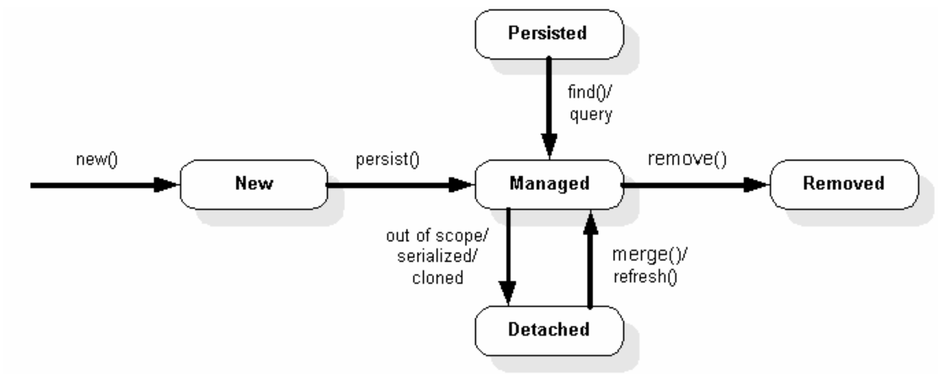


Figure 9.2 : An entity becomes managed when we persist, merge, refresh or retrieve an entity. It may also be attached when we retrieve it. A managed entity becomes detached when it is out of scope, removed, serialized or cloned.

Let's take a close look at the managed and detached states.

Managed Entities

When we talk about *managing an Entity's state*, what we mean is that the `EntityManager` makes sure that the entity's data is synchronized with the database. The `EntityManager` ensures this by doing two things. Firstly, as soon as we ask an `EntityManager` to start managing an Entity, it synchronizes the Entity's state with the database. Secondly, until the Entity is no longer managed, the `EntityManager` ensures that changes to the Entity's data (caused by Entity method invocations, for example) are reflected in the database. The `EntityManager` accomplishes this feat by holding an object reference to the managed Entity and periodically checking for data freshness. If the `EntityManager` finds that any of the Entity's data has changed, it automatically synchronizes the changes with the database. The `EntityManager` stops managing the Entity when the Entity is either deleted or the Entity moves out of persistence provider's reach.

An Entity can become attached to the `EntityManager`'s context in many ways: by passing the Entity to the `persist`, `merge` or `refresh` methods. Also an entity becomes attached when an entity is is retrieved using the `find` method or a query within a transaction. The state of the entity determines which method you will use.

When an Entity is first instantiated as in the following snippet, it is in the *new* or *transien* state since the `EntityManager` doesn't know it exists yet:

Licensed to John Sweitzer <admin@saolailaem.info>

```
Bid bid = new Bid();
```

Hence the entity instance is not managed yet. It will become managed if the `EntityManager`'s `persist()` method creates a new record in the database corresponding to the Entity . This would be the most natural way to for the `Bid` Entity in the previous snippet to be attached to the `EntityManager`'s context:

```
manager.persist(bid);
```

A managed entity becomes detached when it is out of scope, removed, serialized or cloned. For example, the instance of Bid entity will become detached when the underlying transaction commits.

Unlike Entities explicitly created using the `new` operator, an Entity retrieved from the database using the `EntityManager`'s `find()` method or a query is attached if retrieved within a transactional context. A retrieved instance of entity become detached immediately if there is not an associated transaction.

The `merge()` and refresh() methods are meant for entities that have been retrieved from the database and are in detached state, and will attach them to the EntityManager.The `merge()` updates the database with the data held in the entity. The `refresh()`does the opposite of what the `merge` method does—it resets the Entity's state with data from the database. We'll discuss all of these methods in much greater detail in sections 9.3.

Detached Entities

A detached entity is an entity that is no longer managed by EntityManager and there is no guarantee that state of entity is in sync with the database. Detachment and merge operations become handy when you want to pass an entity across application tiers. For example, you can detach an entity and pass to the web tier and then update it and send it back to the EJB-tier where you can merge the detached entity to the persistence context.

`There are three ways to detach an Entity.`

The common way Entities become detached is a little subtler. Essentially, an attached Entity becomes detached as soon as it goes out of the `EntityManager` context's scope. Think of this as the invisible link between an Entity and the `EntityManager` being expired at the end of a logical unit of work or a session. An `EntityManager` session could be limited to a single method call or span an arbitrary length of time (this is reminiscence of Session Beans, isn't it? As we will soon see, this is not entirely an accident). For an `EntityManager` whose session is limited to a method call, all Entities attached to it become detached as soon as a method returns, even if the Entity objects are used outside the method. If this is not absolutely crystal clear right now, it will be once we talk about the `EntityManager` persistence context in the next Section. Entity instances also become detached through cloning or serialization. This is because the `EntityManager` quite literally keeps track of Entities through Java object references. Since cloned or serialized instances don't have the same object references as the original managed Entity, the `EntityManager` has no way of knowing they exist.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

This scenario happens most often in situations where Entities are sent across the network for Session Bean remote method calls.

If you call the `clear` method of EntityManager, it forces all entities in the persistence context to be detached.

Calling the `EntityManager remove()` method will also detach it. This makes perfect common sense since this method removes the data associated with the Entity from the database. As far as the `EntityManager` is concerned, the Entity no longer exists, so there is no need to continue managing it. For our `Bid` Entity, this would be an “apt demise”:

```
manager.remove(bid);
```

We will return to discussion on detachment and merge operations in section 9.3.3.

A good way to remember the Entity lifecycle is through a convenient analogy. Think of an Entity as an aircraft and the `EntityManager` as the air traffic controller. While the aircraft is outside the range of the airport (detached or new), it is not guided by the air traffic controller. However, when it does come into range (managed), the traffic controller manages the aircraft’s movement (state synchronized with database). Eventually, a grounded aircraft is guided into takeoff and goes out of airport range again (detached), at which point the pilot is free to follow his own flight plan (modifying a detached Entity without state being managed).

The *persistence context scope* is the equivalent of airport radar range. It is critical to understand how the persistence context works to use managed Entities effectively. We will examine the relation between the persistence context, its scope and the `EntityManager` in the next Section.

9.1.3 Persistence Contexts, Scope and the EntityManager

The persistence context plays vital part in the internal functionality of the `EntityManager`.

Although we perform persistence operations by invoking methods on the `EntityManager`, the `EntityManager` itself does not directly keep track of the life cycle of an individual Entity. In reality, the `EntityManager` delegates the task of managing Entity state to the currently available persistence context.

In a very simple sense, a persistence context is a self-contained collection of Entities managed by an `EntityManager` during a given persistence scope. The persistence scope is the duration of time a given set of Entities remains managed.

The best way of understanding what this means is to start by examining what the different persistence scopes are and what they do and then backtracking back to the meaning of the term. We’ll explain how the persistence context and persistence scope relates to the `EntityManager` by first exploring what the persistence context is. We will then tell you what a persistence context scope is and how it affects both the `EntityManager` and the persistence context.

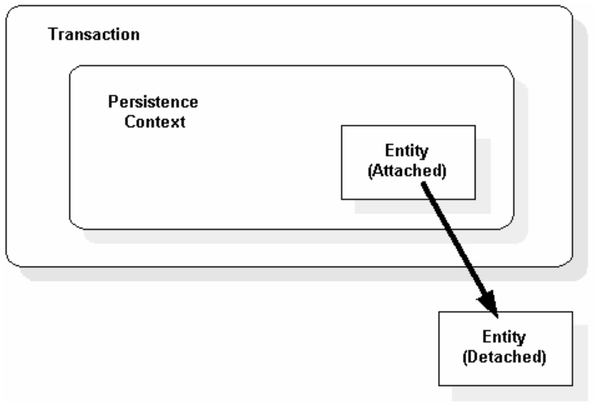
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

There are two different types of persistence scopes: *transaction* and *extended*.

Transaction Scoped EntityManager

If a persistence context is under transaction scope, Entities attached during a transaction are automatically detached when the transaction ends (note, all persistence operations that may result in data changes must be performed inside a transaction, no matter what the persistence scope is). In other words, the persistence context keeps managing Entities while the transaction it is enclosed by is active. Once the persistence context detects that a transaction has either been rolled back or committed, it will detach all managed Entities after making sure all data changes until that point is synchronized with the database. Figure 9.3 depicts this relationship between Entities, the transaction persistence scope, and persistence contexts:



55 **Figure 9.3: Transaction scoped persistence contexts only keep Entities attached within the boundaries of the enclosing transaction.**

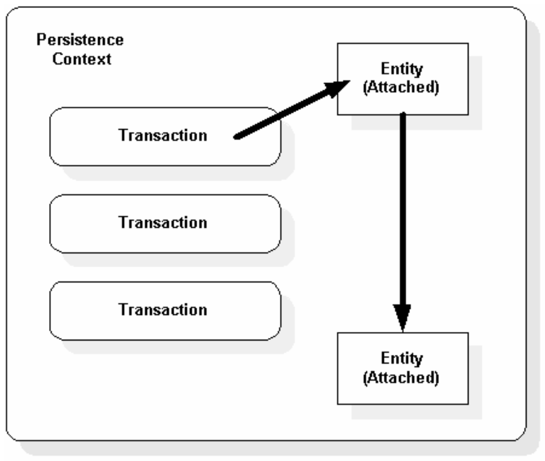
An EntityManager associated with a transaction scoped persistence context is known as transaction-scoped EntityManager.

Extended EntityManager

The life span of extended EntityManager lasts across multiple transactions. An extended EntityManager can only be used with Stateful session beans and lasts as long as the bean instance is alive. Hence in persistent contexts with extended scope, how long Entities remain managed has nothing to do with transaction boundaries. In fact, once attached, Entities pretty much stay managed as long as the `EntityManager` instance is around. As an example, for a Stateful Session Bean, an `EntityManager` with extended scope will keep managing all attached Entities until the `EntityManager` is closed as bean itself is destroyed. As figure 9.4 shows, this means that unless explicitly detached through the `@Remove` method to end the life of stateful bean instance, Entities attached to an extended persistence context will remain managed across multiple transactions.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>



56 **Figure 9.4: For an extended persistence context, once an Entity is attached in any given transaction, it is kept managed for all transactions in the lifetime of the persistence context.**

The term *scope* is used for persistence contexts in the same manner that it is used for Java variable scoping. It is used to describe how long a particular persistence context remains active. Transaction-scoped persistence contexts can be compared to method local variables, in the sense that they are only in effect within the boundaries of a transaction. On the other hand, extended-scoped persistence contexts are more like instance variables that are active for the lifetime of an object—they hang around as long as the `EntityManager` is around.

At this point, we've covered the basic concepts needed to understand the functionality of the `EntityManager`. We are now ready to see the `EntityManager` itself in action.

9.1.4 Using the EntityManager in ActionBazaar

We will explore the EJB 3.0 `EntityManager` interface by implementing an ActionBazaar component. We will implement the `ItemManagerBean` Stateless Session Bean used to provide the operations to manipulate items. As code Listing 9.1 demonstrates, the Session Bean provides methods for adding, updating and removing `Item` Entities using the JPA `EntityManager`.

33 Listing 9.1: ItemManagerBean Using the EntityManager Interface

```
@Stateless
public class ItemManagerBean implements ItemManager {
    @PersistenceContext(unitName="actionBazaar")
    private EntityManager entityManager;                            |#1

    public ItemManagerBean() {}

    public Item addItem(String title, String description,
        byte[] picture, double initialPrice, long sellerId) {
        Item item = new Item();
        item.setTitle(title);
        item.setDescription(description);
        item.setPicture(picture);
        item.setInitialPrice(initialPrice);
        Seller seller = entityManager.find(Seller.class, sellerId); |#2
        item.setSeller(seller);
        entityManager.persist(item);                                |#3
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
        return item;
    }

    public Item updateItem(Item item) {
        entityManager.merge(item);                                  |#4
        return item;
    }

    public Item undoItemChanges(Item item) {
        entityManager.refresh(entityManager.merge(item));           |#5
        return item;
    }

    public void deleteItem(Item item) {
        entityManager.remove(entityManager.merge(item));            |#6
    }
}
```
(annotation) <#1 Inject EntityManager Instance>
(annotation) <#2 Retrieve Entity Using Primary Key>
(annotation) <#3 Persist Entity Instance>
(annotation) <#4 Merge Changes to Database>
(annotation) <#5 Refresh Entity from Database>
(annotation) <#6 Remove Entity from Database>

ItemManagerBean is a pretty good representation of the most common ways the EntityManager API is used. Firstly, an instance of the EntityManager is injected using the @PersistenceContext annotation#1 and used in all the Session Bean methods that manipulate Entities. As you might imagine, the addItem method is used by the presentation layer to add an item posted by the seller to the database. The persist() method is used by addItem to add a new Item Entity into the database#2. The addItem method also uses the EntityManager find() method to retrieve the Seller of the Item using the Entity's primary key#2. The retrieved Seller Entity is set as an association field of the newly instantiated Item Entity along with all other item data. The updateItem method updates the Item Entity data in the database using the merge method#4. This method could be invoked from an administrative interface that allows a seller to update a listing after an item is posted. The EntityManager refresh() method is used in the undoItemChanges method to discard whatever changes were made to an Item Entity and reload it with the data stored in the database#5. The undoItemChanges method could be used by the same administrative interface that uses the updateItem method to allow the user to start over with modifying a listing (think of a HTTP form's "reset" button). Lastly an Item Entity is removed from the database using the remove() method#6. This method could be used by an ActionBazaar administration to remove an offending listing.

Now that we've "surface-scanned" the code in Listing 9.1, we're ready to start our in-depth analysis of the EntityManager API. We'll start from the most logical point: making an EntityManager instance available to the application.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

9.2 Creating EntityManager Instances

EntityManager is like the conductor of orchestra who manages the show. When you are impressed with a show and you want to take the show to your town. You first get in touch and try to hire the conductor. Similarly first obvious step to performing any persistence operation is obtaining an instance of an `EntityManager`.

In Listing 9.1, we do this by injecting an instance using the `@PersistenceContext` annotation. If you are using a container, this is more or less all you will need to know about getting an instance of an `EntityManager`.

All `EntityManager` instances injected using the `@PersistenceContext` annotation are container-managed.This means that the container takes care of the mundane task of looking up, opening and closing the `EntityManager` behind the scenes. In addition, unless otherwise specified, injected `EntityManager` have transaction scope. Just as you aren't limited to using the transaction scope, you are not limited to using a container managed `EntityManager` either.

JPA fully supports creating application-managed `EntityManagers` that you explicitly create, use and release, including controlling how the `EntityManager` handles transactions.This capability is particularly important for using JPA outside the container.

In this section we will explore how to create and use both container and application managed `EntityManagers`.

9.2.1 Container-Managed EntityManagers

As we saw, container-managed `EntityManagers` are created using the `@PersistenceContext` injection.

Let's take a look at the definition of the annotation to start exploring its features:

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceContextType type default TRANSACTION;
    PersistenceProperty[] properties() default {};
}
```

The first element of the annotation, `name`, specifies the JNDI name of the persistence context. This element is used in the very unlikely case you have to explicitly mention the JNDI name for a given container implementation to be able to lookup an `EntityManager`. In most cases, leaving this element empty is fine except when you use `@PersistenceContext` at class level to establish reference to the persistence context.

Persistence Unit

The `unitName` element specifies the name of the *persistence unit*. A *persistence unit* is essentially a grouping of Entities used in an application. This idea is really useful when you have a large Java EE

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

application and would like to separate it into several logical areas (think Java packages). For example, ActionBazaar Entities could be grouped into *general* and *admin* units. Persistence units cannot be set up using code. You must configure persistence units through the persistence.xml deployment descriptor. We will come back to the topic of configuring persistence units in Chapter 11 and will leave it alone for the time being. For now, all you need to understand is that we could get an `EntityManager` for the *admin* unit using the `unitName` element as follows:

```
@PersistenceContext(unitName="admin")
EntityManager entityManager;
```

In the typical case that a Java EE module has a single persistence unit, specifying the `unitName` might seem redundant. In fact, most persistence providers will resolve the unit correctly if don't specify a `unitName`. However, we recommend specifying a persistence unit name even if you only have one unit. This ensures that you are not dependent on container-specific functionality since the specification does not state what the persistence provider must do if the `unitName` is not specified.

EntityManager Scoping

The last element, `type`, specifies the `EntityManager` scope. As we noted, the scope for a container-managed `EntityManager` scope can either be `TRANSACTION` or `EXTENDED`. If the `type` element is left empty, the scope of the `EntityManager` is assumed to be `TRANSACTION`. Not surprisingly, the typical use of the `type` element is to specify `EXTENDED` scope for an `EntityManager` instead of the default `TRANSACTION` scope. The code would look like the following:

```
@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager entityManager;
```

Note, you are not allowed to use extended persistence scope for Stateless Session Beans or Message Driven Beans. If you stop and think for second, the reason should be pretty obvious. The real reason for using extended scope in a bean would be to manage Entity state across multiple method calls, even if each method call is a separate transaction. Since neither Sessions Beans nor Message Driven Beans are supposed to implement such functionality, it makes no sense to support extended scope for these bean types. On the other hand, extended persistence scope is ideal for Stateful Session Beans. An underlying `EntityManager` with extended scope could be used to cache and maintain the application domain across an arbitrary number of method calls from a client. More importantly, you could do this and still not have to give up method level transaction granularilty (most likely using CMT). We will return to the discussion of how you can use an extended persistence context for Stateful Session Beans as an effective caching mechanism in Chapter 13.

The real power of container-managed `EntityManager` is in the high degree of abstraction they offer. Behind the scenes, the container instantiates `EntityManagers`, binds them to JNDI, injects them into beans on demand and closes them when they are no longer needed (typically when a bean is destroyed).

If is difficult to get an appreciation of the amount of menial code the container takes care of until you see the alternative. Keep this in mind when we take a look at application-managed `EntityManagers` in the coming Section. Note container-managed `EntityManagers` are

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

available to all components in a Java EE container including JSF backing beans and Servlets. However be careful of the fact that `EntityManagers` are not thread-safe so injecting them frivolously into presentation layer components can get you into trouble easily. We'll discuss this nuance in the sidebar titled "EntityManagers and Thread-Safety".

EntityManager and Thread-Safety

It is very easy to forget the fact that web-tier components are meant to be used by multiple concurrent threads. Servlet based components like JSPs are deliberately designed this way because they are intended to achieve high throughput through statelessness.

However, this fact means that you cannot use resources that are not thread-safe as Servlet instance variables. The EntityManager falls under this category so injecting it into a web component is a big no-no. One way around this problem is using the SingleThreadModel interface to make a Servlet thread-safe. In practice, this technique severely degrades application performance and is almost always a bad idea.

Some vendors might try to solve this problem to guaranteeing a thread-safe EntityManager. If you are extremely comfortable with your container vendor, you could count on this. Remember though, one very important benefit of using EJB 3.0 is portability across container implementations. You shouldn't give this advantage up frivolously.

If you must use container-managed EntityManagers from your Servlet, the best option is to lookup an instance of the EntityManager inside your method as follows:

```
@PersistenceContext(name="pu/actionBazaar",unitName="ActionBazaar")
public class ItemServlet extends HttpServlet {
    @Resource private UserTransaction ut;
    public void service (HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Context ctx = new InitialContext();
        EntityManager entityManager =  (EntityManager) ctx.lookup("java:comp/env/pu/actionBazaar");
        …
        ut.begin();
        entityManager.persist(item);
        ut.commit();
        ...
    }
    …
}
```

The other alternative is to use an application-managed EntityManager with a JTA transaction. It is worth noting that EntityManagerFactory is thread-safe.

Now that you are warmed up working with EntityManager inside the container let us discuss about application-managed EntityManager.

9.2.2 Application-Managed EntityManager

An application-managed `EntityManager` wants nothing to do with a Java EE container.
This means that we must write code to control every aspect of the `EntityManager`'s lifecycle.

> By and large, application-managed `EntityManagers` are most appropriate for environments where a container is not available, such as Java SE or a lightweight web container like Tomcat.

However, a justification to use application-managed `EntityManagers` inside a Java EE container is to maintain fine-grained control over the `EntityManager` life cycle or transaction management. For this reason as well as to maintain flexibility, the EJB 3.0 API provides a few conveniences for using application-managed `EntityManagers` inside a container. This happens to

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

suit us well too because it provides an effective approach to exploring application-managed EntityManagers by reusing the code in Listing 9.1. Here is the code:

Listing 9.2: ItemManagerBean Using an Application-Managed EntityManager

```
@Stateless
public class ItemManagerBean implements ItemManager {
    @PersistenceUnit                                                   |#1
    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;

    public ItemManagerBean() {}

    @PostConstruct
    public void initialize() {
        entityManager = entityManagerFactory.createEntityManager(); |#2
    }
    ...
    public Item updateItem(Item item) {
        entityManager.joinTransaction();                               |#3
        entityManager.merge(item);
        return item;
    }
    ...
    @PreDestroy
    public void cleanup() {
        if (entityManager.isOpen()) {                                  |#4
            entityManager.close();                                     |#4
        }                                                              |#4
    }
    ...
}
```

(annotation) <#1 Inject EntityManagerFactory Instance>
(annotation) <#2 Create EntityManager>
(annotation) <#3 Explicitly Joining JTA Transaction>
(annotation) <#4 Closing EntityManager>

Eyeballing Listing 9.2, it should be fairly obvious that we are more or less explicitly doing what the container did for us behind the scenes in Listing 9.1. In #2, we create an EntityManager after the bean is constructed while in #4 we close it before the bean is destroyed, mirroring what the container does automatically. The same is true of the code in #3 to explicitly join a container-managed JTA transaction before performing an EntityManager operation.

EntityManagerFactory

As you can see, we get an instance of an application-managed EntityManager using the EntityManagerFactory interface. If you have used JDBC, this is essentially the same idea as creating a Connection from a DriverManager factory. In a Java EE environment you have the luxury to use the @PersistenceUnit annotation to inject an instance of an EntityManagerFactory, just as we do in Listing 9.2#1. This useful annotation is defined as follows:

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceUnit {
    String name() default "";
```

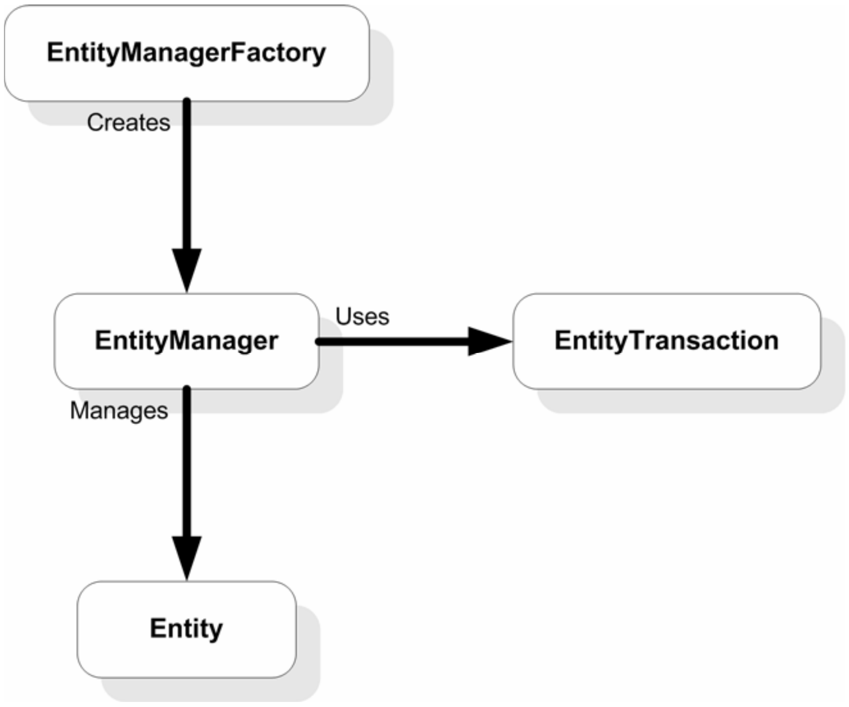
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    String unitName() default "";
}
```

The `name` and `unitName` elements serve exactly the same purpose as they do for the `@PersistenceContext` annotation. While the `name` element can be used to point to the JNDI name of the `EntityManagerFactory`, the `unitName` element is used to specify the name of the underlying persistence unit.

Figure 9.5 shows the relationships between important interfaces made available by JPA in outside the container that we will discuss in this section.



57 **Figure 9.5: Relataionships between different important classes in javax.persistence package for using JPA outside Java EE container.**

The `EntityManagerFactory`'s `createEntityManager()` method creates an application-managed `EntityManager`. This is probably the most commonly used method in the interface, in addition to the `close` method. We don't explicitly close the factory in Listing 9.2 since the container takes care of cleaning up all resources it injects (unlike the `EntityManager`, which is created programmatically and is explicitly closed in #4). Table 9.2 lists all the methods in the `EntityManagerFactory` interface. As you can see, most of them are fairly self-explanatory.

1.18 Table 9.2 EntityManager factory is used to create an instance of application-managed EntityManager

Method	**Purpose**
EntityManager createEntityManager()	Creates an application-managed EntityManager.
EntityManager createEntityManager(Map map)	Creates an application-managed EntityManager with a specified Map. The Map contains vendor specific properties to create the manager.

Licensed to John Sweitzer <admin@saolailaem.info>

void close()	Closes the EntityManagerFactory.
Boolean isOpen()	Checks whether the EntityManagerFactory is open.

Perhaps the most interesting aspect of Listing 9.2 is the `entityManager.joinTransaction()` call in the `updateItem` method#3. Let's discuss this method in just a little more detail.

As we hinted in the beginning of the Section, unlike container-managed `EntityManagers`, application-managed `EntityManagers` do not automatically participate in an enclosing transaction. Instead, they must be asked to join an enclosing JTA transaction by calling the `joinTransaction` method. This method is specifically geared to using application-managed `EntityManagers` inside a container, where JTA transactions are usually available.

Application Managed EntityManager outside Java EE container

In Java SE environments on the other hand, JTA is not a possibility. Resource local transactions must be used in place of JTA for such environments. The `EntityTransaction` interface is designed with exactly this scenario in mind. We will explore this interface by re-implementing the code to update an item from Listing 9.2 for an SE application. Listing 9.3 also serves the dual purpose of being a good template for using application-managed `EntityManagers` without any help from the container.

Listing 9.3: Using an Application-Managed EntityManager Outside a Container

```
EntityManagerFactory entityManagerFactory =                         |#1
    Persistence.createEntityManagerFactory("actionBazaar");         |#1

EntityManager entityManager =                                       |#2
    entityManagerFactory.createEntityManager();                     |#2

try
{

   EntityTransaction entityTransaction =
|#3
    entityManager.getTransaction();                                 |#3

   entityTransaction.begin();
|#4

   entityManager.merge(item);
|#5

   entityTransaction.commit();    |#6

  entityManager.close();
|#7
  entityManagerFactory.close();
|#7
}
catch (Exception e) {}
finally
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
{
   entityManager.close();
   entityManagerFactory.close();
}
```

(annotation) <#1 Get EntityManagerFactory Instance>
(annotation) <#2 Create EntityManager>
(annotation) <#3 Create Transaction>
(annotation) <#4 Begin Transaction>
(annotation) <#5 Merge Item>
(annotation) <#6 Commit Transaction>
(annotation) <#7 Close Resources>

The first thing that should strike you about Listing 9.3 is the amount of boilerplate code involved to accomplish exactly the same thing as the `updateItem` method in Listing 9.1 (cutting and pasting both code snippets into a visual editor and comparing the code side-by-side might be helpful in getting the full picture at a glance).

The `Persistence` object's `createEntityManagerFactory` method used in Listing 9.3#1 is essentially a programmatic substitute for the `@PersistenceUnit` annotation. The single parameter of the `createEntityManagerFactory` method is the name of the `EntityManagerFactory` defined in the `persistence.xml` deployment descriptor. Since the container is no longer helping us out, it is now very important to make sure to close the `EntityManagerFactory` returned by the `createEntityManagerFactory` method when we are done, in addition to closing the `EntityManager`#7. Just like in Listing 9.2, the `EntityManagerFactory`'s `createEntityManager` method is still used to create the application-managed `EntityManager`#2.

However, before merging the `Item` Entity to the database#5, we now create an `EntityTransaction` by calling the `getTransaction` method of the `EntityManager#3`. The `EntityTransaction` is essentially a high-level abstraction over a resource-local JDBC transaction, as opposed to the distributed JTA transaction we *joined* in Listing 9.2. On first blush it is very natural to think that a `joinTransaction` call is still necessary to make the `EntityManager` aware of the enclosing transaction. Remember that since the `EntityManager` itself is creating the transaction instead of the container, it implicitly keeps track of `EntityTransactions`, so the join is not necessary. The rest of the transaction code, namely the begin#4 and the commit#6 do exactly what you would expect. As might be obvious, the `EntityTransaction` interface also has a rollback method to abort the transaction. Note application-managed `EntityManagers` are never transaction-scoped. That is, they keep managing attached Entities until they are closed. Also the `transaction-type` must be set to `RESOURCE_LOCAL` in the `persistence.xml` file for using `EntityTransaction` interface (we will discuss this further in Chapter 11 when we talk about EJB 3.0 packaging and deployment).

Although there is little doubt that the code in Listing 9.3 is pretty verbose and error prone, being able to use application-managed `EntityManagers` outside the confines of the container accomplishes the vital goal of making standardized ORM accessible to all kinds of applications beyond server-side enterprise solutions, including Java Swing based desktop apps as well as enabling integration with web container such as Tomcat or Jetty.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Using JPA in a web container and ThreadLocal Pattern

If you are using application managed entity manager in a web container such as Tomcat or Jetty some persistence provider such as Hibernate recommends you to use ThreadLocal pattern. This is widely known as ThreadLocal session pattern in the Hibernate community. It associates a single instance of EntityManager with a particular request. You have to bind the EntityManager to a thread local variable and set the EntityManager instance to the associated thread as in the following example:

```
private static EntityManagerFactory entityManagerFactory;
    public static final ThreadLocal<EntityManager> _threadLocal = new
ThreadLocal<EntityManager>(); |#1

    public static EntityManagerFactory getEntityManagerFactory() {
        if (entityManagerFactory == null) {
              entityManagerFactory =
Persistence.createEntityManagerFactory("actionBazaar");
        }

        return entityManagerFactory;
    }


    public static EntityManager getEntityManager() {
        EntityManager entityManager = _threadLocal.get();

        if (_threadLocal == null) {
           entityManager = entityManagerFactory.createEntityManager();
|#2
           _threadLocal.set(entityManager);        |#3
        }
        return entityManager;
    }
```

(annotation) <#1 Store EM in ThreadLocal variabke>
(annotation) <#2 Create EntityManager>
(annotation) <#3 Associate EM with a thread>

Check documentation of your persistence provider if it requires you to use ThreadLocal pattern.

This coverage of application-managed `EntityManagers` brings us to the end of the topic of creating `EntityManager` instances. In the next Sections, we are going to tackle the most important part of this Chapter, `EntityManager` operations.

9.3 Managing Persistence Operations

The heart of the JPA API lies in the `EntityManager` operations we will discuss in the coming Sections. As you might have noted in Listing 9.1, although the `EntityManager` interface is small and simple, it is pretty complete in its ability to provide an effective persistence infrastructure. In addition to the CRUD (Create, Read, Update and Delete) functionality we lightly introduced in Listing 9.1, we will also cover a few less-commonly used operations like flushing and refreshing.

We'll start our coverage in the most logical place, persisting new Entities into the database.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

9.3.1 Persisting Entities

Recall that in Listing 9.1, the `addItem` method persists an `Item` Entity into the database. Since Listing 9.1 was quite a few pages back, we will repeat the `addItem` method body as review in Listing 9.4. Although it is not obvious, the code is especially helpful in understanding how Entity relations are persisted, which we will look at in greater detail in a minute. For now, we will concentrate on the `persist()` method itself:

Listing 9.4: Persisting Entities

```
public Item addItem(String title, String description,
    byte[] picture, double initialPrice, long sellerId) {
    Item item = new Item();
    item.setTitle(title);
    item.setDescription(description);
    item.setPicture(picture);
    item.setInitialPrice(initialPrice);
    Seller seller = entityManager.find(Seller.class, sellerId);
    item.setSeller(seller);
    entityManager.persist(item);                                    |#1

    return item;
}
```
(annotation) <#1 Persist Entity>

A new `Item` Entity corresponding to the record being added is first instantiated in the `addItem` method. All of the relevant `Item` Entity data to be saved into the database, such as the item title and description, are then populated with the data passed in by the user. As we recall from Chapter 7, the `Item` Entity has a many to one relationship with the `Seller` Entity. The related seller is retrieved using the `EntityManager find` method and set as a field of the `Item` Entity. The `persist` method is then invoked to save the Entity into the database as depicted in Figure 9.6. Note the `persist` method is intended to create *new* Entity records in the database, not update existing ones.

This means that you should make sure the identity or primary key of the Entity to be persisted does not already exist in the database.

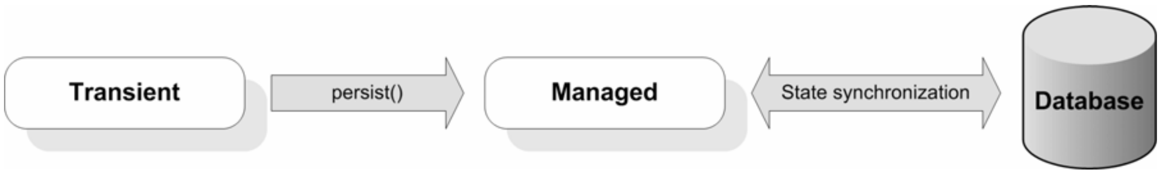


Figure 9.6: Invoking the persist() method on EntityManager interface makes an entity instance managed and its state is synchronized when transaction is committed

If you try to persist an Entity that violates the database's integrity constraint, the persistence provider will throw `javax.persistence.PersistenceException`.

As we noted earlier, the `persist` method also causes the Entity to become managed as soon as the method returns. The `INSERT` statement (or statements) to create the record corresponding to the Entity is not necessarily issued immediately. For transaction-scoped `EntityManagers`, the statement is typically issued when the enclosing transaction is about to commit. In our example, this means the SQL statements are issued when the `addItem` method returns. For extended-scoped (or

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

application-managed) EntityManagers, the INSERT statement is probably issued right before the EntityManager is closed. The INSERT statement can also be issued at any point when the EntityManager is *flushed*. We will discuss automatic and manual flushing in more detail in just a few Sections. For now, you just need to know that under certain circumstances, either the EntityManager or you can choose to perform pending database operations (like an INSERT to create a record), without waiting for the transaction to end or the EntityManager to close. The INSERT statement corresponding to Listing 9.4 to save the Item Entity could look something like the following:

```
INSERT INTO ITEMS VALUES (TITLE, DESCRIPTION, SELLER_ID, ...) VALUES
("Toast with the face of Bill Gates on it",
    "This is an auction for...", 1, ...)
```

An interesting thing to note here is that the ITEM_ID primary key that is the *Identity* for the Item Entity is not included in the generated INSERT statement. This is because the key generator scheme for the itemId Identity field of the Entity is set to IDENTITY. If the key generation scheme was set to SEQUENCE or TABLE instead, the EntityManager would have generated a SELECT statement to retrieve the key value first and then include the retrieved key in the INSERT statement. As we mentioned in Section 9.3, all persistence operations that require database updates must be invoked within the scope of a transaction. If an enclosing transaction is not present when the persist method is invoked, a TransactionRequiredException exception is thrown. The same is true for the EntityManager flush, merge, refresh and remove methods we will discuss shortly.

Persisting Entity Relations

One of the most interesting aspects of persistence operations is the handling of Entity relations. JPA gives us a number of options to handle this nuance in a way that suits a particular application-specific situation.

Let's explore these options by revisiting Listing 9.4. The addItem method is one of the simplest cases of persisting Entity relations. The Seller Entity is retrieved using the find method, so it is already managed and any changes to it are guaranteed to be transparently synchronized. Recall from Chapter 7 that there is a bidirectional one-to-many relationship between the Item and Seller Entities. This relationship is realized in Listing 9.4 by setting the Seller using the item.setSeller method. Let's assume that the Seller Entity is mapped to the SELLER table. Such a relationship between the Item and Seller Entities is likely implemented through a foreign key to the SELLER.SELLER_ID column in the ITEMS table. Since the Seller Entity is already persisted, all the EntityManager has to do is to set the SELLER_ID foreign key in the generated INSERT statement. Examining the INSERT statement presented earlier, this is how the SELLER_ID value is set to 1. Note if the seller property of Item was not set at all, the SELLER_ID column in the INSERT statement would be set to NULL.

Things become a lot more interesting when we consider the case when the Entity related to the one we are persisting does not exist in the database yet. This situation does not happen very often for one-many and many-many relations. In such cases, the related Entity is more than likely already saved in the database. However, it does occur a lot more often for one-to-one relations. For purposes of meaningful illustration, we will stray from our ItemManager example and take a look at saving

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

User Entities with associated BillingInfo Entities. Recall that we introduced this unidirectional one-to-one relation in Chapter 7. The method outlined in Listing 9.5 receives user information such as username, email as well as billing information such as credit card type, credit card number and persists both the User and related BillingInfo Entities into the database.

Listing 9.5: Persisting Relations

```
public User addUser(String username, String email,
    String creditCardType, String creditCardNumber,
        Date creditCardExpiration) {
    User user = new User();
    user.setUsername(username);
    user.setEmail(email);

    BillingInfo billing = new BillingInfo();
    billing.setCreditCardType(creditCardType);
    billing.setCreditCardNumber(creditCardNumber);
    billing.setCreditCardExpiration(creditCardExpiration);

    user.setBillingInfo(billing);
    entityManager.persist(user);                                    |#1

    return user;
}
```

(annotation) <#1 Persist both User and BillingInfo>

As we see, neither the User Entity nor the related BillingInfo Entity is managed when the persist method is invoked since both are newly instantiated. Let us assume for the purpose of this example that the User and BillingInfo Entities are saved into the USERS and BILLING_INFO tables, with the one-to-one relation modeled with a foreign key on the USERS table referencing the BILLING_ID key in the BILLING_INFO table. As you might guess from looking at Listing 9.5, two INSERT statements, one for the User and the other for the BillingInfo Entity, are issued by JPA. The INSERT statement on the USERS table will contain the appropriate foreign key to the BILLING_INFO table.

Cascading Persist Operations

Believe it or not, it is not the default behavior for JPA to persist related Entities. By default the BillingInfo Entity would not be persisted and you would not see an INSERT statement generated to persist the BillingInfo Entity into BILLING_INFO table. The key to understanding why this is not what happens in Listing 9.5 lies in the @OneToOne annotation on the billing property of the User Entity:

```
public class User {

@OneToOne(cascade=CascadeType.PERSIST)
    public void setBillingInfo(BillingInfo billing) {
```

Notice the value of the cascade element of the @OneToOne annotation. We deferred the discussion of this element in Chapter 7 so that we could discuss it in a more relevant context here.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Cascading in ORM based persistence is very similar to the idea of cascading in databases. The `cascade` element essentially tells the `EntityManager` how or if to propagate a given persistence operation on a particular Entity into Entities related to it.

By default, the cascade element is empty, which means that *no* persistence operations are propagated to related Entities. Alternatively, the cascade element can be set to `ALL`, `MERGE`, `PERSIST`, `REFRESH` and `REMOVE`. Table 9.3 lists the effect of each of these values.

1.19 Table 9.3: Effects of various cascade type values.

CascadeType Value	Effect
CascadeType.ALL	All EntityManager operations are propagated to related Entities.
CascadeType.MERGE	Only EntityManager.merge operations are propagated to related Entities.
CascadeType.PERSIST	Only EntityManager.persist operations are propagated to related Entities.
CascadeType.REFRESH	Only EntityManager.refresh operations are propagated to related Entities.
CascadeType.REMOVE	Only EntityManager.remove operations are propagated to related Entities.

Since in our case we have set the `cascade` element to `PERSIST`, when we `persist` the `User` Entity, the `EntityManager` figures out that a `BillingInfo` Entity is associated with the `User` Entity and it must be persisted as well. As Table 9.3 indicates, the `persist` operation would still be propagated to `BillingInfo` if the `cascade` element were set to `ALL` instead. However, if the element was set to any other value or not specified the operation would not be propagated and we would have to perform the `persist` operation on the `BillingInfo` Entity separately from the `User` Entity. For example, let us assume that the `cascade` element on the `@OneToOne` annotation is not specified. In order to make sure both related Entities are persisted, the `addUser` method would have to change as to look like Listing 9.6.

Listing 9.6: Manually Persisting Relations

```
public User addUser(String username, String email,
    String creditCardType, String creditCardNumber,
        Date creditCardExpiration) {
    User user = new User();
    user.setUsername(username);
    user.setEmail(email);

    BillingInfo billing = new BillingInfo();
    billing.setCreditCardType(creditCardType);
    billing.setCreditCardNumber(creditCardNumber);
    billing.setCreditCardExpiration(creditCardExpiration);

    entityManager.persist(billing);                           |#1

    user.setBillingInfo(billing);
    entityManager.persist(user);                              |#2

    return user;
}
```

(annotation) <#1 Persist BillingInfo>
(annotation) <#2 Persist User>

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

As you can see, the BillingInfo Entity is persisted first. The persisted BillingInfo Entity is then set as a field of the User Entity. When the User Entity is persisted, the generated key from the BillingInfo Entity is used in the foreign key for the generated INSERT statement.

Having explored the persist operation, let us now move on to the next operation in the EntityManager CRUD sequence—retrieving Entities.

9.3.2 Retrieving Entities by Primary Key

JPA supports several ways to retrieve Entity instances from the database. By far the simplest way is retrieving an Entity by its primary key using the find method we introduced in Listing 9.1. The other ways of retrieving Entities all involve using the query API and JPQL, which we will discuss in Chapter 10. Recall that the find method was used in the addItem method in Listing 9.1 to retrieve the Seller instance corresponding to the Item to add:

```
Seller seller = entityManager.find(Seller.class, sellerId);
```

The first parameter of the find() method specifies the Java type of the Entity to be retrieved. The second parameter specifies the *identity* value for the Entity instance to retrieve. Recall from Chapter 7 that an Entity Identity can either be a simple Java type identified by the @Id annotation or a composite primary key class specified through the @EmbededId or @IdClass annotation. In the example in Listing 9.1, the find method is passed a simple java.lang.Long value matching the Seller Entity's @Id annotated Identity, sellerId.

Although this is not the case in Listing 9.1, the find() method is fully capable of supporting composite primary keys. To see how this code might look like, let us assume for sake of illustration that the Identity of the Seller Entity consists of the seller's first and last name instead of a simple numeric identifier. This Identity is encapsulated in a composite primary key class annotated with the @IdClass annotation. Listing 9.7 shows how this Identity class can be populated and passed to the find method:

Listing 9.7: Find by Primary Key Using Composite Keys

```
SellerPK sellerKey = new SellerPK();                                  |#1

sellerKey.setFirstName(firstName);                                    |#1
sellerKey.setLastName(lastName);                                      |#1

Seller seller = entityManager.find(Seller.class, sellerKey);          |#2
```

(annotation) <#1 Creating and Setting Composite Key>
(annotation) <#2 Retrieving By Composite Key>

The find method does what it does by inspecting the details of the Entity class passed in as the first parameter and generating a SELECT statement to retrieve the Entity data. This generated SELECT statement is populated with the primary key values specified in the second parameter of the find method. For example, the find method in Listing 9.1 could generate a SELECT statement that looks something like the following:

```
SELECT * FROM seller WHERE seller_id = 1
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Note if an Entity instance matching the specified key does not exist in the database, the `find` method will not throw any exceptions. Instead, the `EntityManager` will return null or an empty Entity and your application must handle this situation. It is not strictly necessary to call the `find` method in a transactional context. However, the retrieved Entity is *detached* unless a transaction context is available, so it is generally advisable to call the `find` method inside a transaction. One of the most important features of the `find` method is that it utilizes `EntityManager` caching. If your persistence provider supports caching and the Entity already exists in the cache then the `EntityManager` returns a cached instance of Entity instead of retrieving it from the database. Most persistence providers like Hibernate and Oracle TopLink support caching, so you can more or less count on this extremely valuable optimization.

There is one more important JPA feature geared toward application optimization—lazy and eager loading. The generated `SELECT` statement in our example attempts to retrieve all of the Entity field data when the `find` method is invoked. In general this is exactly what will happen for Entity retrieval since it is the default behavior for JPA. However, in some cases, this is not desirable behavior. *Fetch modes* allow us to change this behavior to optimize application performance when needed.

Entity Fetch Modes

We briefly mentioned fetch modes in previous Chapters but never really discussed them in great detail. Discussing Entity retrieval is an ideal place to fully explore fetch-modes.

As we suggested, the `EntityManager` normally loads all Entity instance data when an Entity is retrieved from the database. In ORM-speak this is called *eager fetching or eager loading*. If you have ever dealt with application performance problems due to premature or inappropriate caching, you probably already know that eager fetching is not always a good thing. The classic example we used in previous Chapters is loading large binary objects (BLOB), such as pictures. Unless you are developing a heavily graphics-oriented program such as an online photo album, it is very unlikely that loading a picture as part of an Entity used in a lot of places in the application is a very good idea. Because loading BLOB data typically involves long running, I/O-heavy operations, they should be loaded cautiously and only as needed. In general, this optimization strategy is called *lazy fetching*.

JPA has more than one mechanism to support lazy fetching. Specifying column fetch-mode using the `@Basic` annotation is the easiest one to understand. For example, we can set the fetch-mode for the `picture` property on the `Item` Entity to be lazy as follows:

```
@Column(name="PICTURE")
@Lob
@Basic(fetch=FetchType.LAZY)
public byte[] getPicture() {
    return picture;
}
```

A `SELECT` statement generated by the `find` method to retrieve `Item` Entities would not load data from the `ITEMS.PICTURE` column into the `picture` field. Instead, the `picture` data will be automatically loaded from the database when the property is first accessed through the `getPicture` method.

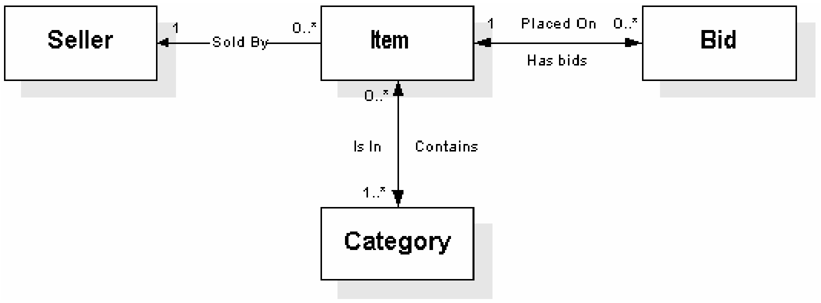
Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Be advised, however, that lazy-fetching is a double-edged sword. Specifying that a column be lazily fetched means that the `EntityManager` will issue an additional `SELECT` statement just to retrieve the `picture` data when the lazily loaded field is first accessed. In the extreme case, imagine what would happen if all Entity data in an application is lazily loaded. This would mean that the database would be flooded with a very large number of frivolous `SELECT` statements as Entity data is accessed. Also, lazy-fetching is an optional EJB 3.0 feature, which means not every persistence provider is guaranteed to implement it. You should check your provider's documentation before spending too much time figuring out which Entity columns should be lazily fetched.

Loading Related Entities

One of the most intricate uses of fetch modes is to control the retrieval of related Entities. Not too surprisingly, the `EntityManager find` method must retrieve all Entities related to the one returned by the method. Let's take the ActionBazaar `Item` Entity. The `Item` Entity is an exceptionally good case because it has a many-to-one, a one-to-many and two many-to-many relations. The only relationship type not represented in the `Item` Entity is one-to-one. The `Item` Entity has a many-to-one relation with the `Seller` Entity (a seller can sell more than one item, but an item can be sold by only one seller), a one-to-many relation with the `Bid` Entity (more than one bid can be put on an item) and a many-to-many relation with the `Category` Entity (an item can belong to more than one category and a category contains multiple items). These relations are depicted in Figure 9.7.



59 **Figure 9.7: The Item Entity is related to three other Entities, Seller, Bid and Category. The relationships to Item are ManyToOne, OneToMany and ManyToMany respectively.**

When the `find` method returns an instance of an `Item`, it also automatically retrieves the `Seller`, `Bid` and `Category` Entities associated with the instance and populates them into their respective `Item` Entity properties. As we see in Listing 9.8, the single `Seller` Entity associated with the `Item` is populated into the `seller` property, the `Bid` Entities associated with an `Item` are populated into the `bids List` and the `Category` Entities the `Item` is listed under are populated into the `categories` property. It might surprise you to know some of these relations are retrieved lazily.

All the relationship annotations we saw in Chapter 8, including the `@ManyToOne`, `@OneToMany` and `@ManyToMany` annotations have a `fetch` element to control fetch-modes just like the `@Basic` annotation discussed in the previous Section. None of the relationship annotations in Listing 9.8 specify the `fetch` element, so the default for each annotation takes effect.

34

35 Listing 9.8: Relationships in the Item Entity

```
public class Item {
```

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @ManyToOne                                                    |#1
    public Seller getSeller(){
    ...

    @OneToMany                                                    |#2
    public List<Bid> getBids(){
    ...

    @ManyToMany                                                   |#3
    public List<Category> getCategories(){
    ...
}
```
(annotation) <#1 Many-to-One Relation with Seller>
(annotation) <#2 One-to-Many Relation with Bids>
(annotation) <#3 Many-to-Many Relation with Categories>

By default, some of the relationship types are retrieved lazily while some are loaded eagerly. We will discuss why each default makes sense as we go through each relationship retrieval case for the `Item` Entity. The `Seller` associated to an `Item` is retrieved eagerly, because the fetch-mode for the `@ManyToOne` annotation is defaulted to `EAGER#1`. To understand why this is sensible, it is very helpful to understand how the `EntityManager` implements eager fetching. In effect, each eagerly fetched relation turns into an additional `JOIN` tacked onto to the basic `SELECT` statement to retrieve the Entity. To see what we mean, let us take a look at how the `SELECT` statement for an eagerly fetched `Seller` record related to an `Item` looks like:

Listing 9.9: SELECT Statement for Eagerly Fetched Seller Related to an Item
```
SELECT
    *
FROM
    ITEMS
INNER JOIN                                                        |#1
    SELLER
ON
    ITEM.SELLER_ID = SELLER.SELLER_ID
WHERE ITEM.ITEM_ID = 100
```

(annotation) <#1 Inner Join For Many-to-One Relation>

As the Listing shows, an eager fetch means that the most natural way of retrieving the `Item` Entity would be through a single `SELECT` statement using a `JOIN` between the `ITEMS` and `SELLER` tables. It is important to note the fact that the `JOIN` will result in a *single row,* containing columns from both the `SELLER` and `ITEMS` tables. In terms of database performance, this is more efficient than issuing one `SELECT` to retrieve the `Item` and issuing a separate `SELECT` to retrieve the related `Seller`. This is exactly what would have happened in case of lazy fetching and the second `SELECT` for retrieving the `Seller` will be issued when the `Item`'s `seller` property is first accessed. Pretty much the same thing applies to the `OneToOne` annotation so the default for it is also eager loading. More specifically, the `JOIN` to implement the relation would result in a fairly efficient single combined row in all cases.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Lazy vs. Eager Loading of related Entities

In contrast, the `@OneToMany` and `@ManyToMany` annotations are defaulted to *lazy-loading*. The critical difference is that for both of these relationship types, *more than one* Entity is matched to the retrieved Entity. Think about `Category` Entities related to a retrieved `Item` for example. `JOINs` implementing eagerly loaded `OneToMany` and `ManyToMany` relations usually return more than one row. In particular, a row is returned for every related Entity matched.

The problem becomes particularly obvious when you consider what happens when multiple `Item` Entities are retrieved at one time (for example as the result of a JPQL query, discussed in the next Chapter). (N_1 + N_2 + … + N_x) rows would be returned, where N_i is the number of related `Category` Entities for the i^{th} `Item` record. For non-trivial numbers of N and i, the retrieved result set could be quite large, potentially causing significant database performance issues. This is why the JPA makes the conservative assumption of defaulting to lazy loading for `@OneToMany` and `@ManyToMany` annotations.

Table 9.4 lists the default fetch behavior for each type of relationship annotation.

Table 9.4: Behavior of loading of associated entity is different for different kind of associations by default. We can change the loading behavior by specifying fetch element with the association.

Relationship Type	**Default Fetch Behavior**	**Number of Entities Retrieved**
OneToOne	EAGER	Single Entity Retrieved
OneToMany	LAZY	Collection of Entities Retrieved
ManyToOne	EAGER	Single Entity Retrieved
ManyToMany	LAZY	Collection of Entities Retrieved

The relationship defaults are not right for all circumstances, however. While the eager fetching strategy makes sense for `@OneToOne` and `@ManyToOne` relations under most circumstances, they are a bad idea in some cases. For example, if an Entity contains a large number of one-to-one and many-to-one relationships, eagerly loading all of them would result in a large number of `JOINs` chained together. Executing a relatively large number of joins can be just as bad as loading an (N_1 + N_2 + … + N_x) results set. If this proves to be a performance problem, some of the relations should be loaded lazily. The following is an example of explicitly specifying the fetch mode for a relation (it happens to be the familiar `Seller` property of the `Item` Entity):

```
@ManyToOne(fetch=FetchType.LAZY)
public Seller getSeller() {
    return seller;
}
```

You should not take the default lazy loading strategy of the `@OneToMany` and `@ManyToMany` annotations for granted either. For particularly large data sets, this can result in a very big number of `SELECTs` being generated against the database. This is known as the N+1 problem, where 1 stands for the `SELECT` statement for the originating Entity and where N stands for SELECT statement to retrieve each related Entity. In some cases, you might find out that you are better off using eager loading even for `@OneToMany` and `@ManyToMany` relations. In Chapters 10 and 13, we will discuss how you can eagerly load related Entities without having to change the fetch mode on an association on a per-query basis.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Unfortunately, the choice of fetch modes is not cut and dry and depends on a whole host of factors including the database vendor's optimization strategy, database design, data volume and application usage patterns. In the real world, ultimately these choices are often made through trial and error. Luckily, with JPA, performance tuning just means a few configuration changes here and there as opposed to time-consuming code modifications.

Having discussed Entity retrieval, we can now move into the third operation of the CRUD sequence, updating Entities.

9.3.3 Updating Entities

Recall that the `EntityManager` makes sure that changes made to attached Entities are always saved into the database behind the scenes. This means that for the most part, our application does not need to worry about manually calling any methods to update the Entity. This is perhaps the most elegant feature of ORM based persistence since this hides data synchronization behind the scenes and truly allows Entities to behave like POJOs. Take the code in Listing 9.10 that calculates an ActionBazaar power-seller's creditworthiness for example:

Listing 9.10: Transparent Management of Attached Entities

```
public void calculateCreditWorthiness (Long sellerId) {
    PowerSeller seller = entityManager.find(                          |#1
        PowerSeller.class, sellerId);                                 |#1

    seller.setCreditWorth(seller.getCreditWorth()                     |#2
        * CREDIT_FACTOR                                               |#2
            * getRatingFromCreditBureauRobberBarons(seller));         |#2
    seller.setCreditWorth(seller.getCreditWorth()                     |#2
        + (seller.getCreditWorth()                                    |#2
            * FEEDBACK_FACTOR                                         |#2
                * seller.getBuyerFeedbackRating()));                  |#2
    seller.setCreditWorth(seller.getCreditWorth()                     |#2
        + (seller.getCreditWorth()                                    |#2
            * SELLING_FACTOR                                          |#2
                * getTotalHistoricalSales(seller)));                  |#2
}
```

(annotation) <#1 Find Seller Entity>
(annotation) <#2 Changes to Entity Transparently Saved>

Other than looking up the `PowerSeller` Entity, little else is done using the `EntityManager` in the `calculateCreditWorthiness` method. As you know this is because the `PowerSeller` is managed by the `EntityManager` as soon as it is returned by the `find` method. Throughout the relatively long calculation for determining creditworthiness, the `EntityManager` will make sure that the changes to the Entity wind up in the database.

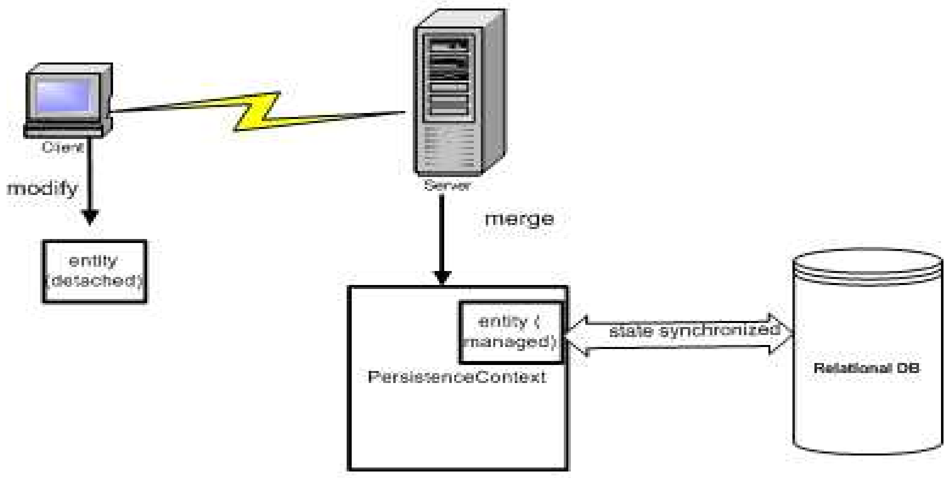
Detachment and Merge operations

Although managed Entities are extremely useful, the problem is that it is difficult to keep Entities attached at all times. Often the problem is that the Entities will need to be detached and serialized at the web tier, where the Entity is changed, outside the scope of the `EntityManager`. In addition, recall that Stateless Session Beans cannot guarantee that calls from the same client will be services by

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

the same bean instance. This means that an Entity cannot be guaranteed to be handled by the same `EntityManager` instance across method calls, making automated persistence ineffective.



60 **Figure 9.8: An entity instance can be detached and serialized to a separate tier where client makes changes to the entity and sends back to the server. The server can use merge operation to attach the entity to the persistence context.**

This is exactly the model the `ItemManager` Session Bean introduced in Listing 9.1 assumes. The `EntityManager` used for the bean has transactional scope. Since the bean uses CMT, Entities become detached when transactions end the end of the method. This means that Entities returned by the Session bean to its clients are always detached, just as the newly created `Item` Entity returned by the `ItemManager`'s `addItem` method:

```
public Item addItem(String title, String description,
    byte[] picture, double initialPrice, long sellerId) {
    Item item = new Item();
    item.setTitle(title);
    ...
    entityManager.persist(item);

    return item;
}
```

At some point, we will want to reattach the detached Entity to a persistence context to synchronize it with the database. This is exactly what the `EntityManager merge` method is designed to do. Figure 9.8 depicts the merge operation.You should remember that like all attached Entities, the Entity passed to the `merge()` method is not necessarily synchronized with the database immediately, but it is guaranteed to be synchronized with the database at some point. We use the `merge` method in the `ItemManager` bean in the most obvious way possible, to update the database with an existing `Item`:

```
public Item updateItem(Item item) {
    entityManager.merge(item);
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    return item;
}
```

As soon as the `updateItem` method returns, the database is updated with the data from the `Item` Entity. The `merge` method must only be used for an Entity that exists in the database. An attempt to merge a non-existent Entity will result in an `IllegalArgumentException.` The same is true if the `EntityManager` detects that the Entity you are trying to `merge` has already been deleted through the `remove` method, even if the DELETE statement has not been issued yet.

Merging Relations

By default, Entities associated to the Entity being merged are not merged as well. For example, the `Seller`, `Bid` and `Category` Entities related to the `Item` are not merged when the `Item` is merged in the previous code snippet. However, as mentioned in section 9.3.1, this behavior can be controlled using the `cascade` element of the `OneToOne`, `OneToMany`, `ManyToOne` and `ManyToMany` annotations. If the element is set to either `ALL` or `MERGE`, the related Entities are merged. For example, the following code will cause the `Seller` Entity related to the `Item` to be merged since the cascade element is set to `MERGE`:

```
public class Item {
    @ManyToOne(cascade=CascadeType.MERGE)
    public Seller getSeller() {
```

Note, like most of the `EntityManager`'s methods, the `merge` method must be called from a transactional context or it will throw a `TransactionRequiredException`.

We will now move onto the final element of the CRUD sequence, *deleting* an Entity.

Detached Entities and the DTO Anti-Pattern

If you have spent even a moderate amount of time using EJB 2.x you are probably thoroughly familiar with the Data Transfer Object (DTO) anti-pattern. In a sense, the DTO anti-pattern was really necessary because of Entity Beans. The fact that EJB 3.0 detached Entities are really nothing but POJOs makes the DTO anti-pattern less of a necessity of life. Instead of having to create separate DTOs from domain data just to pass back and forth between the business and presentation layers, you may simply pass detached Entities. This is exactly the model we follow in this Chapter.

However, if your Entities contain behavior, you might be better off using the DTO pattern anyway, to safeguard business logic from inappropriate usage outside a transactional context. In any case, if you decide to use detached Entities as a substitute to DTOs, you should make sure they are marked java.io.Serializable.

9.3.4 Deleting Entities

The `deleteItem` method in the `ItemManagerBean` in Listing 9.1 deletes an `Item` from the database. An important detail to notice about the `deleteItem` method (repeated next) is that the Item to be deleted was first attached to the `EntityManager` using the `merge` method:

```
public void deleteItem(Item item) {
    entityManager.remove(entityManager.merge(item));
}
```

This is because the `remove()` method can only delete *currently attached* Entities and the `Item` Entity being passed to the `deleteItem` method is not managed. If a detached Entity is passed to

Licensed to John Sweitzer <admin@saolailaem.info>

the remove method, it throws an IllegalArgumentException. Before the deleteItem method returns, The Item record will be deleted from the database using a DELETE statement like the following:

```
DELETE FROM items WHERE item_id = 1
```

Just as with the persist and merge methods, the DELETE statement is not necessary issued immediately but is guaranteed to be issued at some point. In the meanwhile, the EntityManager marks the Entity as *removed* so that no further changes to it are synchronized (as we noted in the previous Section).

Cascading Remove Operations

Just as in merging and persisting Entities, you must set the cascade element of a relationship annotation to either ALL or REMOVE for related Entities to be removed with the one passed to the remove method. For example, we can specify that the BillingInfo Entity related to a Bidder be removed with the owning Bidder Entity as follows:

```
@Entity
public class Bidder {
    @OneToOne(cascade=CascadeType.REMOVE)
    public BillingInfo setBillingInfo() {
```

From a common sense perspective, this setup makes perfect sense. There is no reason for a BillingInfo Entity to hang around if the enclosing Bidder Entity it is related to is removed. When it comes down to it, the business domain really determines if deletes should be cascaded. In general, you might find that the only relationship types where cascading removal makes sense are one-to-one and one-to-many relations. You should be very careful when using cascade delete because the related Entity you are cascading the delete to may be related to other Entities you don't know about. For example, let's assume that there is a one-to-many relationship between Seller and Item Entities and you are using cascade delete to remove a Seller and the related the Items. Remember the fact that other Entities such as the Category Entity also hold references to the Items you are deleting and these relationships would become meaningless!

Handling Relations

If your intent was really to cascade delete the Items associated with the Seller, you should iterate over all the Categories that hold reference to the deleted Items and remove the relationships first. The following code does this:

```
List<Category> categories = getAllCategories();
List<Item> items = seller.getItems();
for (Item item: items) {
    for (Category category: categories) {
        category.getItems().remove(item);
    }
}
entityManager.remove(seller);
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

The code gets all `Categories` in the system and makes sure that all `Items` related to the `Seller` being deleted are removed from referencing `Lists` first. It then proceeds with removing the `Seller`, cascading the remove to the related `Items`.

Not surprisingly, the `remove` method must be called from a transactional context or it will throw a `TransactionRequiredException`. Also, trying to remove an already removed Entity will raise `IllegalStateException`.

Having finished the basic `EntityManager` CRUD operations, we will now move on to the two remaining major persistence operations–flushing data to the database and refreshing from the database.

9.3.5 Controlling Updates with Flush

We've been talking about `EntityManager` flushing on and off throughout the Chapter. It is time we discussed this concept fully.
For the most part, you will probably be able to get away without knowing too much about this `EntityManager` feature. However, there are some important cases where not understanding `EntityManager` flushing could be a great disadvantage. Recall that `EntityManager` operations like `persist`, `merge` and `remove` do not cause immediate database changes. Instead, these operations are postponed until the `EntityManager` is *flushed*. The true motivation for doing things this way is performance optimization. Batching SQL as much as possible instead of flooding the database with a bunch of frivolous requests saves a lot of communication overhead and avoids unnecessarily tying down the database.

By default, the database flush mode is set to `AUTO`. This means that the `EntityManager` performs a flush operation automatically as needed. In general, this is done at the end of a transaction for `TRANSACTION` scoped `EntityManagers` and when the manager is closed for application-managed or `EXTENDED` scope `EntityManagers`. In addition, if Entities with pending changes are used in a query, the persistence provider will flush changes to the database before executing the query.

You can set the flush mode to `COMMIT` if you don't like the idea of auto-flushing and want greater control over database synchronization. You can do so using the `EntityManager setFlushMode` method as follows:

```
entityManager.setFlushMode(FlushModeType.COMMIT);
```

If the flush mode is set to `COMMIT`, the persistence provider will only synchronize with the database when the transaction commits. However, you should be very careful in doing this, as it will be your responsibility to synchronize Entity state with the database before executing a query.

If you do not do this and an `EntityManager` query returns stale Entities from the database, the application can wind up in an inconsistent state.

In reality, resetting flush mode is often overkill. This is because you can explicitly flush the `EntityManager` when you need to, using the `flush` method as follows:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

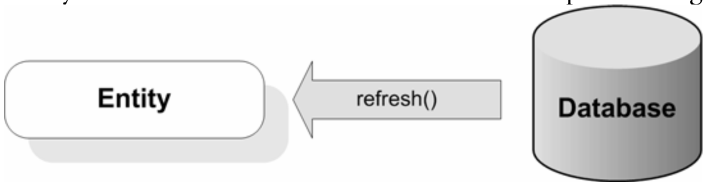
```
entityManager.flush();
```

The `EntityManager` synchronizes the state of every managed Entity with the database as soon as this method is invoked. Like everything else, manual flushing should be used in moderation and only when really needed. In general, batching database synchronization requests is a very good optimization strategy to try to preserve.

We will now move onto the last persistence operation we will discuss in this Chapter, refreshing Entities.

9.3.6 Refreshing Entities

The `refresh` operation re-populates Entity data from the database. In other words, given an Entity instance, the persistence provider matches the Entity with a record in the database and resets the Entity with retrieved data from the database as depicted in figure 9.9.



61 **Figure 9.9: Refresh operation repopulates the entity from database overriding any changes in the entity**

This `EntityManager` method is not used very frequently. However, there are some circumstances where it is extremely useful. In Listing 9.1, we use the method to undo changes made by the `ItemManager` client and return fresh Entity data from the database:

```
public Item undoItemChanges(Item item) {
    entityManager.refresh(entityManager.merge(item));
    return item;
}
```

The `merge` operation is performed first in the `undoItemChanges` method because the `refresh` method only works on managed Entities. It is extremely important to note that just like the `find` method, the `refresh` method uses the Entity Identity to match database records. As a result, you must make sure the Entity being refreshed exists in the database.

The `refresh` method really shines when you consider a subtle but very common scenario. To illustrate this scenario, let us go back to the `addItem` method in Listing 9.1:

```
public Item addItem(String title, String description,
    byte[] picture, double initialPrice, long sellerId) {
    Item item = new Item();
    item.setTitle(title);
    ...
    entityManager.persist(item);

    return item;
}
```

Note a subtle point about the method: it assumes that the `Item` Entity is not altered by the database in any way when the record is inserted into the database. It is easy to forget that this

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

is often never the case with relational databases. For most INSERT statements issued by the usual application, the database will fill-in column values not included in the INSERT statement using table defaults. For example, let us assume that the Item Entity has a postingDate property that is not populated by the application. Instead, this value is set to the current database system time when the ITEMS table record is inserted. This could be implemented in the database by utilizing default column values or even database triggers.

Since the persist method only issues the INSERT statement and does not load the data that was changed by the database as a result of the INSERT, the Entity returned by the method would not include the generated postingDate field. This problem could be fixed by using the refresh method as follows:

```
public Item addItem(String title, String description,
    byte[] picture, double initialPrice, long sellerId) {
    Item item = new Item();
    item.setTitle(title);
    ...
    entityManager.persist(item);
    entityManager.flush();
    entityManager.refresh(item);

    return item;
}
```

After the persist method is invoked, the EntityManager is flushed immediately so that the INSERT statement is executed and the generated values are set by the database. The Entity is then refreshed so that we get the most up-to-date data from the database and populate it into the inserted Item instance (including the postingDate field). In most cases you should try to avoid using default or generated values with JPA due to the slightly awkward nature of the code just introduced. Luckily, this awkward code is not necessary while using fields that use the JPA @GeneratedValue annotation since the persist method correctly handles such fields.

Before we wrap up this Chapter, we will introduce Entity life cycle based listeners.

9.4 Entity Lifecycle Listeners

We saw in earlier Chapters that both Session and Message Driven Beans allow us to listen for lifecycle callbacks like PostConstruct and PreDestroy. Similarly, Entities allow us to receive callbacks for life-cycle events like persist, load, update and remove. Just as in Session and Message Driven Beans, you can do almost anything you need to in the life-cycle callback methods, including invoking an EJB, or use APIs like JDBC or JMS. In the persistence realm, however, life-cycle callbacks are usually used to accomplish tasks like logging, validating data, auditing, sending notifications after a database change or generating data after an Entity has been loaded. In a sense, callbacks are the database triggers of JPA.

Table 9.6 lists the callbacks supported by the API:

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

1.20 Table 9.6: Callbacks supported by JPA and when they are called.

Lifecycle method	When it is performed
PrePersist	Before the EntityManager persists an Entity instance.
PostPersist	After an Entity has been persisted.
PostLoad	After an Entity has been loaded by a query, find or refresh operation.
PreUpdate	Before a database update occurs to synchronize an Entity instance.
PostUpdate	After a database update occurs to synchronize an Entity instance.
PreRemove	Before EntityManager removes an Entity.
PostRemove	After an Entity has been removed.

Entity life-cycle methods need not be defined in the Entity itself. Instead, you can choose to define a separate *entity listener* class to receive the life-cycle callbacks. We highly recommend this approach because defining callback methods in the Entities themselves will clutter up the domain model you might have carefully constructed. Moreover, Entity callbacks typically contain crosscutting concerns rather than business logic directly pertinent to the Entity. For our purposes, we will explore use of entity callbacks using separate listener classes, default listeners and execution order of entity listeners if you multiple listeners.

9.4.1 Using an Entity Listener

Let's take a look at how Entity life cycle callbacks look like by coding up an entity listener on the `Item` entity that notifies an ActionBazaar admin if an item's initial bid amount is set higher than a certain threshold. It is ActionBazaar policy to scrutinize items with extremely high initial prices to check against possible fraud, especially for items such as antiques and artwork. Listing 9.11 shows what the code looks like:

Listing 9.11: Item Entity Listener

```
public class ItemMonitor {
    ...
    public ItemMonitor() {}
    @PrePersist                                                      |#1
    @PreUpdate                                                       |#1
    public void monitorItem(Item item) {
        if (item.getInitialBidAmount() >
            ItemMonitor.MONITORING_THRESHOLD) {
            notificationManager.sendItemPriceEmailAlert(item);
        }
    }
}

@Entity
@EntityListeners(actionbazaar.persistence.ItemMonitor.class)         |#2
public class Item implements Serializable {
```

(annotation) <#1 Callbacks>
(annotation) <#2 Registering Listener>

As Listing 9.11 outlines our listener, `ItemMonitor`, has a single method `monitorItem` that receives callbacks for both the `PrePersist` and `PreUpdate` events. The `@EntityListeners` annotation on the `Item` Entity specifies `ItemMonitor` to be the life-cycle callback listener for the `Item` Entity. All we have to do to receive a callback is to annotate our method with a callback annotation such as `@PrePersist`, `@PostPersist`, `@PreUpdate` and so on. The `monitorItem` method checks to see if the initial bid amount set for the item to be inserted or updated is above the

Licensed to John Sweitzer <admin@saolailaem.info>

threshold specified by the `ItemMonitor.MONITORING_THRESHOLD` variable and sends the ActionBazaar admin an email alert if it is. As you might have guessed by examining Listing 9.11, Entity listener callback methods follow the form `void methodName(Object)`. The single method parameter of type `Object` specifies the Entity instance for which the life-cycle event was generated. In our case, this is the `Item` Entity.

If the life-cycle callback method throws a runtime exception, the intercepted persistence operation is aborted. This is an extremely important feature to validate persistence operations.

For example, if you have a listener class to validate that all Entity data is present before persisting an Entity, you could abort the persistence operation if needed by throwing a runtime exception.

Listing 9.12 shows an example listener class that can be used to validate an entity state before an entity is persisted. You can build validation logic in a PrePersist callback and if the callback fails then the entity will not be persisted. For example, ActionBazaar sets a minimum price for the initialPrice for items being auctioned and not items are allowed to be listed below that price.

Listing 9.12: ItemVerifier validates price set for an item

```
public class ItemVerifier{

…

    public ItemVerifier() {
    }
        @PrePersist
        public  void newItemVerifier(Item item){
            if (item.getInitialPrice()<MIN_INITIAL_PRICE)
                  throw new
ItemException("Item Price is lower than Minimum Price Allowed");
   }


}
```

Note all Entity listeners are stateless in nature and you cannot assume that there is a one-to-one relation between an Entity instance and a listener instance.

One great drawback of Entity listener classes is that they do not support dependency injection. This is due to the fact that entities may be used outside container, where DI is not available.

For crosscutting concerns like logging or auditing, it is really inconvenient to have to specify listeners for individual Entities. Keeping this problem in mind, JPA enables us to specify default Entity listeners that receive callbacks from all Entities in a persistence unit. We will take a look at this mechanism next.

9.4.2 Default Listener Classes

ActionBazaar audits all changes made to Entities. You can think of this as an ActionBazaar version of a transaction log. This feature can be implemented using a default listener like the following:

```
public class ActionBazaarAuditor {
    ...
```

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

```
    @PrePersist
    @PostPersist
    ...
    @PostRemove
    public void logOperation(Object object) {
         Logger.log("Performing Persistence Operation on: "
             + object.getName());
```

The `ActionBazaarAuditor` listens for all persistence operations for all Entities in the ActionBazaar persistence unit and logs the name of the Entity that the callback was generated on. Unfortunately, there is no way to specify default Entity listeners using annotations and we must resort to using the `persistence.xml` deployment descriptor. Since we have not yet fully described the persistence deployment descriptor, we will simply note the relevant descriptor snippet below, leaving a detailed analysis to Chapter 11:

```
<persistence-unit name="actionBazaar">
    ...
    <default-entity-listeners>
        actionbazaar.persistence.ActionBazaarAuditor.class
    </default-entity-listeners>
    ...
```

In the snippet above, the `default-entity-listeners` element lists the default Entity listeners for the "actionBazaar" persistence unit. Again, do not worry too much about the specific syntax at the moment, as we will cover it in greater detail later.

This brings us to the interesting question of what happens if there is both a default listener and an Entity specific listener for a given Entity, as in the case of our `Item` Entity. The `Item` Entity now has two life-cycle listeners, the default `ActionBazaarAuditor` listener and the `ItemMonitor` listener. How do you think they interact? Moreover, since Entities are POJOs that can inherit from other Entities, both the superclass and subclass may have attached listeners. For example, what if the `User` Entity has an attached listener named `UserMonitor` while the `Seller` subclass also has an attached listener, `SellerMonitor`. How these listeners relate to each other is determined by the order of execution as well as exclusion rules.

9.4.3 Listener Class Execution Order and Exclusion

If an Entity has default listeners, Entity class specific listeners and inherited superclass listeners, the default listeners are executed first. Following OO constructor inheritance rules, the superclass listeners are invoked after the default listeners. Subclass listeners are invoked last. Figure 9.10 depicts this execution order:

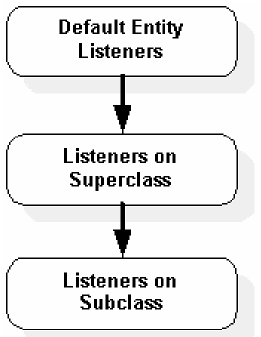


Figure 9.10: Entity Listener Execution Order. Default Entity Listeners are Executed First, then Superclass and Subclass Listeners.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

If there is more than one listener listed on any level, the execution order is determined by the order in which they are listed in the annotation or deployment descriptor. For example, in the following Entity listener annotation, the `ItemMonitor` listener is called before `ItemMonitor2`:

```
EntityListeners({actionbazaar.persistence.ItemMonitor.class,
    actionbazaar.persistence.ItemMonitor2.class})
```

You cannot programmatically control this order of execution. However, if needed, you can exclude default and superclass listeners from being executed at all. Let us assume for a second that we need to disable both default and superclass listeners for the `Seller` Entity. You can do this with the following code:

```
@Entity
@ExcludeDefaultListeners
@ExcludeSuperClassListeners
@EntityListeners(actionbazaar.persistence.SellerMonitor.class)
public class Seller extends User {
```

As you can see from the code, the `@ExcludeDefaultListeners` annotation disables any default listeners while the `@ExcludeSuperClassListeners` annotation stops superclass listeners from being executed. As a result, only the `SellerMonitor` listener specified by the `@EntityListeners` annotation will receive life-cycle callbacks for the `Seller` Entity. Unfortunately, neither the `@ExcludeDefaultListeners` nor the `@ExcludeSuperClassListeners` annotation currently enables us to block specific listener Classes. We will hope that this is a feature that will be added in a future version of JPA.

9.5 Best Practices

Throughput the Chapter, we have provided you some hints on the best practices of using the `EntityManager` interface. Before we conclude this Chapter, in this section we will solidify the discussion of best practices by discussing a few of the most important ones in detail.

Use Container Managed Entity Managers. If you are building an enterprise application that will be deployed to a Java EE container we strongly recommend that you use container-managed Entity managers. Furthermore if you are manipulating Entities from the Session Bean and MDB tier you should use declarative transactions in conjunction with container-managed `EntityManagers`. Overall, this will let you focus on application logic instead of worrying about the mundane details of managing transactions, managing EntityManager life cycles and so on.

Avoid InjectingEntity Managers into the Web Tier. If you are using the `EntityManager` API directly from a component in the web tier such as a Servlet, we recommend that you avoid injecting entity managers because the `EntityManager` interface is not thread-safe. Instead you should use a JNDI lookup to grab an instance of a container-managed `EntityManager`. Better yet, use the Session Facade pattern discussed in Chapter 12 instead of using `EntityManager` API directly from the web tier and take advantage of the benefits offered through Session Beans such as declarative transaction management.

Use the Entity Access Object Pattern. Instead of cluttering your business logic with `EntityManager` API calls, use a Data Access Object (we call it Entity Access Object) discussed in Chapter 12. This practice allows you to abstract the `EntityManager` API from the business-tier.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=273

Licensed to John Sweitzer <admin@saolailaem.info>

Separate Callbacks into External Listeners. Do not pollute your domain model with crosscutting concerns such as auditing and logging code. Use external Entity listener classes instead. This way, you could swap listeners in and out as needed.

9.6 Summary

In this Chapter, we covered the most vital aspect of JPA, persistence operations using Entity managers. We also covered persistence contexts, persistence scopes, various types of Entity managers and their usage patterns. We even briefly covered Entity lifecycles and listeners. We highly recommend Java EE container managed persistence contexts with CMT-driven transactions. In general, this strategy minimizes the amount of careful consideration of what is going on behind the scenes and consequent meticulous coding you might have to engage in otherwise. However, there are some valid reasons why you might want to use application-managed `EntityManagers`. In particular, the ability to use the `EntityManager` outside the confines of a Java EE container is an extremely valuable feature, especially to those of us using lightweight technologies like Apache Tomcat and the Spring Framework or even Java SE Swing based client/server applications.

We avoided covering a few relatively obscure `EntityManager` features like lazily obtaining Entity references using the `getReference` method or using the `clear` method to force detachment of all entities in a persistence context. We encourage you to research these remaining features on your own. However, a critical feature that we did not discuss in the Chapter is robust Entity querying using the powerful query API and JPQL. We will discuss this in detail in the next Chapter.