Published on **ONJava.com** (http://www.onjava.com/)
http://www.onjava.com/pub/a/onjava/2005/06/29/spring-ejb3.html

POJO Application Frameworks: Spring Vs. EJB 3.0

by Michael Juntao Yuan
06/29/2005

Albert Einstein once said, "Everything should be made as simple as possible, but not simpler." Indeed, the pursuit of scientific truth has been all about simplifying a theory's underlying assumptions so that we can deal with the issues that really matter. The same can be said for enterprise software development.

A key to simplifying enterprise software development is to provide an application framework that hides complexities (such as transaction, security, and persistence) away from the developers. A well-designed framework promotes code reuse, enhances developer productivity, and results in better software quality. However, the current EJB 2.1 framework in J2EE 1.4 is widely considered poorly designed and over-complicated. Unsatisfied with the EJB 2.1 framework, Java developers have experimented with a variety of other approaches for middleware services delivery. Most noticeably, the following two frameworks have generated a lot of developer interest and positive feedback. They are posed to become the frameworks of choice for future enterprise Java applications.

- The Spring framework is a popular but non-standard open source framework. It is primarily developed by and controlled by Interface21 Inc. The architecture of the Spring framework is based upon the Dependency Injection (DI) design pattern. Spring can work standalone or with existing application servers and makes heavy use of XML configuration files.
- The EJB 3.0 framework is a standard framework defined by the Java Community Process (JCP) and supported by all major J2EE vendors. Open source and commercial implementations of pre-release EJB 3.0 specifications are already available from JBoss and Oracle. EJB 3.0 makes heavy use of Java annotations.

These two frameworks share a common core design philosophy: they both aim to deliver middleware services to loosely coupled plain old Java objects (POJOs). The framework "wires" application services to the POJOs by intercepting the execution context or injecting service objects to the POJO at runtime. The POJOs themselves are not concerned about the "wiring" and have little dependency upon the framework. As a result, developers can focus on the business logic, and unit test their POJOs without the framework. In addition, since POJOs do not need to inherit from framework classes or implement framework interfaces, the developers have a high degree of flexibility to build inheritance structures and construct applications.

However, while sharing an overall philosophy, the two frameworks use very different approaches to deliver POJO services. While numerous books and articles have been published to compare either Spring or EJB 3.0 to the old EJB 2.1, no serious study has been done to compare Spring to EJB 3.0. In this article, I will examine some key differences behind the Spring and EJB 3.0 frameworks, and discuss their pros and cons. The topics covered in this article also apply to other lesser-known enterprise

middleware frameworks, as they all converge on the "loosely coupled POJO" design. I hope this article will help you choose the best framework for your needs.

Vendor Independence

One of the most compelling reasons for developers to choose the Java platform is its vendor independence. EJB 3.0 is an open standard designed for vendor independence. The EJB 3.0 specification is developed and supported by all major open source and commercial vendors in the enterprise Java community. The EJB 3.0 framework insulates developers from application server implementations. For instance, while JBoss's EJB 3.0 implementation is based on Hibernate, and Oracle's is based on TopLink, developers need to learn neither Hibernate- nor TopLink-specific APIs to get their applications running on JBoss and Oracle. Vendor independence differentiates the EJB 3.0 framework from any other POJO middleware frameworks available today.

However, as many EJB 3.0 critics are quick to point out, the EJB 3.0 specification has not yet reached the final release at the time of this writing. It will likely be another one to two years before EJB 3.0 is widely adopted by all major J2EE vendors. But even if your application server does not yet support EJB 3.0 natively, you can still run EJB 3.0 applications in the server by downloading and installing an "embeddable" EJB 3.0 product. For instance, the JBoss embeddable EJB 3.0 product is open source and runs in any J2SE-5.0-compatible environment (i.e., in any Java application server). It is currently under beta testing. Other vendors may also soon release their own embeddable EJB 3.0 products, especially for the "data persistence" part of the specification.

On the other hand, Spring has always been a non-standard technology and will remain that way in the foreseeable future. Although you can use the Spring framework with any application server, Spring applications are locked into both Spring itself and the specific services you choose to integrate in Spring.

- While the Spring framework is an open source project, Spring has a proprietary XML format for configuration files and a proprietary programming interface. Of course, this type of lock-in happens to any non-standard product; it is not specific to Spring. But still, the long-term viability of your Spring application depends on the health of the Spring project itself (or Interface21 Inc., which hires most of Spring's core developers). In addition, if you use any of the Spring-specific services, such as the Spring transaction manager or Spring MVC, you are locked into those APIs as well.
- Spring applications are not agnostic to back-end service providers, either. For instance, for data persistence services, the Spring framework comes with different DAO and template helper classes for JDBC, Hibernate, iBatis, and JDO. So if you need to switch the persistence service provider (e.g., change from JDBC to Hibernate) for a Spring application, you will need to refactor your application code to use the new helper classes.

Service Integration

From a very high level, the Spring framework sits above application servers and service libraries. The service integration code (e.g., data access templates and helper classes) resides in the framework and is exposed to the application developers. In contrast, the EJB 3.0 framework is tightly integrated into the application server and the service integration code is encapsulated behind a standard interface.

As a result, EJB 3.0 vendors can aggressively optimize the overall performance and developer experience. For instance, in JBoss's EJB 3.0 implementation, when you persist an Entity Bean POJO using the `EntityManager`, the underlying Hibernate session transaction is automatically tied to the calling method's JTA transaction, and it commits when the JTA transaction commits. Using a simple `@PersistenceContext` annotation (see later in this article for an example), you can even tie the `EntityManager` and its underlying Hibernate transaction to an application transaction in a stateful session bean. The application transaction spans across multiple threads in a session and it is very useful in transactional web applications, such as multi-page shopping carts. The above simple and integrated

programming interface is made possible due to the tight integration between the EJB 3.0 framework, Hibernate, and Tomcat inside of JBoss. A similar level of integration is also archived between Oracle's EJB 3.0 framework and its underlying Toplink persistence service.

Another good example of integrated services in EJB 3.0 is clustering support. If you deploy an EJB 3.0 application in a server cluster, all of the fail-over, load-balancing, distributed cache, and state replication services are automatically available to the application. The underlying clustering services are hidden behind the EJB 3.0 programming interface and they are completely transparent to EJB 3.0 developers.

In Spring, it is more difficult to optimize the interaction between the framework and the services. For instance, in order to use Spring's declarative transaction service to manage Hibernate transactions, you have to explicitly configure the Spring `TransactionManager` and Hibernate `SessionFactory` objects in the XML configuration file. Spring application developers must explicitly manage transactions that span across several HTTP requests. In addition, there is no simple way to leverage clustering services in a Spring application.

Flexibility in Service Assembly

Since the service integration code in Spring is exposed as part of the programming interface, application developers have the flexibility to assemble services as needed. This feature enables you to assemble your own "lightweight" application servers. A common usage of Spring is to glue Tomcat together with Hibernate to support simple database-driven web applications. In this case, Spring itself provides transaction services and Hibernate provides persistence services--this setup creates a mini application server in itself.

EJB 3.0 application servers typically do not give you that kind of flexibility in picking and choosing on the services you need. Most of the time, you get a set of prepackaged features, some of which you might not need. However, if the application server features a modular internal design, as JBoss does, you might be able to take it apart and strip out the unnecessary features. In any case, it is not a trivial exercise to customize a full-blown application server.

Of course, if the application scales beyond a single node, you would need to wire in services (such as resource pooling, message queuing, and clustering) from regular application servers. The Spring solution would be just as "heavyweight" as any EJB 3.0 solution, in terms of the overall resource consumption.

In Spring, the flexible service assembly also makes it easy to wire mock objects, instead of real service objects, into the application for out-of-the-container unit testing. In EJB 3.0 applications, most components are simple POJOs, and they can be easily tested outside of the container. But for tests that involve container service objects (e.g., the persistence `EntityManager`), in-container tests are recommended, as they are easier, more robust, and more accurate than the mock objects approach.

XML Versus Annotation

From the application developer's point view, Spring's programming interface is primarily based upon XML configuration files while EJB 3.0 makes extensive use Java annotations. XML files can express complex relationships, but they are also very verbose and less robust. Annotations are simple and concise, but it is hard to express complex or hierarchical structures in annotations.

Spring and EJB 3.0's choices of XML or annotation depend upon the architecture behind the two frameworks: since annotations can only hold a fairly small amount of configuration information, only a pre-integrated framework (i.e., most of plumbing has been done in the framework) can make extensive use of annotations as its configuration option. As we discussed, EJB 3.0 meets this requirement, while Spring, being a generic DI framework, does not.

Of course, as both EJB 3.0 and Spring evolve to learn from each other's best features, they both support XML and annotations to some degree. For instance, XML configuration files are available in EJB 3.0 as

an optional overriding mechanism to change the default behavior of annotations. Annotations are also available to configure some Spring services.

The best way to learn the differences between the XML and annotation approaches is through examples. In the next several sections, let's examine how Spring and EJB 3.0 provide key services to applications.

Declarative Services

Spring and EJB 3.0 wire runtime services (such as transaction, security, logging, messaging, and profiling services) to applications. Since those services are not directly related to the application's business logic, they are not managed by the application itself. Instead, the services are transparently applied by the service container (i.e., Spring or EJB 3.0) to the application at runtime. The developer (or administrator) configures the container and tells it exactly how/when to apply services.

EJB 3.0 configures declarative services using Java annotations, while Spring uses XML configuration files. In most cases, the EJB 3.0 annotation approach is the simpler and more elegant way for this type of services. Here is an example of applying transaction services to a POJO method in EJB 3.0.

```
public class Foo {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public bar () {
      // do something ...
    }
}
```

You can also declare multiple attributes for a code segment and apply multiple services. This is an example of applying both transaction and security services to a POJO in EJB 3.0.

```
@SecurityDomain("other")
public class Foo {

    @RolesAllowed({"managers"})
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public bar () {
      // do something ...
    }
}
```

Using XML to specify code attributes and configure declarative services could lead to verbose and unstable configuration files. Below is an example of XML elements used to apply a very simple Hibernate transaction to the `Foo.bar()` method in a Spring application.

```
<!-- Setup the transaction interceptor -->
<bean id="foo"
  class="org.springframework.transaction
        .interceptor.TransactionProxyFactoryBean">

    <property name="target">
        <bean class="Foo"/>
    </property>

    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>

    <property name="transactionAttributeSource">
        <ref bean="attributeSource"/>
    </property>
</bean>
```

```
<!-- Setup the transaction manager for Hibernate -->
<bean id="transactionManager"
  class="org.springframework.orm
         .hibernate.HibernateTransactionManager">

    <property name="sessionFactory">
        <!-- you need to setup the sessionFactory bean in
             yet another XML element -- omitted here -->
        <ref bean="sessionFactory"/>
    </property>
</bean>

<!-- Specify which methods to apply transaction -->
<bean id="transactionAttributeSource"
  class="org.springframework.transaction
         .interceptor.NameMatchTransactionAttributeSource">

    <property name="properties">
        <props>
            <prop key="bar">
        </props>
    </property>
</bean>
```

The XML complexity would grow geometrically if you added more interceptors (e.g., security interceptors) to the same POJO. Realizing the limitations of XML-only configuration files, Spring supports using Apache Commons metadata to specify transaction attributes in the Java source code. In the latest Spring 1.2, JDK-1.5-style annotations are also supported. To use the transaction metadata, you need to change the above `transactionAttributeSource` bean to an `AttributesTransactionAttributeSource` instance and add additional wirings for the metadata interceptors.

```
<bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy
           .DefaultAdvisorAutoProxyCreator"/>
<bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor
           .AttributesTransactionAttributeSource"
    autowire="constructor"/>
<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor
           .TransactionInterceptor"
    autowire="byType"/>
<bean id="transactionAdvisor"
    class="org.springframework.transaction.interceptor
           .TransactionAttributeSourceAdvisor"
    autowire="constructor"/>
<bean id="attributes"
    class="org.springframework.metadata.commons
           .CommonsAttributes"/>
```

The Spring metadata simplifies the `transactionAttributeSource` element when you have many transactional methods. But it does not solve the fundamental problems with XML configuration files--the verbose and fragile transaction interceptor, `transactionManager`, and `transactionAttributeSource` are all still needed.

Dependency Injection

A key benefit of middleware containers is that they enable developers to build loosely coupled applications. The service client only needs to know the service interface. The container instantiates service objects from concrete implementations and make them available to clients. This allows the container to switch between alternative service implementations without changing the interface or the client-side code.

The Dependency Injection pattern is one of best ways to implement loosely coupled applications. It is much easier to use and more elegant than older approaches, such as dependency lookup via JNDI or container callbacks. Using DI, the framework acts as an object factory to build service objects and injects those service objects to application POJOs based on runtime configuration. From the application developer's point of view, the client POJO automatically obtains the correct service object when you need to use it.

Both Spring and EJB 3.0 provide extensive support for the DI pattern. But they also have some profound differences. Spring supports a general-purpose, but complex, DI API based upon XML configuration files; EJB 3.0 supports injecting most common service objects (e.g., EJBs and context objects) and any JNDI objects via simple annotations.

The EJB 3.0 DI annotations are extremely concise and easy to use. The `@Resource` tag injects most common service objects and JNDI objects. The following example shows how to inject the server's default `DataSource` object from the JNDI into a field variable in a POJO. `DefaultDS` is the JNDI name for the `DataSource`. The `myDb` variable is automatically assigned the correct value before its first use.

```
public class FooDao {

    @Resource (name="DefaultDS")
    DataSource myDb;

    // Use myDb to get JDBC connection to the database
}
```

In addition to direct field variable injection, the `@Resource` annotation in EJB 3.0 can also be used to inject objects via a setter method. For instance, the following example injects a session context object. The application never explicitly calls the setter method--it is invoked by the container before any other methods are called.

```
@Resource
public void setSessionContext (SessionContext ctx) {
    sessionCtx = ctx;
}
```

For more complex service objects, special injection annotations are defined. For instance, the `@EJB` annotation is used to inject EJB stubs and the `@PersistenceContext` annotation is used to inject `EntityManager` objects, which handle database access for EJB 3.0 entity beans. The following example shows how to inject an `EntityManager` object into a stateful session bean. The `@PersistenceContext` annotation's `type` attribute specifies that the injected `EntityManager` has an extended transaction context--it does not automatically commit with the JTA transaction manager, and hence it can be used in an application transaction that spans across multiple threads in a session.

```
@Stateful
public class FooBean implements Foo, Serializable {

    @PersistenceContext(
      type=PersistenceContextType.EXTENDED
    )
    protected EntityManager em;

    public Foo getFoo (Integer id) {
        return (Foo) em.find(Foo.class, id);
    }
}
```

The EJB 3.0 specification defines server resources that can be injected via annotations. But it does not support user-defined application POJOs to be injected into each other.

In Spring, you first need to define a setter method (or constructor with arguments) for the service object in your POJO. The following example shows that the POJO needs a reference to the Hibernate session factory.

```
public class FooDao {

    HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate (HibernateTemplate ht) {
        hibernateTemplate = ht;
    }

    // Use hibernateTemplate to access data via Hibernate
    public Foo getFoo (Integer id) {
        return (Foo) hibernateTemplate.load (Foo.class, id);
    }
}
```

Then, you can specify how the container gets the service object and wire it to the POJO at runtime through a chain of XML elements. The following example shows the XML element that wires a data source to a Hibernate session factory, the session factory to a Hibernate template object, and finally, the template object to the application POJO. Part of the reason for the complexity of the Spring code is the fact that we need to inject the underlying Hibernate plumbing objects manually, where the EJB 3.0 `EntityManager` is automatically managed and configured by the server. But that just brings us back to the argument that Spring is not as tightly integrated with services as EJB 3.0 is.

```
<bean id="dataSource"
  class="org.springframework
         .jndi.JndiObjectFactoryBean">
    <property name="jndiname">
        <value>java:comp/env/jdbc/MyDataSource</value>
    </property>
</bean>

<bean id="sessionFactory"
  class="org.springframework.orm
         .hibernate.LocalSessionFactoryBean">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>

<bean id="hibernateTemplate"
  class="org.springframework.orm
         .hibernate.HibernateTemplate">
    <property name="sessionFactory">
        <ref bean="sessionFactory"/>
    </property>
</bean>

<bean id="fooDao" class="FooDao">
    <property name="hibernateTemplate">
        <ref bean="hibernateTemplate"/>
    </property>
</bean>

<!-- The hibernateTemplate can be injected
        into more DAO objects -->
```

Although the XML-based Dependency Injection syntax in Spring is complex, it is very powerful. You can inject any POJO, including the ones defined in your applications, to another POJO. If you really want to

use Spring's DI capabilities in EJB 3.0 applications, you can inject a Spring bean factory into an EJB via the JNDI. In some EJB 3.0 application servers, the vendor might define extra non-standard APIs to inject arbitrary POJOs. A good example is the JBoss MicroContainer, which is even more generic than Spring, as it handles Aspect-Oriented Programming (AOP) dependencies.

Conclusions

Although Spring and EJB 3.0 both aim to provide enterprise services to loosely coupled POJOs, they use very different approaches to archive this goal. Dependency Injection is a heavily used pattern in both frameworks.

With EJB 3.0, the standards-based approach, wide use of annotations, and tight integration with the application server all result in greater vendor independence and developer productivity. With Spring, the consistent use of dependency injection and the centralized XML configuration file allow developers to construct more flexible applications and work with several application service providers at a time.

Acknowledgments

The author would like to thank Stephen Chambers, Bill Burke, and Andy Oliver for valuable comments.

Resources

- The Spring framework (see also CodeZoo: Spring)
- EJB 3.0
- JBoss EJB 3.0
- Oracle Application Server EJB 3.0 Preview

Michael Juntao Yuan specializes in end-to-end enterprise solutions and is a mobile geek and avid open source supporter.
