Spring Framework for rapid open source J2EE Web Application Development: A case study

John Arthur
Science Applications International Corporation
JOHN.A.ARTHUR@saic.com

Shiva Azadegan
Towson University
azadegan@towson.edu

Abstract

In the highly competitive arena of web application development, it is important to develop the application as accurately, economically, and efficiently as possible. One way to increase productivity is to decrease complexity. This has been an underlying theme in a recent movement to change the way programmers approach developing Java 2 Platform, Enterprise Edition (J2EE) web applications. The focus of the change is how to create J2EE-compliant software without using Enterprise Java Beans (EJB). The foremost alternative is the **Spring Framework**, which provides less services but it is much less intrusive than EJB. The driving force behind this shift is the need for greater productivity and reduced complexity in the area of web application software development and implementation. In this paper we briefly describe Spring underlying architecture and present a case study using Spring.

Introduction

Due to the fundamental nature of the software's architecture, the choice of architecture usually has the greatest impact on productivity compared to any other individual aspect of software development (1). One of the best ways to increase productivity is to find the simplest architecture which fulfills the needs of the client, while still providing the desired requirements of J2EE architecture. The core building blocks for any J2EE application are the presentation layer (User Interface, UI, tier), the business services layer (Middle tier), and the data access layer (Enterprise Information Systems, EIS, tier). These three layers need to be implemented at the bare minimum to be considered J2EE compliant. A number of systems also include a persistence layer as part of the business services layer.

EJB Architecture

The Local EJB architecture is shown in Figure 1. As previously mentioned, the three layers are presented: the UI tier, the Middle tier, and the EIS tier. The UI tier is normally supplied by a Model-View-Controller (MVC) framework. The business layer is provided by "business methods" interfaces that the web tier objects can directly access without having to use the EJB APIs. The business objects are stateless session beans with local interfaces, running inside an EJB container, and will provide transaction management, thread management, and role-base security. All data access will be through entity beans in the "classic" J2EE architecture. Container Managed Persistence (CMP) entity beans are regarded as the best choice. Java Transaction API (JTA) accessed through the EJB container provides transaction coordination to the EIS tier.

Lightweight container architecture

As illustrated in Figure 2, the lightweight container also uses an MVC framework to implement the web tier. The Spring framework can use a dedicated web framework such as Struts or WebWork , or can be managed by the lightweight container to integrate closely with the business objects. All the business objects are Plain Old Java Objects (POJO's) running inside the lightweight container. The lightweight container provides enterprise services by invoking an Aspect-Oriented Programming (AOP) Interceptor. This way it can transparently weave additional behavior before or after business method

Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05)
0-7695-2294-7/05 $20.00 © 2005 IEEE

execution [1]. Since lightweight containers rely on AOP interceptors, there is no need to depend upon the container API's, and therefore, they are usable outside any container.

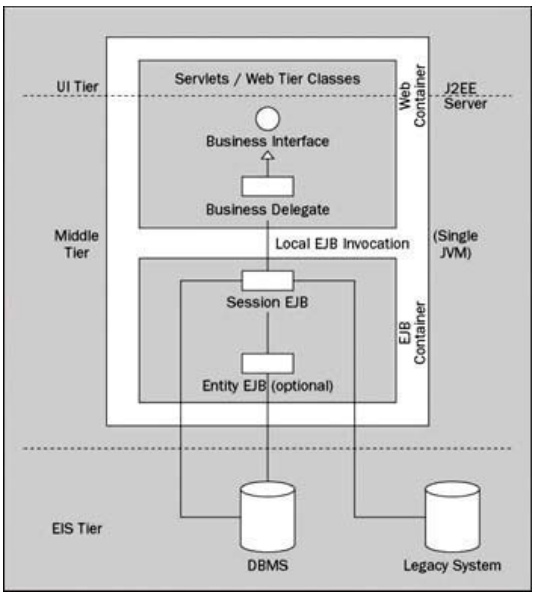

Figure 1 - Local EJB architecture presenting all three core building blocks (1)

For the best results, the business objects should be accessed exclusively though their interfaces. Data Access can use either JDBC or a lightweight Object/Relational (O/R) mapping layer to provide transparent persistence. The EIS tier is the same as the Local EJB EIS tier.

Spring Framewrk.

The goal is to produce web application software accurately, economically, and efficiently, it is important to emphasize a number of objectives: simplicity, testability, and portability. The simpler it is to develop and maintain code, the more cost effective the project will be over the long run. EJBs are very complex and require an EJB container, which translates into the need for a high-end application server. The necessity of an application server also adds to the administrative complexity of using EJBs [2]. The lightweight container only requires a servlet engine. This also addresses the issue of portability. EJBs are written specifically for EJB containers and rely heavily on the EJB APIs. Due to the non-invasive nature of the lightweight container, business objects designed for lightweight containers can be run outside of the container. The ease of testing also factors heavily into the picture.

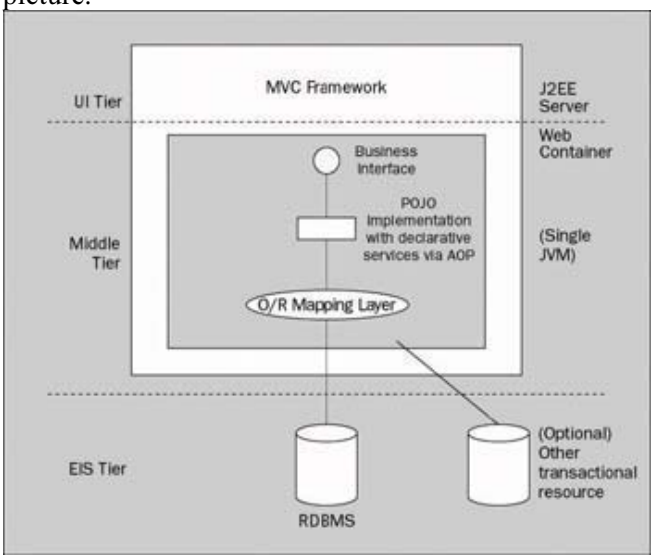

Figure 2 - EJB lightweight container illustrating the MVC framework implementing the web tier (1).

It is difficult to test the business logic of EJBs because the implementing classes are dependent on the EJB container. This is not the case with the lightweight framework, because the business objects are POJO's. The POJO's make it easy to unit test outside the application server and can even incorporate JUnit to assist with testing. The final determining factor was a personal one; the first author has tried to learn to implement EJB architecture in the past and had always found it very confusing and obtuse. After working with the Spring framework for less than a week, he was easily able to implement some rather complex functionality that he was never able to achieve using EJBs. The Spring framework is much more intuitive.

How does the Spring framework provide the necessary infrastructure without the complexity of EJB? The key is type of Inversion of Control (IoC) the Spring framework implements. Inversion of Control concept is easy thought of as the Hollywood Principle: "Don't call us, We'll call you" [3].

There are two main types of IoC: dependency lookup and dependency injection as shown in the Figure 3 below.

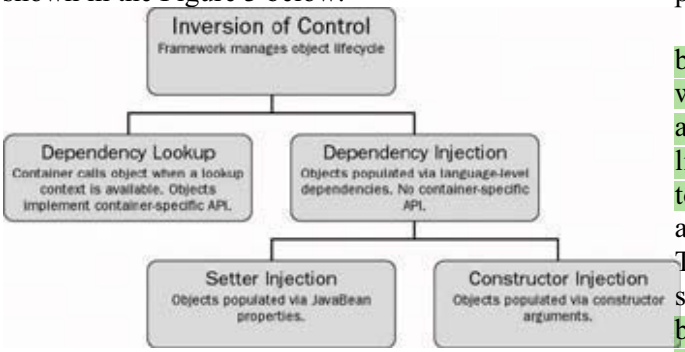

Figure 3 - Inversion of Control Hierarchy (1)

Dependency Lookup has the container provide callbacks to components, and a lookup context. Therefore each component uses container APIs to look up resources. EJB uses the dependency lookup form of IoC, managed objects are responsible for doing their own lookups. "The fundamental principle of Dependency Injection for configuration is that application objects should not be responsible for looking up the resource or collaborators they depend on. Instead, an IoC container should configure the objects, externalizing resource lookup from application code into the container. (1)" The container is wholly responsible for dependency lookup and does not depend on special container API's or interfaces. The lookup is completely removed the application code, the objects are easily usable outside the container, and not special interfaces are required.

There are two flavors of Dependency Injection: Setter Injection and Constructor Injection:

Setter Injection expresses dependencies via JavaBean properties. The setter methods are invoked immediately after the object is instantiated by the container, before it handles any business methods, therefore, there are no threading issues relating to these properties. There is no longer any need for exception handling, as the container can handle configuration-time errors. *Constructor Injection* components express dependencies through constructor arguments. Spring Framework offers support for both Setter Dependency Injection and Constructor Dependency Injection. There are numerous examples of Setter Dependency Injection in the demonstration program.

Spring can serve as the backbone for the business object layer, or middle tier, of a J2EE web application. Spring provides a web application context concept, a powerful lightweight IoC container that seamlessly adapts to a web environment. It can be accessed from any kind of web tier, whether Struts, WebWork, Tapestry, JSF, Spring web MVC, or a custom solution [3]. Spring framework was used as the business services layer for the demonstration program. After experimenting with both Struts and JSF, Spring's web MVC was implemented for the presentation layer. Spring's MVC was found to be more intuitive than either Struts or JSF at this time.

The EIS layer is represented by a MySQL database. The last step on the architecture was to decide how to implement data access objects and data persistence. After thorough research, it was discovered that the Hibernate O/R mapping tool is relatively stable, strongly supported by the Spring Framework, and has Xdoclet support for attribute-oriented programming (explained later in the text). Other benefits of using Hibernate are that it is very intuitive with a shallow learning curve as well as having its own Hibernate Query Language [5].

Development Environment

We researched a number of tools in order to aid development. Eclipse IDE was used due to the fact that it is open-source and has a very large support group. Eclipse also has integrated support for the three other development tools used in the project. The automated build tool ANT, is open-source and widely supported. It is integrated into the Eclipse IDE and allows for a consistent build process. Since this development environment was targeted a team of developers instead of just solo development, a SourceForge account was registered to host the application development. A number of version control tools were researched, including SourceSafe, Subversion, and CVS. Subversion was found to be a very good version control tool, but it is not widely supported at this time and required a number of additional plug-ins to work with Eclipse. CVS

was used as the control versioning tool. While there was no need to incorporate a continuous integration builder for this solo project, CruiseControl was set up to do continuous integration builds for all the projects at the production site. CruiseControl allows the developer to identify build issues almost immediately, and since it is automated, at no cost to the developers. Maven was used to automatically generate a project administration site. Finally, Tomcat was used as the servlet container.

To demonstrate a number of different functionalities of the Spring Framework, a basic, but flexible, issue tracking system was created. There is application called AppFuse which "kickstarts" Spring web application development. AppFuse features include Container Managed Authentication (CMA), Remember Me, Self Registration, Password Hint and GZip Compression. AppFuse also has support for Ant, XDoclet, Spring, Hibernate (or iBATIS), JUnit, Cactus, StrutsTestCase, Canoo's WebTest, Struts Menu, Display Tag Library, OSCache, JSTL and Struts (or Spring MVC) [4].

The basis for Spring Inversion of Control features are found in the Spring *BeanFactory* and the *ApplicationContext*. "The *BeanFactory* provides an advanced configuration mechanism capable of managing beans of any nature, using potentially any kind of storage device (3)." The *ApplicationContext* builds on top of the *BeanFactory* and adds other functionality. The *BeanFactory* provides the configuration framework and basic functionality, with the *ApplicationContext* adds enhanced capabilities to it, some of them perhaps more J2EE and enterprise-centric. An *ApplicationContext* is a complete superset of a *BeanFactory*. Below is a simple example of a bean definition found in the action-servlet.xml file in the demonstration program, which is part of the entire ApplicationContext.

```
<bean id="projectStatusFormController"
        class="com.panogenesis.webapp.action.Proj
        ectStatusFormController"
        autowire="byName">
        <property name="commandName">
        <value>projectStatus</value></property>
        <property name="commandClass">
        <value>com.panogenesis.model.ProjectStat
        us</value></property>
        <property  name="validator"> <ref
        bean="beanValidator"/></property>
        <property name="formView"><value>
        projectStatusForm</value></property>
        <property name="successView"> <value>
        projectStatuses.html</value></property>
</bean>
```

The bean is given an id and bound to a class. Each property of the bean which needs to be accessed is defined using the <property/> tag. The *ApplicationContext* also provides the messaging functionality though the *MessageSource*, access to resources, event propagation through the *ApplicationEvent* class and *ApplicationListerner* interface, and loading of multiple contexts.

Since the main difference between using the Spring Framework and EJB's lies in the business layer, this is the area we will focus on. The first step is to create your business object using a Plain Old Java Object (POJO). The issue class is the object that will contain the basic information about an issue to be tracked. This includes a unique identifier, owner id, creator id, description, which project it belongs to, which components are involve, a collection of associated histories, etc. Since we will be using Hibernate as our Object/Relational mapper for our persistence layer, we can kill two birds with one stone. By using attribute-oriented programming, we can define the Hibernate properties of the Issue class directly in the issue class source code by using Xdoclet tags. This includes defining associations and collections with other objects. These Xdoclet tags are used to generate the .hbm.xml file for each class that is persisted.

Next, we create the data access objects (DAO). Spring always programs to an interface, so we must create a DAO for the issue class. Since we are using Hibernate, extend the *BaseDAOHibernate* object. The DAOImpl object contains all the code for retrieving data from the data source. Now that the DAO objects have been created, we wire them into the Spring Framework. This is very simple, all that has to be done is to add the issueDAO to the applicationContext.xml and bind it to the IssueDAOHibernate object in the following way:

```
<bean id="issueDAO"
class="com.panogenesis.dao.hibernate.Issue
DAOHibernate" autowire="byName"/>
```

This creates a SpringBean named issueDAO for the class IssueDAOHibernate. The target object must be mapped to the Hibernate factory session to be accessed. Below is an example of how the Issue object is mapped.

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

        <value>com/panogenesis/model/Issue.hb
        m.xml</value>
         ...
      </list>
    </property>
    <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        @HIBERNATE-DIALECT@</prop>
    </props>
    </property>
  </bean>
```

This mapping allows Hibernate to access the Issue object through the hibernate interface and creates and allows Spring to access the SessionFactory bean.

Next, we create the actual business service objects, or Managers. These managers are used to decouple the business service layer from the persistence layer by programming to the DAO interface. There is no reference to any particular persistence strategy in the manager interface, which allows one to easily change the persistence level to another O/R mapper or tool such as ObjectRelation Bridge (OJB) or native JDBC. This is also where the transactions are declared and the first example of an AOP interceptor in the demonstration code.

```
<bean id="txProxyTemplate" lazy-init="true"
class="org.springframework.transaction.interceptor.T
ransactionProxyFactoryBean">
    <property name="transactionManager"><ref
bean="transactionManager"/></property>
    <property name="transactionAttributes">
      <props>
        <prop key="save*">
PROPAGATION_REQUIRED </prop>
        <prop key="remove*">
PROPAGATION_REQUIRED</prop>
        <prop key="*">
PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>

  <bean id="issueManager" parent=
"txProxyTemplate">
    <property name="target">
      <bean
class="com.panogenesis.service.impl.IssueManagerI
mpl" autowire="byName"/>
    </property>
  </bean>
```

The txProxyTemplate uses an AOP interceptor to handle transactions. The IssueManager is a child of txProxyTemplate whose target is the IssueManagerImpl class.

Another example of AOP in Spring is the way the admin interceptor is handled. It also demonstrates the "access to resources" functionality of the *ApplicationContext*.

```
 <bean id="adminUrlMapping"
class="org.springframework.web.servlet.handler.Sim
pleUrlHandlerMapping">
    <property name="interceptors">
      <list>
        <ref bean="adminInterceptor"/>
      </list>
    </property>
    <property name="mappings">
      <props>
        <prop key="/activeUsers.html">
filenameController</prop>
        <prop key="/projects.html">
projectController</prop>
        <prop key="/editProject.html">
projectFormController</prop>
        <prop key="/issueResolutions.html">
issueResolutionController</prop>
        <prop key="/editComponent.html">
componentFormController</prop>
        <prop key="/editProjectVersion.html">
projectVersionFormController</prop>
      </props>
    </property>
  </bean>
  <bean id="adminInterceptor"
```

```
        class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
      <property name="authorizedRoles">
<value>admin</value></property>
   </bean>
```

In the code above, the ApplicationContext defines the url mapping that should be accessible to the users with the admin role only. It does this by setting all the pages that should be access only to administrators to props under the mappings property. The interceptor has been defined as a property and the adminInterceptor has been added to the list of references for the interceptors. The adminInterceptor Spring bean is then defined where the authorizedRoles value for the interceptor is set to "admin".

This use of AOP for security and transaction management is one way that Spring addresses the needed functionality of EJBs without having to use EJB. Also, every Spring object that we have created so far is not dependent on the container. Therefore it can be easily unit tested using JUnit. Unlike EJB objects which are constrained by the EJB component model which in turn places restrictions on practicing normal Object Oriented design, Spring imposes fewer constraints on objects increases the ability to achieve greater Object orientation [1].

Conclusion

In conclusion, the use of a lightweight container such as the Spring Framework does have definite advantages over using EJBs in most cases. The Spring Framework allows us to be free of the need for an application server. This decreases the complexity application server administration and allows for greater portability since Spring only requires a servlet container. Using Spring also increases the potential for code reuse for the same reason; code written to the EJB API is only going to run under an EJB container. The necessity of the EJB container for the EJB object greatly increases the difficulty of testing the EJB object. Using the Spring Framework, the objects are easily testable outside of the container which saves time and decreases the complexity of testing. Lastly, the Spring Framework in non-invasive due to the Dependency Injection IoC pattern. This minimizes the dependency of application code on the container, which in turn improves productivity, maximizes the opportunity for code reuse, and improves testability [1].

References:

1) Johnson R, Hoeller J, *Expert One-on-One J2EE Development Without EJB*. 2004, Indianapolis: Wiley Publishing Inc, online book, chunkid=129744793.

2) Raible M. *Spring Live*. 2004, Colorado: SourceBeat Publishing, online book.

3) Johnson R, Hoeller J, Arendsen A, *Spring: java/j2ee application framework – reference documentation version 1.* 2004: available on-line

4) Raible M. *Matt Raible's Appfuse Wiki Site*. http://raibledesigns.com/wiki/Wiki.jsp?page=AppFuse

5) *Hibernate: Hibernate Reference Manual version 2.1.6*, 2004