A Guide to Migrating Enterprise Applications to Spring

Prepared by: Colin Sampaleanu, Co-Founder, SpringSource

October 14, 2008

SpringSource 411 Borel Ave, Suite 101, San Mateo, California 94402, USA **T** +1 650 425 3515 **F** +1 650 212 3515 **E** insidesales@springsource.com

Introduction

First introduced in late 2002 by Rod Johnson in “Expert One-on-One J2EE Design and Development” (Wrox Books), Spring began as a simple framework that provided an easy way to wire application components together. This framework has grown into an alternative platform to the full-stack Enterprise Java platform (Java EE). It includes features normally found on expensive or heavyweight full-stack application servers: support for transactions, security, messaging, remoting, Web Services, persistence, and many other features.

Compared to traditional full-stack Java EE[1], the Spring platform's plain Java programming model, flexible XML or annotation based configuration, Aspect Oriented Programming support, and powerful Enterprise Service layer abstractions and functionality have a number of tangible benefits. Targeting Spring allows applications to be written with less code while also reducing complexity, allows greatly increased portability across multiple environments, allows greater code re-usability, and makes it much easier to test application code.

With the current widespread Spring usage in enterprise Java, developers, architects, and production teams implementing on Spring have a large body of experience and resources to draw upon. Developer tools for Spring include plug-ins for all major Java IDEs, with Spring also available via Maven and Ivy repositories. SpringSource Enterprise provides a certified and indemnified version of the platform for companies that require more comprehensive support, as well as additional tooling.

This document provides only a relative high level introduction to Spring Framework. For in depth articles on Spring, please visit:

- Introduction to Spring Framework 2.5: http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25
- Spring in Production white paper: http://www.springsource.com/whitepapers.

Traditional Java EE Development

Traditional full-stack Java EE web applications are deployed as an Enterprise Archive (EAR), which is typically comprised of a combination of Java EE specific and non Java EE specific component types. The code in these applications typically makes use of both Java EE specific and non-Java EE specific APIs. Some of the major pieces are:

- Enterprise JavaBeans. Typically, application developers expose services as EJBs. EJBs are normally packaged into JARs with deployment descriptors and require class-path configuration settings in the EAR’s MANIFEST.MF file. There are three major types of EJBs:

[1] This white paper targets Java EE 1.4, the version most commonly used to implement and deploy existing Java EE applications. Java EE 5.0, which offers a different component programming model while also implementing a legacy container, has not yet seen widespread adoption, and will be covered in a future white paper.

- Session Beans: these usually provide services and business logic (or act as a facade and delegate to plain Java implementations)
- Entity Beans: a data persistence mapping component
- Message-Driven Beans: which consume messages from a message service in asynchronous fashion

- Non-Java EE components: A significant amount of functionality in a typical full stack Java EE application is agnostic to Java EE. This may include plain Java business logic or services that are delegated to by an EJB layer, data access objects, domain objects, and various areas of functionality brought in as third-party libraries. This functionality is typically packaged as JAR files.
- A Model-View-Controller (MVC) based Web Application. Until recently, the Java EE specification only suggested using the Model 2 pattern, not a specific API. Many existing web applications are written using the Struts (version 1) framework. Recent versions of the Java EE specification include the JavaServer Faces API as an optional Web Framework.
- Java Naming and Directory Interface. The JNDI API is used as a lookup mechanism for finding application components such as EJBs, connection pools (DataSources), and JMS queues within a server environment.
- Other Java EE and Java SE APIs: Full stack Java EE applications will normally make fairly extensive use of a number of other APIs provided as part of the standard Java EE and Java SE execution environments. Since Java EE is an umbrella specification, a number of APIs included as part of the overall specification are in fact available outside of a full Java EE app server environment (e.g. JMS, JTA, Servlets, JMS, etc.). Only some APIs, such as those available to EJBs, require a full-stack application server environment.According to Java EE business logic should be written in EJBs. Therefore, even for the simplest pieces of logic, three artifacts (Remote Interface, Home Interface, Implementation) and a configuration in a rather complex XML Deployment Descriptor had to be implemented. This was such a tedious task that as a work-around generators like XDoclet were invented.

Since Java EE applications are hosted within a nested set of containers, and implementation of classpath lookups in those containers has varied from vendor to vendor, so troubleshooting class loader problems has become one of the most frustrating challenges for Java EE developers.

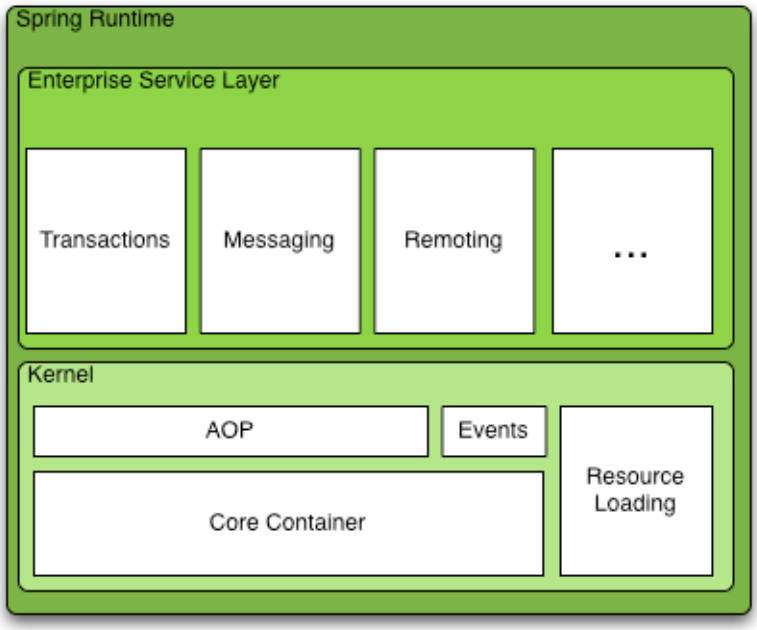
It is relatively common for applications not needing EJB functionality to be packaged as simpler Web Archives (WAR) files, which can execute in both a full-stack Java EE application server and lighter weight Java EE Servlet engine only. The WAR archive may contain both web-related and non web related code in this case.

Traditional full-stack Java EE applications are highly coupled to their environments, and there is a significant amount of code in these applications working directly with low-level, base Java EE APIs.

The Spring Framework

The Spring Framework is an application development and runtime platform that can logically be broken down into two major types of functionality. At the heart of the Spring is a Kernel which is responsible for configuring, managing and enhancing the Spring managed components in the application. In addition, an Enterprise Service Layer provides

functionality and abstractions around enterprise features such as transactions, data-access, messaging, remoting, web UIs and a number of other areas.



Compared to traditional component frameworks such as EJB, one of the key differentiators of the core Spring container is that managed components are typically POJOs (Plain Old Java Objects) that do not need to implement special interfaces or be aware of the container lifecycle. These objects are configured using the Inversion of Control (IoC) principle, meaning that developers do not directly wire objects together with Java code, but rely on the container to create, write, and manage objects. Some of the benefits of IoC, when also combined with Spring's enterprise service abstractions, include:

- Decoupling of components from their execution environment
- Separation of configuration data from application code
- Easy ability to test components in isolation via unit tests, or in combinations via integration tests, without modifying the existing code
- Ability to wrap existing POJOs in a transparent and declarative fashion to “inject” functionality into them, such as transaction support, security, auditing, logging, caching and more. Aside from removing verbose and unsafe boilerplate code, this has additional beneficial effects towards the portability and testability aspects mentioned above.

All Spring features can be deployed within a single, low-cost or free web container such as Tomcat, as EJB container facilities are optional. In a web scenario, typical packaging is a WAR file containing embedded JAR files. The major components of a typical web-based Spring application include:

- Spring-Managed Components: Using Spring's runtime configuration facilities, any object can be managed by Spring, and can be transparently wrapped to add enterprise features such as security, transaction support, logging, and any number of other facilities. The complexity of wiring components together is lifted from the components themselves, and allows developers to focus on writing business applications.

- Spring Enterprise Service APIs and Functionality - The functionality and abstractions provided by Spring around various enterprise services simplify application complexity and decrease development time. These services are accessed via programmatic APIs, or transparently when Spring managed components are wrapped to add this functionality.

- Components and Code Not Managed by Spring: Spring is only used to wire together and manage objects where it adds concrete value. Some classes of objects are typically left as non-managed components, such as data transfer objects, domain objects, third-party code, etc. These objects, while not being Spring managed, may still make use of Spring's enterprise service APIs in a programmatic fashion.

- Web Frameworks: Spring MVC is a full-featured web framework that is easily wired into an existing Spring Application. Spring also supports any other web framework, and includes direct integration classes for Struts, JSF, and other platforms out of the box.

Any standards-compliant web application container (Servlet Engine) can host a Spring web application. SpringSource recommends Apache Tomcat as a proven, popular, reliable, no-cost or low-cost platform for hosting Spring-based applications. Tomcat is licensed using the Apache 2.0 license.

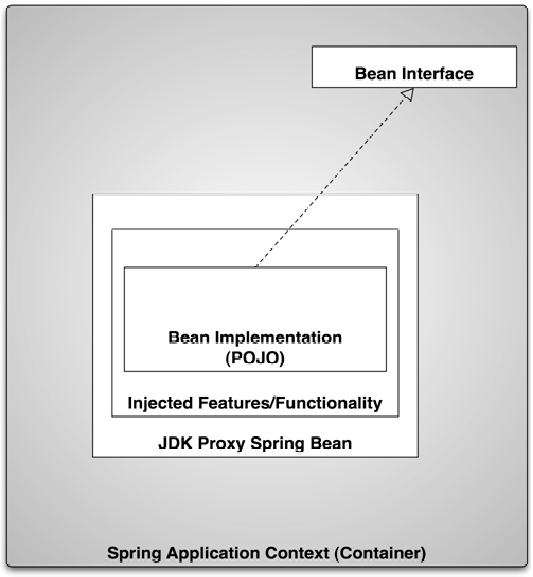
Spring does not require a web application container for non-web use. Unlike Java EE applications, a Spring application can be invoked as a Java command-line program. One common alternate use for Spring is in wiring systems together as a Spring-based daemon application. Some Spring APIs lend themselves to this effort, including Spring Batch, Spring Integration, and even the base Spring JDBC API for moving and updating data.

Kernel Details

Spring Container Overview (including AOP)

The Spring container (and related AOP functionality) is the core engine that drives component configuration and management, and enhances usage of the rest of Spring. Automatically controlling relationships between objects, it is known as a Dependency Injection (DI) or Inversion of Control (IoC) container because the framework handles making sure objects receive any dependencies they need.

Spring-managed objects, also called "beans", are typically configured using one or more XML configuration files, and/or annotations contained within the classes themselves. Spring supports both mechanisms and several others for maximum flexibility. The Spring container is called an Application Context, and it is possible, although certainly not common or necessary, to have multiple Application Contexts in an application.



Spring-managed components are typically "Plain Old Java Objects" or POJOs. These application classes generally do not need to implement special interfaces or extend base classes, or be aware of the container they are running in. This facilitates writing portable and reusable code. However, developers may still hook into component and container lifecycles where needed.

At runtime, objects are created based on the specified configuration metadata, and optionally (if needed) wrapped with a dynamic proxy in a transparent fashion. This provides an extension point for the framework to inject key services such as transaction support, security, auditing, and logging. This transformation process is an example of AOP (Aspect Oriented Programming), and is a key enabler of Spring functionality.

XML and Annotation-driven Configuration

Two of the most common ways of specifying Spring configuration are via the use of XML and Java annotations. In XML configuration, one or more files are created that describe configuration of Spring managed objects in a simple XML syntax. With annotation-driven development, Java annotations inside the source code describe configuration. In either case, the injection generally happens upon loading the context into memory at runtime. The use of a Spring-aware IDE makes working with either form of configuration both easier to work with and to visualize.

For different scenarios, there are pros and cons to the use of XML vs annotations for specifying configuration metadata. As such Spring allows the use of either mechanisms, and in a combined fashion as well. When using annotations, Spring supports both its own native annotations as well as well as standard (but weaker) annotations introduced by the JSR-250 specification.

Container Startup

Spring Application Context startup may be enabled in declarative fashion to happen at web app startup, but core APIs also enable creation of a Spring Application Context programmatically from within various application types (web, command line, etc), and provides a mechanism to look up an object from within a running context. Lifecycle events can be wired in to monitor and participate in all stages of a managed object. As mentioned, most application code does not have to be — and should not be — aware of the Spring container.

Enterprise Server Layer Details

Spring's Enterprise Service Layer provides functionality and abstractions around enterprise features such as transactions, data-access, messaging, remoting, web UIs and a number of other areas. These enterprise services may be accessed

via a programmatic API (provided by Spring), or, in a declarative fashion by specifying that application code should be transparently wrapped to add functionality. Sometimes (with transaction management being one concrete example) either mechanism is available and usable at the discretion of the application developer, depending on the best fit for the application.

It should be noted that it is convenient and customary to configure code (including components provided directly by Spring) using Spring's enterprise service layers in the Spring container. However, the programmatic APIs are generally also usable without the container being involved.

Web Frameworks

Spring supports any web framework simply by creating a Spring context within the container and using it directly. Here are some popular options:

- Spring MVC – The Spring MVC Web Framework is a request/response oriented web application framework. It provides easy integration with other layers of the application, also managed by the Spring IoC container. Spring MVC includes rendering support for JSPs, Velocity, XSLT, Freemarker, Jasper Reports, Excel, and PDF, and supports Tiles for page composition.

- Spring Faces – Spring supports direct integration with JSF via the Spring Faces project, allowing injection of Spring managed components into JSF Managed Beans, and integrating with Spring Web Flow for conversational state and navigation management.

- Spring Web Flow – In addition to the stateless Spring MVC architecture, Spring also provides a powerful Web Flow API for business process based navigation and conversational state management in web user interfaces. Web Flow works on top of both Spring MVC and JSF/Spring Faces, with integration also available for some other Web UI frameworks.

- Other Supported Web Frameworks - Spring also supports integration with the Struts 1.x web framework. Virtually every other web framework come with Spring integration, however those that do not can use Spring for business and persistence logic simply by creating a Spring ApplicationContext.

Transactions

Spring includes comprehensive transaction management support, using either a declarative (annotation or XML-based) approach that is preferred by most users, or a programmatic template approach. In either case, working through Spring eliminates the coupling to either JTA or local transaction management that non-Spring approaches generally imply. When deploying the Spring-based application, it is a simple one or two line configuration change to pick and choose between Spring's JTA transaction manager support (in which case Spring delegates to JTA) or an appropriate local transaction management strategy (in which case Spring itself drives transactions on the local resource, e.g., the DataSource, JPA EntityManager, Hibernate Session, etc.).

Transaction support is integrated with JDBC and ORM support code, so that resources such as connections, JPA EntityManagers, Hibernate Sessions, etc., are managed and shared across the same transaction.

Spring JDBC

The Spring JDBC API makes programming database tasks simpler and more efficient. Using a JDBC Template object, developers do not need to concern themselves with standard JDBC setup/teardown boilerplate in order to perform standard database operations. This reduces the amount of coding and eliminates resource leaks. Spring supports both straight SQL and stored procedure access within the Spring JDBC template. Integration with Spring managed transactions is standard.

Spring ORM (JPA/Hibernate/iBatis/etc.)

Spring supports a series of Object-Relational Mapping (ORM) suites, including Hibernate, iBATIS, Toplink and JDO. Spring also supports deployment of the Java Persistence Architecture (JPA) platform, which provides a standards-based, annotation-driven interface to all but iBATIS. In all cases, Spring provides Factory Beans that create instances of the proper ORM objects within the Spring Context. Spring's JPA integration includes a set of helper classes and JUnit test classes to allow testing outside of the web container.

Spring JMS Support

Enterprise Java applications that need message queuing functionality typically rely on a message queue integrated and available as part of a full stack Java EE application server, or otherwise on a standalone message queue product. In either case, the preferred API for accessing the queue is JMS.

For asynchronous reception of messages, Spring provides a Message Driven POJO approach that is similar to EJB's Message Driven Bean, where a pluggable Spring messaging container handles threading, and drives messages to either a standard MessageListener or even a plain Java receiver that is agnostic to JMS. For synchronous reception of messages, or sending of messages, Spring offers a programmatic template approach that reduces typical boilerplate.

Remoting

Components do not need to be marked in any special way to be made available on the network. The Spring remoting APIs each provide a Service Exporter which wraps the original object to enable the given protocol. Spring also provides a client proxy to the remoted object from a client. This framework allows developers to expose several protocols on the same instance of an object (RMI, HTTPInvoker, JAX-RPC and SOAP support, for example).

Currently, clustering of remoted applications via similar semantics to a multi-tier EJB-based application of stateful services is not supported directly in the Spring Framework. Until a fully supported clustering solution is available, consider either using Spring embedded within the Java EE server application for stateful services, or convert to using stateless services. It is generally accepted that the use of stateful services (as implemented by EJB Stateful Session Beans) does not scale to the same level as does the use of stateless services, and therefore such a conversion should also result in increased application scalability.

Threading and Scheduling

Spring supports a pluggable threading/task execution abstraction, with support for CommonJ, Java 5 (java.util.concurrent), Java 1.4 (backport), JDK Timer, and other variants. In addition, Spring supports tight integration with Quartz for scheduling jobs.

Security

Spring Security (packaged as a separate Portfolio project) is a comprehensive and flexible security framework for Java. It allows role and ACL based security at both the web and method level, comes with advanced functionality out of the box, and is easily extensible to handle custom needs.

Web Services

Spring Web Services (packaged as a separate Portfolio project) enables the creation of document-driven Web services. Facilitating contract-first SOAP service development, it allows the creation of flexible and decoupled (SOA style) web services using one of the many ways to manipulate XML payloads.

Batch Processing

Spring Batch (packaged as a separate Portfolio project) brings the Spring programming model to batch processing in Java, providing structure around the batch processing concepts, and simplifying the creation of batch processing functionality in Java.

integration

Spring Integration (packaged as a separate Portfolio project) is a framework which implements a number of common Enterprise Integration Patterns following the Spring programming model, allowing a lightweight, "plain-Java", and extensible approach to integration.

Testability

Traditional full stack Java EE applications are generally constrained to running within a full Java EE app server environment. Some application components such as EJBs are required to implement base Java EE interfaces or extend base classes, and are thus are both aware of and coupled to the container and container lifecycle. Many traditional Java EE applications also directly use lower level or non-universal APIs such as JTA. These factors generally make testing application code in isolated fashion outside of the executing app server via simple unit tests or IDE driven integration tests very difficult or impossible.

The Spring platform's focus on plain Java components, Inversion of Control configuration model, and Spring's enterprise service abstractions facilitate testing application code outside the container in isolation via unit tests, or in combinations via integration tests, without modifying the existing code.

Spring includes Junit integration for writing and running integration tests out of the IDE or from the command-line build. This is may then be coupled to a continuous integration server such as Bamboo, CruiseControl, or Hudson, to run tests in an automated fashion.

Deployment

Spring places no restrictions on deployment models on the developer. Spring-based applications may be deployed in a full stack Java EE app server, in a lightweight Servlet engine, as a standalone fat client (e.g. Swing or Eclipse RCP) or as "headless" daemon style application with no UI.

Spring's resource loading abstractions allow application context definitions to be read from a variety of sources, such as the Java classpath, relative to a web app WEB-INF directory, or a filesystem location.

In an app server environment, applications utilizing Spring are typically deployed in EAR file or WAR file format. In a Servlet environment, applications utilizing Spring are typically deployed in a WAR file format.

Existing applications not using Spring, whatever the deployment format, are generally able to be modified in an incremental fashion (without changing deployment format) to start using Spring, in order to provide IoC container support, access to JPA and other database APIs, and access to other parts of the Spring portfolio.

Architectural Decisions

Migrating Existing Applications

Caution should be used in migrating existing, mature applications away from legacy, full stack Java EE to another component framework. As in any project plan, the benefits must outweigh the costs. A clear understanding of scope, complexity, and technical challenges are preconditions to estimating a migration plan. In cases where full stack Java EE is highly adopted in an organization and an application will not continue to be enhanced or refactored in significant fashion, remaining with full stack Java EE may make more sense. However, there are a number of considerations which may indicate it is worth migrating to some usage of Spring:

- Cost of development effort with traditional technologies if there is a need to enhance or change application functionality in a significant fashion or on an ongoing basis.
- Expense of licensing, running and maintaining heavyweight Java EE Application Server technologies
- Quality concerns due to inability to easily implement and execute unit and integration tests within traditional Java EE applications.
- Interest in simplifying deployment architecture from a large, multi-tier platform into a single-tier web application to reduce overall complexity

It must also be emphasized that Spring usage is not an all or nothing proposition. While it may be very beneficial to significantly refactor some parts of the code or architecture of a traditional full stack Java EE architecture, it is quite possible to start using Spring in an incremental fashion. Within the existing application, the many Spring enterprise

service APIs may be accessed programmatically, or Spring container usage may be combined with continued usage of other component models, such as EJB.

Physical Separation of Layers

In the early days of Java EE it was considered best practice to physically separate web tier code from middle tier code. In the original EJB 1.x specification, EJBs only supported Remote interfaces, such that all method invocations to them and between them were remote. This physical separation was endorsed in the original Sun Java BluePrints document outlining architectural best practices.

Over time, practical experience proved that remoting of EJBs or other code does not make sense in most scenarios. While application servers may support clustered remoting, it is extremely expensive in terms of latencies and invocation times to do a remote method invocation instead of an in-VM invocation.

Based on this experience, EJB 2.x introduced Local interfaces, with best-practice changing to a recommendation to attempt to keep all processing for a particular web invocation within the same VM, with remoting used only where it adds tangible value, such as performing some expensive processing task on another dedicated machine. Scalability is then achieved in a so-called horizontal fashion by deploying other nodes with identical configurations, and putting a load balancer on top to direct requests among the physical boxes.

Despite the current best practice, some current full stack Java EE applications still exhibit a physical deployment separation between the web and middle tiers, so it is important to take into consideration any remoting scenarios when considering a move to Spring. Refer to the preceding Remoting section for a description of Spring's remoting capabilities. Third party products such as Terracotta JVM Clustering add additional remoting and clustering capabilities above and beyond those native to Spring.

Deployment, Management and Monitoring Tools

Many Java EE application servers have robust deployment and monitoring capabilities, and provide more than a simple JMX console. Tomcat out of the box provides an admin console that is not as comprehensive as that found in most full stack app servers.

SpringSource Application Management Suite, part of SpringSource Enterprise, includes a comprehensive management console that allows visibility and management at the application, app server (including Tomcat), and operating system level, including clustered scenarios.

Migration Strategies

If an application is deemed as a candidate for migration from a Java EE platform into a Spring-based application, the following guidelines will help you research the proper technologies to apply for your case.

Spring Minimum Requirements

Spring’s minimum requirements vary by version, and are documented in detail on the Spring Framework website. However, for Spring 2.5.x web applications, the current requirements are:

- Java: JDK 1.4.2 and above (Spring Framework 2.0.x supports JDK 1.3)
- Servlet version: 2.3

Replace EJBs with Spring Beans

If the existing application contains EJBs, they will need to be rewritten as plain java objects. The following guidelines may guide your porting strategy:

- Session EJBs can be migrated to POJOs managed by Spring.
- Entity Beans can be converted to JPA Entity classes.
- Message-driven beans can be converted to Message-driven POJOs.

Web Applications

Depending on the state of your current web application, migrating to the Spring platform may simply involve upgrading to a recent web application container (Tomcat 5.5 or above is recommended when using Tomcat). If the web application contains references to EJBs, these may be converted as described above, and the new components can be injected from or looked up in the Spring Application Context instead of being looked up in JNDI.

If the web application needs to be rewritten, due to code quality or stability issues, or for future productivity gains when adding new functionality, consider rewriting it in the Spring MVC framework for maximum integration with the Spring library. Alternatively, select a supported web framework and integrate it into the Spring container to provide access to Spring Beans natively.

Migrating Data Sources

If the source Application Server provided the data source via JNDI, they need to be moved either to JNDI DataSources in Tomcat's server.xml configuration file, or created in Spring's configuration file directly using a Jakarta Commons DBCP or C3P0 connection pool. In practice, there is essentially no value in maintaining the DataSources in JNDI in this environment, and use of Commons DBCP is used in the great majority of cases.

Data Access Objects

If your application uses the Data Access Object pattern, consider converting your data access objects to use supporting Spring functionality to reduce complexity, reduce the amount of code needed and improve overall ease of maintenance. For JDBC based data access objects, the central supporting class is Spring's JdbcTemplate, while supporting functionality is available for each major ORM (e.g., all JPA variants). For each data access framework, Spring will manage resources (Connections, EntityManagers, Sessions, etc.) while integrating with the higher level transaction support code.

Transaction Support

Java EE's support for declarative wrapping of EJBs to provide transaction support is a commonly used feature of full stack Java EE applications. Spring provides declarative transaction support using either an annotation based

(@Transactional) approach or via the use of regular expression based XML definitions, providing a functional superset of Java EE's capabilities. In either case, Spring's transaction management wrapper will support either JTA or local resource transaction management.

When migrating transactional EJBs to Spring managed beans, the Spring managed beans should at that time be wrapped with Spring's declarative transaction support to implement equivalent transactionality.

Security

Java EE provides for simple role based declarative security at both the web (Servlet) level and EJB level. Spring Security is a comprehensive and flexible security framework for Java that is a functional superset of Java EE security, while providing much more functionality. It allows role and ACL based security at both the web and method level, comes with advanced functionality out of the box, and is easily extensible to handle custom needs.

Existing standalone usage of Servlet spec security may be kept as-is in a Tomcat environment. That said, given the very limited functionality of Servlet spec security, there are strong advantages to migrating this usage to Spring Security. Where existing apps use both Servlet Spec security and EJB security, it is recommended that both be converted to Spring security.

When migrating EJBs that are currently secured via EJB declarative security to Spring managed beans, the Spring managed beans may at that time be wrapped with Spring Security's declarative method level security functionality for equivalent behavior. Code may then be extended as needed to take advantage of Spring Security's enhanced functionality (e.g. ACL support).

Messaging

Tomcat does not provide an embedded JMS server, nor does Spring. When migrating to Spring, if you are using an existing external JMS implementation (whether a commercial product such as WebSphere MQ, Tiboco, etc., or an open source product such as ActiveMQ) you will not have to change your JMS server. When relying on a JMS server embedded as part of your full stack application server (e.g. WebLogic JMS), you will need to switch to an external provider.

Existing Message Driven EJBs should be converted to use Spring's Message Driven POJO functionality, which provides a functional superset of the EJB capabilities. Existing code which programmatically uses the JMS APIs to send or receive messages may be left as-is, but it is generally recommended that this code be converted to take advantage of Spring's JmsTemplate abstraction. That enables it to take advantage of Spring's transaction abstraction and reduce complexity and future maintenance costs.

Migration Cases

Below is a set of rough guidelines to help you determine whether migrating to Spring and Tomcat makes sense for your organization. These are only suggestions; it is up to you and your team to determine the best course of action based on sound research. Keep in mind that for most migration scenarios, there will be a certain minimal amount of work to

migrate the application so that it uses some Spring functionality, and runs in the Tomcat environment, while additional refactoring allowing the application to take best advantage of Spring will pay off in reduced complexity and future maintenance benefits.

Factors leading to low complexity in migrating application to Spring + Tomcat:

- Straightforward DAO-based database access (using either JDBC or ORM)
- No reliance on Session EJBs, or reliance on a straightforward use of Session EJBs (e.g. smaller quantity or delegating to plain java business objects)
- No reliance on Entity EJBs
- Security implemented using standard Java EE specifications, or no desire to move away from custom third party security APIs
- Well architected, layered application platform
- No integration with proprietary application server frameworks
- No middle tier stateful clustering need
- No need for distributed transaction support

Factors leading to medium complexity in migrating application to Spring + Tomcat:

- Reliance on Entity Beans.
- Use of existing JMS provider in full stack application server.
- Security implemented using third-party security APIs such as SiteMinder, but desire to integrate with or move to Spring Security
- Strong organizational support of EJB and Java EE; Spring not yet adopted
- Reliance on availability of distributed transaction support

Factors leading to highest complexity in migrating application to Spring + Tomcat:

- Heavy use of EJB-specific technologies
- Reliance on custom application server APIs (re-architect to remove dependence first)
- Poor documentation and/or lack of clear understanding of existing code, database or requirements

Technology Comparisons: Java EE and Spring

Java EE or App Server Feature	Spring Alternative	Other Options
Web Tier		
Proprietary web framework (NetUI, etc)	Implement a web-based MVC in Spring MVC	A Spring-supported MVC or Component backed web framework such as JavaServer Faces or Struts 2 can be configured with Spring injection
UI-centric proprietary workflow	Implement Spring WebFlow. WebFlow can be implemented on all major web frameworks, including Spring MVC, Struts 1.x, Struts 2, and JavaServerFaces, and can be adapted to run in many other web application frameworks.	
EJB Components		
Stateless Session Beans	Spring managed POJOs; use default Singleton scope	
Stateful Session Beans	Spring managed POJOs. You could emulate the behavior using Session Scope (the bean is cached in the HTTP Session) or in Prototype scope (if the bean is shared across client instances like a singleton). Other alternatives exist, especially if implementing a flow-based user interface with Spring WebFlow.	Move state to servlet tier session or back to database
Container Managed Entity Beans	Spring and JPA-backed ORM with Hibernate (EJBQL can be reasonably replaced with JPAQL as a query language)	Simple Data Access Objects with Spring JDBC

Java EE or App Server Feature	Spring Alternative	Other Options
Transactions		
Container Managed Transactions in EJBs	Use declarative Spring approach as described above.	
Use of UserTransaction JTA transaction API	Implement Spring transaction wrapping using @Transaction annotation, AOP-based Transaction definition, or using Spring's TransactionTemplate. Decide in the configuration whether or not to use JTA or local resource (JDBC DataSource, JPA EntityManager, etc) transactions.[1]	
Fully Distributed transactions between multiple data sources	Although Spring supports JTA, Tomcat does not implement an XA capable transaction manager out of the box. If JTA is truly needed, consider plugging in an external XA capable JTA transaction manager such as as Atomikos (http://www.atomikos.com/Main/AtomikosCommunity)[2] or Geronimo JTA (http://geronimo.apache.org).	Implement Spring on a full stack Java EE-compliant JTA-capable application server.
Database/Data Sources		
Use of JNDI Datasources	Define a DataSource within Spring's Dependency Injection framework and inject into data-facing objects	Create a JNDI context within Tomcat and mount the DataSource using the jee: namespace in Spring configuration.
DAO-Pattern Database Objects	Consider simplifying logic and setup/teardown code using Spring JDBC or ORM integration.	Keep majority of code as-is and only inject the JDBC datasource into the existing DAOs

Java EE or App Server Feature	Spring Alternative	Other Options
Messaging, Integration and Web Services		
Built-in JMS container	Implement external JMS container, such as open source ActiveMQ or commercial alternative.	
Proprietary vendor Web Service implementations such as BEA's WLI web services	Implement WS-* based web services using Spring Web Services API.	For RESTful web services, consider implementing Spring MVC controllers that respond to RESTful URLs.[3]
Use of BPM or other vendor specific business workflow engines such as Weblogic Integration	No direct BPM implementation in Spring	If UI-centric BPM, consider Spring WebFlow. Alternately, consider Spring Modules jBPM integration with JBoss, or OpenSymphony Workflow Integration, or integrate with another BPM vendor via a SOA or ESB
Architecture/Other		
Existing objects created or looked up via factory objects.	Re-wire object dependencies by using Spring's native Dependency Injection. By removing factories from the development process and wiring in the Spring configuration, you may build separate unit tests which can isolate a given Spring component and make true unit testing a real option.	
Use of EJB Interceptors	Replace with Spring's AspectJ Aspect Oriented Programming mechanisms.[4]	
Multiple, distributed application servers configured as different tiers (Web Tier and Business Tier, for example)	Simplify architecture and run on flat, single tier platform. Consider load balancing and avoid caching data shared between users. Some applications may still require multiple nodes and this may be supported via Spring Remoting.[5]	Consider distributed VM technologies such as Terracotta Software to transparently provide large, clustered VM instances for scalability

Java EE or App Server Feature	Spring Alternative	Other Options
Security		
Security Configuration implemented in JAAS modules	Implement security using Spring Security API (formerly known as Acegi). Spring Security can be wired to many different data sources and can be configured for custom authorization and authentication methods.	Filter or Interceptor based custom Security, or forward your security requirements in Spring Security to the existing JAAS module
Security assigned to EJB, web components using xml deployment descriptors	Use Spring Security, either with Annotations or within the Spring XML configuration	
Monitoring		
Existing custom integration with a platform monitoring tool	Consider implementing JMX monitoring and integrating with a JMX-capable platform monitoring server. The @ManagedResource annotation, for example, provides an easy way to expose monitoring attributes via JMX.[6]	Spring Application Monitoring Service (AMS) provides a distributed, production-capable monitoring platform for any JMX capable application server, including Spring Source Application Server.

Table Notes:

[1] The Spring transaction abstraction allows the code to be decoupled from JTA. In most cases, with a single data source, you do not need to use JTA to implement transactions. You can add JTA when appropriate without changing any of the application code.

[2] The Atomikos distribution includes sample Spring configuration. Atomikos is available in both an open source, Community edition, and a higher performance commercial version, ExtremeTransactions.

[3] Using Spring Annotations to build Restful URLs and parse parameters is part of the release plan for Spring 3.0. For Spring 2.5, see also http://find23.net/2008/08/18/13/

[4] Although EJB 3.0 interceptors provide similar features for advising code, they must be defined in each EJB separately via the ejb-jar.xml file or via an annotation. AspectJ's pointcut language allows for attaching aspects to code based on a

search pattern such as a series of packages or class pattern names, as well as by attaching to annotations. For more on Spring AOP, visit http://static.springframework.org/spring/docs/2.5.x/reference/aop.html

[5] For information on Spring Remoting, visit http://static.springframework.org/spring/docs/2.5.x/reference/remoting.html

[6] For more on Spring JMX, visit http://static.springframework.org/spring/docs/2.5.x/reference/jmx.html