



Migrating JEE Applications from WLS/WAS to SpringSource tc Server™

A Technical Perspective

WHITE PAPER

Table of Contents

1. Migrating JEE Applications to tc Server™	3
2. Planning and Cost Analysis	3
2.1 Migration Complexity Factors	4
3. JEE Specification and JEE Application Servers	6
4. Java EE 5.0 API Support	7
5. Code Migration Strategies	9
5.1 Layering and Interfaces	9
5.2 Migrating Significant Code-Level Changes Piece-by-Piece	9
5.3 Service Layer Migration Strategy	10
5.4 Data Access Layer and Data Access Objects	10
5.5 Migrating Web and Presentation Layer Code	10
5.6 Migrating Transactional Code	11
5.6.1 JTA: Non-EJB and EJB Bean-Managed Transactions	11
5.6.2 EJBs with Container-Managed Transactions (CMT)	11
5.7 Session EJB Migration Strategy	12
5.7.1 Session Beans with Container-Managed Transactions (CMT)	12
5.7.2 Session Beans with Bean Managed Transactions (BMT)	12
5.7.3 Session Beans with Container-Managed Security	12
5.8 Message-Driven EJB Migration Strategy	12
5.9 Entity EJB Migration Strategy	13
5.10 EJB Client Access Code Migration Strategy	14
6. Remoting Services	14
7. DataSources	14
8. Security	14
9. Messaging	15
10. Administration and Management	15
10.1 Server Administration and Management	15
10.2 Application Administration and Management	16
11. Conclusion	16

1. Migrating JEE Applications to tc Server™

In this whitepaper, we will be discussing technical considerations for migrating IT projects from commercial JEE Application Servers to light weight JAVA containers, specifically the SpringSource tc Server™. Many IT organizations have been re-thinking their commitment to commercial JEE Application Servers, due to both challenging business environments that drive the need for more cost effective application architectures and the need to transition to more responsive/agile applications development architectures. When IT organizations talk about “migrating” their applications, they generally are focusing on one or more of three distinct situations. These are:

- Moving existing applications (or slices of applications) from JEE servers to lightweight, modular, horizontally scalable container infrastructures
- Expanding access to existing JEE applications by adding services layers in lightweight containers.
- Transitioning new application development away from JEE application servers and focusing on light weight containers and modern application frameworks.

There are many excellent reasons to consider moving applications from commercial JEE servers sold by Oracle/BEA, IBM, etc. In this whitepaper, we focus on the developer’s perspective in migrating JEE application code from commercial JEE servers to the SpringSource tc Server. In addition to functionality provided by Tomcat, SpringSource tc Server™ offers sophisticated management and monitoring that can significantly reduce the effort of migration, and in particular, testing/debugging and performance optimization during the development phase.

2. Planning and Cost Analysis

One of the most important roles for the developers in the early stages of a migration project is to participate in the planning and cost/benefit analysis. Only the developers can accurately assess the difficulty of moving any particular application from commercial JEE Application Servers to other infrastructures. This is because migrating a web application that uses only the servlet container is generally very straight forward (the most common problem being that the JEE app servers are more forgiving of coding errors), while moving an application that utilizes entity beans, session beans, or other application server services and features may even prove too complex/costly to migrate at all. In practice, most of today’s applications will fall somewhere in between those extremes.

Successful migration projects also require close coordination between IT and development, both during the cost assessment and during the subsequent project implementation, in order to assure that the cost benefit balance is achieved as planned. Every company or individual will have their own criteria for estimating costs, but as an example, consider this grouping:

Minimal effort

- One day or less of new development work. Runs pretty much as is (packaged as WAR, no use of unsupported APIs)
- Testing effort primarily
 - Note:** DO NOT underestimate this, since most test environments are not particularly portable!

Low effort

- Several days to two weeks of new development work
- Few changes to application code, plus testing effort...see note above

High effort

- Many weeks or months of new development work
- Higher risk

Assess the level of development effort in these areas:

- Java EE APIs used
- JEE application server services, outside the JEE specification, but included in the app server to differentiate the product and/or meet specific market needs (something like CORBA, for example)
- Third-party libraries and services
- Packaging
- Types and number of components
- Code quality assessment, including factors such as coupling, layering, and so on.
 - Note:** both major JEE application servers allow (different) liberties with the servlet and JSP specification that Tomcat does not
- Maintenance and manageability history

The new SpringSource *Migration Impact Quick Scan Tool* will allow scanning candidate applications to gather information about various APIs, types of components (such as EJBs), and number of components. This planning tool will provide an excellent starting point and is both faster and more accurate than line by line code review. Contact your SpringSource sales representative for further information regarding this valuable resource.

2.1 Migration Complexity Factors

Several factors contribute to application migration complexity, when migrating applications from a full-stack Java EE environment to a tc Server™ environment. The following tables serve as a guide for creating such a cost analysis, and will typically be divided among the various development teams for assessment:

DOCUMENTATION	MIGRATION COMPLEXITY
Good documentation with clear understanding of existing code, database, and requirements.	Low complexity
Poor documentation and/or lack of clear understanding of existing code, database, or requirements.	Medium-to-high complexity

Table 1: Documentation Complexity

In particular, we are talking about how well the application architecture and implementation are documented, not user documentation. In all too many cases, applications begin life as a “throw something together by next Monday” project, with an expected lifetime of weeks, then the application gets expanded, re-purposed, maintained, expanded again, etc... all without a shred of documentation.

ARCHITECTURE	MIGRATION COMPLEXITY
Well-architected, layered application.	Low complexity
Organically grown, non-layered code and architecture, combined with need to refactor.	Low-to-medium complexity

Table 2: Architecture Complexity

TECHNOLOGIES	MIGRATION COMPLEXITY
Familiarity with tc Server™ and/or lightweight technologies such as Spring Framework.	Low complexity
Strong organizational support of legacy EJB and full stack Java EE technologies; tc Server™ and/or lightweight technologies such as Spring Framework not yet adopted.	Medium complexity

Table 3: Technology Complexity

INTEGRATION	MIGRATION COMPLEXITY
No integration with proprietary application server frameworks or APIs.	Low complexity
Integration with proprietary application server frameworks or APIs.	Low to high complexity depending on extent of usage

Table 4: Integration Complexity

EJBS	MIGRATION COMPLEXITY
No reliance on session EJBs, or reliance only on a straightforward use of session EJBs, for example, in a small quantity or delegating to plain old Java objects (POJOs).	Low complexity
Heavy use of session EJBs.	Medium complexity
Reliance on stateful middle tier clustering (stateful session EJBs).	Medium complexity
Need for distributed transactions.	Medium-to-high complexity
Straightforward DAO-based database access (using either JDBC or ORM).	Low complexity
Reliance on entity beans.	Medium-to-high complexity depending on amount of code
Servlet-spec security usage.	Low complexity
Declarative EJB (container-managed) security usage.	Medium complexity, or low-to-medium complexity if refactoring to Spring Security
Using existing full stack application server's built-in JMS provider	Low-to-medium complexity depending on ability to use external commercial or open source JMS container. Generally only licensing (no code changes) concerns.

Table 5: EJB Complexity

3. JEE Specification and JEE Application Servers

The full JEE specification is huge, and today's commercial JEE Application Servers include many features and functions that go even beyond the JEE specification, so it is very important to understand what parts of the JEE specification functionality are native to tc Server, what portions are contained in optional packages, and what portions are outside the JEE specifications. Both JEE Application Servers and the JEE standard itself have evolved over more than a decade; the evolution was highly driven by the most demanding (and often esoteric) customer requirements. As with many mature software projects, JEE and JEE Application servers have now evolved to "do absolutely everything for absolutely everyone" (or at least they keep trying), and this has resulted in highly complex, huge, monolithic products.

- Even more important than the specification, or application server features, is understanding what parts of all that complexity you are actually making use of in the application. Experience shows that even sophisticated “JEE Applications” typically use very small parts of an Application server’s capabilities and typical “JAVA web applications” use less than 10-15 percent of a WLS or WAS application server. This has several implications:
- Much of the capability (and cost and complexity) of the JEE Application Server is effectively wasted.
- Hundreds of megabytes of un-needed footprint and complexity reduce developer productivity, and significantly increase IT operations/maintenance costs.
- You actually will only need very small portions of the total JEE Application Server functionality to support your application. tc Server, along with carefully selected modular extensions, provides a very productive and efficient alternative infrastructure for those applications.

In this whitepaper we will explore what portions of the JEE specification are covered by tc Server™ and what to do about those JAVA services you may need in the migration project. We’ll also discuss some of the functionality that goes “beyond the specification”, in particular in administration and management.

4. Java EE 5.0 API Support

The Java EE 5.0 specification includes a number of Java Optional Packages. For each package, the specification defines a required version with respect to environments that support a full-stack Java EE 5.0 environment. A tc Server™ environment provides access to many of these APIs, as a standard capability, or by enabling you to “drop in” standard implementation JARs to the Tomcat lib directory, or by bundling the standard implementation JARs with the deployed applications. In some cases, commercial alternatives also exist.

It is a definite advantage of the tc Server™ environment that many of these packages are not embedded out-of-the-box. This is because, as previously noted, most applications only utilize a small portion of the JEE standard, and even less of the WLS/WAS proprietary functionality. You can add needed capabilities to the environment easily, in “drop-in” fashion, which means you run the latest version of the specifications and you significantly reduce server bloat (from nearly a GB to tens of MB) by bringing in only required capabilities. Additionally, you can choose among multiple service providers, and server operators and application deployers can decide whether to add a package as a standard server feature (by dropping it into the server lib directory), or to embed it in each application.

The following table summarizes the availability of the Java Optional Packages defined by Java EE 5.0, in the tc Server™ environment, in the same order as Table EE.6-1 in the Java EE 5.0 specification document.

JAVA EE API SUPPORT	TC SERVER™
EJB 2.1 and 3.0	Client access only. No support for running EJBs themselves.
Servlet 2.5	Standard
Java Server Pages (JSP) 2.1	Standard
JavaMail 1.4	Drop in standard JAR to Tomcat lib directory or embed in applications.

JAVA EE API SUPPORT	TC SERVER™
JMS 1.1	Client: Drop in standard JAR. Server: No embedded implementation. Multiple third-party (open-source or commercial) products available to run alongside or embedded.
JTA 1.1	No embedded implementation. Plug-in external JTA provider if needed.
JAF 1.1	Drop in standard JAR to Tomcat lib directory or embed in applications.
Connector 1.5	Not supported
Web Services 1.2	Drop in standard or third-party JAR. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.
JAX-RPC 1.1	Drop in standard or third-party JAR. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.
JAX-WS 2.0	Drop in standard or third-party JAR. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.
JAXB 2.0	Drop in standard JAR. Add standard JAR to Tomcat lib directory or embed in applications.
SAAJ 1.3	Drop in standard JAR to Tomcat lib directory or embed in applications.
JAXR 1.0	Drop in standard JAR to Tomcat lib directory or embed in applications.
Java EE Management 1.1	Not supported
Java EE Deployment 1.2	Not supported
JACC 1.1	Not supported
JSP Debugging 1.0	Standard
JSTL 1.2	Standard

JAVA EE API SUPPORT	TC SERVER™
Web Services Metadata 2.0	Drop in standard JAR to Tomcat lib directory or embed in applications.
JSF 1.2	Drop in standard or third-party JAR. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.
Common Annotations 1.0	Standard. Supported in Servlets per specification.
StAX 1.0	Drop in standard or third-party JAR. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.
Java Persistence 1.0	Standard or third-party drop-in, including updated 2.0 spec. Multiple open-source and commercial implementations available. Add JARs to Tomcat lib directory or embed in applications.

5. Code Migration Strategies

The sections that follow examine various aspects of the code migration process, including layers, transactions, EJBs, remoting, data sources, security, messaging, and management.

5.1 Layering and Interfaces

A properly layered, interface-driven, and decoupled architecture that uses inversion of control and dependency injection patterns offers tangible benefits:

- Reduced code coupling
- Reduced complexity
- Easier and enhanced testing
- Increased agility

These benefits facilitate code refactoring. An existing architecture that is not adequately layered and interface-based requires additional refactoring to make it so. Consider whether this additional refactoring, above and beyond what is immediately needed for the migration effort, will pay off in future ongoing maintenance and enhancement of the application.

5.2 Migrating Significant Code-Level Changes Piece-by-Piece

If you need to migrate code-level changes, consider refactoring and migrating piece by piece instead of in a “big bang” process, as this is generally less disruptive and more manageable. You can work in multiple iterations, possibly with working code between iterations, until all refactoring is completed.

Where migration involves refactoring away from technology that is not available in the target environment, with the goal of working code at the end of each iteration, the migration effort should remain in the source environment until such technology use is completely refactored. For example, consider an existing application that implements business services as session EJBs, currently running in a full-stack Java EE environment, that are being refactored to execute in a tc Server™ environment. Migration in pieces allows only a subset of the EJBs to be refactored in each iteration (to POJO implementations of those services); this does imply staying in the source full-stack environment until all EJBs are refactored if the goal is to test working code between each iteration.

Migration in pieces usually requires one of two strategies: migration by vertical slice, or migration by layer. In the migration by vertical slice approach, a complete “vertical slice” encompassing multiple layers (services, data access, domain, and so on) often representing a logical application component, is migrated in one iteration. Over multiple iterations, code gradually moves from legacy vertical slices to migrated vertical slices. In the migration by layer approach, code is migrated a layer (for example, services) at a time. The decision to use a migration by vertical slice or migration by layer approach is application-specific and depends on existing architecture and existing code coupling. In both cases, the idea is to perform refactoring in manageable chunks.

5.3 Service Layer Migration Strategy

Many larger Java applications have a distinct set of functionality that implements a service or application layer, to the extent that this code implements use cases or functionality specific to the particular application. In best-practice fashion, this layer typically is hidden behind use case-specific interfaces, and often interacts with a data access layer (by interface), and domain objects or data transfer objects.

In applications that use EJBs, it is typical for the service layer to be implemented as EJBs, and readers are directed to consult later sections on migrating EJBs, when migrating applications that implement a service layer through EJBs.

Usually transactions and method level security are effected in the service layer. See Migrating Transactional Code.

Where a more than minimal refactoring effort is required in the service layer anyway, consider whether to expense potentially additional effort towards effecting a properly layered and interface based codebase, as described in a preceding section. This effort will pay off in the migration effort itself and/or in subsequent maintenance costs.

5.4 Data Access Layer and Data Access Objects

Many larger Java applications have a distinct set of functionality that implements a data-access layer, to the extent that this code is responsible for persisting data transfer objects or domain objects to one or more datastores. Typically, code in this layer is hidden behind a set of interfaces that abstract the exact persistence technology in use. However, in practice, it is not practical to try to abstract away the use of lower level persistence technology such as JDBC, as opposed to higher level technology such as object-relational mapping (ORM). Code in this layer is also often referred to as data-access objects (DAOs).

Where data access functionality is implemented through entity EJBs, it is necessary to migrate this code as described in EJB Entity Bean Migration Strategy.

5.5 Migrating Web and Presentation Layer Code

Migration to tc Server™ does not normally entail significant refactoring of web tier or presentation layer code. In many case, web tier code can remain untouched. Areas that may require refactoring are related to EJB access and to remoting. Where web tier code is a client of EJB services, and is directly performing EJB stub lookup, a straightforward conversion process to the use of plain Java services may be required. Where code is using EJB remoting, and continued use of remoting is actually necessary, refactoring to an alternate remoting technology is necessary. See EJB Client Access Code Migration Strategy and Remoting.

5.6 Migrating Transactional Code

Before migrating existing transactional code to a tc Server™ environment, it is important to understand the options available in a tc Server™ environment for demarcating and driving transactions in application code.

- Local resource transaction APIs: Transactions are driven through the appropriate local resource transaction APIs on local resources such as the JDBC Connection object, JPA's EntityManager, Hibernate's Session, or equivalent.
- JTA (Java Transaction API): tc Server™ out-of-the box does not include a transaction manager implementation that supports JTA APIs. However, you can plug in a third-party external JTA transaction manager (such as Atomikos) to the tc Server™ environment to support JTA APIs. Because a third-party JTA transaction manager is an additional component that must be obtained, supported, and possibly licensed, examine the desirability of this approach against the alternative of refactoring code away from the use of JTA.

Although application-specific code may drive transactions using these APIs, it is common for application code to work instead through a third-party abstraction layer which itself talks to these APIs. As an example, Spring Framework offers transaction functionality (see the next section), which allows both a declarative and programmatic approach to transaction demarcation, and abstracts away the distinction between JTA and local resource transactions.

5.6.1 JTA: Non-EJB and EJB Bean-Managed Transactions

tc Server™ out-of-the-box does not include a transaction manager implementation that supports the JTA APIs. You can choose among several approaches to migrating application code that uses JTA (whether POJO services or EJBs using bean-managed transactions):

- The most direct migration approach, often requiring no code level refactoring, is to plug in an external JTA transaction manager (such as Atomikos). Because a third-party JTA transaction manager is an additional component that must be obtained, supported, and possibly licensed, examine the desirability of this approach against the alternative of refactoring code away from the use of JTA.
- Eliminate JTA API usage by refactoring code to use corresponding local resource transaction APIs. This would include, for example, direct driving of transactions through the JDBC Connection object, JPA's EntityManager, Hibernate's Session, or equivalent. Application code that uses local resource APIs should essentially run as-is in a tc Server™ environment.
- Refactor to use the Spring Framework transaction management functionality and abstractions. With this approach, code is decoupled from the exact transaction management strategy, and works equally well with JTA and local DataSource transactions. It also provides integration with lower level, data-access support infrastructure.
 - When moving code that drives transactions programmatically, refactor according to Spring's programmatic TransactionTemplate.
 - Alternatively, consider refactoring to the use of Spring's declarative transaction support. In this variant, transactions are demarcated via external (to the code) XML-based declarations, or internal annotations right in the source. While moving to this declarative approach will require additional work, transaction handling boilerplate will be removed from application code, with the general net benefits of significantly simplifying and clarifying application logic, reducing the amount of code that must be maintained, and reducing the chance of bugs.

5.6.2 EJBs with Container-Managed Transactions (CMT)

See Session EJB Migration Strategy.

5.7 Session EJB Migration Strategy

EJB 2.x session beans are not available in a tc Server™ environment. A migration to tc Server™ entails refactoring application logic currently implemented as session beans. EJB 2.x session beans are generally considered dead-end, legacy technology, which is another factor that drives refactoring of this code.

The general approach to refactoring session beans is to re-implement the logic as POJO services. General concerns stem from handling transactions for the session beans, and to a lesser degree, the handling of security.

5.7.1 Session Beans with Container-Managed Transactions (CMT)

CMT implies that transactions are applied to EJB code through a declarative process, where the EJB deployment descriptors are annotated to indicate that the container should make certain methods transactional when invoked. The use of CMT must be refactored by one of several approaches:

- Where Spring Framework is an available option, the preferred approach is to refactor to use Spring's declarative transaction functionality, as there is essentially a one-to-one mapping to the semantics of applying transaction demarcation. Spring's transaction support is a complete functional superset of that available in an EJB container. It can drive transactions through both JTA and local resource APIs, and contains additional integration with Spring's data access support code. Spring supports declarative demarcation of transactions through either external (to the code) XML-based declarations, or internal annotations right in the source.
- Alternatively, you can implement transactions programmatically (similar to bean managed transactions), where application code explicitly drives transactions. Spring's programmatic TransactionTemplate is a good choice, as you can decouple application code from the specific JTA or local DataSource transaction APIs. Additionally, Spring provides integration with lower-level data access support functionality. Otherwise, application level code may drive transactions by using the specific transaction API available. This includes JTA where an external JTA transaction manager has been plugged into the tc Server™ environment, or a local resource transaction API that drives transactions directly through the JDBC Connection object, JPA's EntityManager, Hibernate's Session, or equivalent.

5.7.2 Session Beans with Bean Managed Transactions (BMT)

See JTA: Non-EJB and EJB Bean-Managed Transactions, as the same general approach applies to migrating BMT code to tc Server™ as POJO code, whether using JTA or local resource APIs.

5.7.3 Session Beans with Container-Managed Security

You can annotate EJBs in a declarative fashion in their XML descriptors to indicate that the container should apply simple role-based security to certain EJB methods. You need to refactor container-managed security by one of these approaches:

- The most direct approach is to refactor to the use of declarative security through Spring Security. Spring Security, an extensible security framework, generally offers a complete superset of capabilities offered by EJB container-managed security, while also offering a large number of other capabilities, and being extensible in a container-agnostic fashion.
- Alternatively, you can replace container-managed security with custom application-specific code created during refactoring, or by the use of another security framework.

5.8 Message-Driven EJB Migration Strategy

EJB message-driven beans (MDBs) are not available as a technology choice in a tc Server™ environment. A migration to tc Server™ entails some refactoring away from MDBs. However, because code inside MDBs does not have a significant dependency on the EJB container (as opposed to other types of EJBs), the preferred refactoring approach requires minimal (if any) code-level changes.

- The preferred approach is to refactor to use Spring Framework's Message Driven POJO (MDP) functionality. MDBs are not inherently tied to the EJB container at the code level, because they are generally a class implementing the JMS MessageListener interface. Spring's MDP container functionality can also handle message-dispatching (including all needed internal threading and looping functionality) to MessageListeners. Refactoring to the use of Spring does not generally imply code level changes, but rather configuration of the Spring MDP container. Spring's MDP capabilities are a functional superset of EJB MDBs, but offer a number of additional capabilities, including the ability to drive messages to complete POJOs that do not even implement MessageListener and to allow the automatic conversion of a JMS message to other data types, when receiving or sending messages.
- Without Spring Framework, refactoring requires the creation of additional application-level code to handle the looping and threading functionality around message reception. This functionality was previously handled by the EJB MDB container.

5.9 Entity EJB Migration Strategy

EJB 2.x entity beans are not available in a tc Server™ environment. Migration to tc Server™ requires refactoring of persistence logic currently implemented as entity beans. Entity beans are considered dead-end, legacy technology, and their use fell out of favor years ago. Where legacy applications still use entity beans, their disadvantages are a good argument for refactoring.

The goal in refactoring away from entity beans is to end up with an interface-based, decoupled data-access layer, so that to the extent possible, higher layers are mostly agnostic to actual persistence choices. Three technologies exist for implementing persistence logic in the data access layer:

- JDBC: JDBC is a lower-level approach but is a strong choice in some scenarios. Consider the use of Spring Framework's JDBC support, which eliminates most resource management and try/catch boilerplate typically associated with JDBC, while also offering low-overhead mechanisms to implement mapping logic. Some scenarios lending themselves to JDBC include:
 - Higher transaction volumes and/or simpler object graphs
 - Legacy schemas not amenable to object relational mapping (ORM)
 - Strong SQL and/or DBA skills in-house
 - Use of stored procedures
- Hybrid JDBC/ORM: The best example of this approach is iBatis, from Apache.org. iBatis essentially externalizes input/output and mapping logic to external files containing the SQL, while also offering some elements from object relational mappers, such as declarative mapping of data types, and automatic eager or lazy loading of related entities. iBatis does not perform change tracking in the nature of a true ORM. However, Spring Framework's iBatis support eliminates most resource management and try/catch boilerplate that would otherwise have to be used with iBatis.
- ORM frameworks: ORM frameworks, such as Java Persistence Architecture (JPA), Java Data Objects (JDO), or defacto-standards such as Hibernate, employ a higher-level, declarative approach to persistence, where the framework performs mapping between Java objects and the database, tracks changes to entities while in use, and handles related functions such as eager or lazy-loading of related entities. Some scenarios lending themselves to ORM use include:
 - Somewhat lower transaction volumes and/or more complex object graphs
 - No legacy schemas with use of DB features or relationships precluding use of ORM
 - Weaker SQL and/or DBA skills in-house
 - Limited or no use of stored procedures

5.10 EJB Client Access Code Migration Strategy

Client code that accesses EJB functionality sometimes accesses EJBs through an abstraction or lookup layer, so that the client code is only aware of plain Java business interfaces. In general, this type of client code does not require significant refactoring when you are moving away from EJBs.

Where EJB client code has direct awareness of service EJBs, EJB stubs during lookup through JNDI, EJB exceptions, or EJB remote interfaces, you must refactor the client code as part of refactoring the service EJB code to plain Java service implementations. For migrations in which Spring Framework is available and many EJBs exist, consider migrating EJB client code to use Spring's EJB proxies as an initial step. Client code working through Spring's EJB proxies is generally unaware of EJBs, and is concerned simply with business interfaces. The decoupling achieved in this fashion will then facilitate the Migrate in Pieces migration strategy, as it allows client code to be unaware of when exactly EJB service code is migrated.

6. Remoting Services

Existing application code that accesses or implements non-EJB remoting services through RMI can migrate essentially unchanged to a tc Server™ environment.

Convert remote EJBs to POJO services as described in the preceding section on migrating EJBs, and where remoting is still needed, implement remoting using RMI or another appropriate protocol.

Where remoting is desired, and some refactoring is needed regardless, consider using Spring's transparent remoting functionality. Spring Remoting allows POJO services to be exported in a declarative fashion, over a number of wire protocols, without the services having to be aware of or coupled to the remoting protocol. Similarly, on the client, side, Spring can create client-side remoting proxies such that client code calling through the proxies is unaware that it is talking to anything other than a local business interface.

7. DataSources

In a full stack application server environment, DataSources are typically application server-managed connection pools bound to the JNDI tree. When migrating to tc Server, you can bind an instance of the high concurrency connection pool provided with tc Server™ to tc Server's JNDI tree, allowing client code to work as-is. Another common connection pool in usage in Tomcat environments is Apache Commons database connection pool (DBCP), although the high concurrency pool is recommended for its higher scalability.

It is also quite common to embed the connection pool in the deployed application itself, where it is a product, such as DBCP, that allows this scenario. DBCP makes this process very easy, because you can create and configure it as a simple JavaBean.

8. Security

Existing code utilizing Servlet spec security in the web layer will normally work unchanged in a tc Server™ environment. That said, it is quite rare for any real world application to depend on the Servlet spec security model and all JEE server vendors have augmented JEE security by adding sophisticated user privilege management services to their servers, as well as offering stand alone security products compatible with their JEE servers.

It is slightly more common for applications based on WAS to utilize container-managed security for EJBs, although even there it is more common to utilize IBM's widely adopted Tivoli Access Management security

or any of a number of third party application security products. In the case of WLS, BEA incorporated a comprehensive user/application security service directly into the server, and also offers a commercial upgrade, WebLogic Enterprise Security (WES). Both IBM Tivoli and Oracle/BEA WES work with Tomcat as does CA's SiteMinder, and user level security generally transfers fairly smoothly

Object level security, provided by Oracle's BEA WebLogic Server, transports less well, primarily because access control at that level requires both tight integration within the container and distributed rules processing, neither available with Tivoli nor SiteMinder. Perhaps fortunately, object level security never caught on and most developers continue to hard-code such rules directly into their applications anyway.

9. Messaging

The tc Server™ environment does not provide an embedded message queue. As such, when moving applications from a full-stack application server environment where an embedded message queue in that environment is being used, an external message queuing product must be used alongside tc Server. A number of options are available to you, both open source (for example, ActiveMQ) and commercial.

Convert EJB message-driven beans as described in Message-Driven EJB Migration Strategy. Where messaging-related code is being refactored, consider using Spring's extensive messaging functionality, to reduce messaging boilerplate in general, and to take advantage of Spring's out-of-the-box container managed Message Driven POJO capability.

Other messaging code should normally work unchanged in the tc Server™ environment, once an external queue is plugged in, with the caveat that any vendor-specific code must be converted to the new message queuing product.

10. Administration and Management

Completely separate from the JEE specification (and other services) provided by the commercial JEE servers, these products also provide sophisticated management capabilities that are used by developers and IT administrators alike.

Commercial JEE Application Servers provide Administration and Management in two areas, Server and Application. Oracle/BEA WebLogic sets the standard for such services, with IBM somewhat more limited and SUN, jBoss, etc providing relatively basic capabilities. In the case of WLS, the administrative application is more than 50% of the total product footprint. Tomcat lacks most such capabilities, which left the IT organization facing "do it ourselves" or third party products as the alternatives. SpringSource's tc Server™ includes sophisticated Server and Application control and monitoring, effectively bringing Tomcat into the "Enterprise Web Application Server" marketplace.

10.1 Server Administration and Management

While a complete discussion of JEE Application servers administration and management is far beyond the scope of this whitepaper (indeed, there are full books written on the topic), it is important to note that the primary objective of all that functionality is to enable an IT administrator to securely install, configure, monitor, and control application servers throughout the enterprise, utilizing a common GUI and scripting capability. This eliminates the laborious and error prone need to treat servers as individuals and manually deal with each server.

When the number of servers starts growing large (enterprise class use), manual processes become prohibitive. Likewise, while monitoring the behavior of a few servers is possible, though labor intensive, manually monitoring dozens/hundreds/thousands of instances is simply impossible.

As noted above, Tomcat lacks even the most basic of these capabilities, while tc Server™ provides both GUI and script based server management.

10.2 Application Administration and Management

Even more important than the situation with server administration and management is application administration and management. This is because there are far more applications than servers, they change more frequently, and their performance is more affected by dynamic use conditions.

Once again, Tomcat lacks even the most basic functionality to manage applications, but tc Server's ability to manage and monitor applications across the enterprise enables IT organizations to use Tomcat in enterprise scale application deployments. We will discuss the advantages of tc Server™ over Tomcat in more detail in other whitepapers in this series.

11. Conclusion

Many JEE applications can be migrated to run on Tomcat, by utilizing the capabilities in tc Server™ and by adding selected services as needed. Some of these migrations are quite straight forward, depending on what JEE Application Server services have been utilized. Others may require more significant investment, although the benefits often justify even the larger scale projects.

We've seen projects migrate from WLS/WAS to tc Server™ in a few person months, where they used only servlet functionality, and other projects utilizing JEE entity beans and session beans taking multiple person years. Projects already implemented utilizing the Spring Framework will generally be somewhat simpler to migrate, because the Spring abstraction insulates the application code itself from many of the idiosyncrasies of any individual commercial JEE Application Server. With careful development planning, the risks can be assessed prior to significant project investment and good cost/benefit decisions made.

