Red Hat JBoss Web Framework Kit 2.7
Spring Guide

---

for use with Red Hat JBoss Enterprise Application Platform

Red Hat Customer Content Services

Red Hat JBoss Web Framework Kit 2.7 Spring Guide

for use with Red Hat JBoss Enterprise Application Platform

Red Hat Customer Content Services

Legal Notice

Abstract

This guide is an introduction to developing Spring-based applications for the Red Hat JBoss Enterprise Application Platform.

Table of Contents

Chapter 1. Spring Future Support

Red Hat JBoss Web Framework Kit 2.x has a published end-of-life date of June 2015. For more information, see https://access.redhat.com/support/policy/updates/jboss_notes.

Red Hat continues to test Spring for use with Red Hat JBoss Enterprise Application Platform. Future Spring testing and support will be part of JBoss EAP releases. For more information, see https://access.redhat.com/solutions/1314313 on the Red Hat Customer Portal.

Report a bug

Chapter 2. Introduction to Red Hat JBoss Web Framework Kit

2.1. About Red Hat JBoss Web Framework Kit

Red Hat JBoss Web Framework Kit is a set of enterprise-ready versions of popular open source frameworks. Together, these frameworks provide a solution for developing light and rich Java-based web applications.

JBoss Web Framework Kit comprises enterprise distributions of JBoss community web application frameworks and tested third-party frameworks. These leading frameworks support fast and easy client-side and server-side application development and testing, with frameworks including RichFaces, jQuery, jQuery Mobile, Hibernate Search, Spring, and Arquillian.

The breadth of JBoss Web Framework Kit frameworks provides choice for Java application development. Each framework has been tested and certified for use in applications deployed to Red Hat JBoss Enterprise Application Platform, Red Hat JBoss Web Server, or Red Hat OpenShift JBoss EAP cartridge. Inclusion in JBoss Web Framework Kit ensures stable versions of frameworks are available over long-term enterprise product life cycles, with regular releases for fixes and nonintrusive feature updates. Further, Red Hat JBoss Developer Studio (an Eclipse-based development environment) provides integrated project templates, quickstarts and tooling for many of the JBoss Web Framework Kit frameworks.

For the complete list of the frameworks composing JBoss Web Framework Kit and the certified platform and framework configurations, see https://access.redhat.com/site/articles/112543 on the Red Hat Customer Portal.

Report a bug

2.2. About the JBoss Web Framework Kit Tiers

The frameworks composing JBoss Web Framework Kit are categorized in four distinct tiers. A description of each tier and the associated Red Hat support is detailed here.

Tier 1 - Included Components

These are components that are based wholly or partly on open source technologies that support broad collaboration and where Red Hat maintains a leadership role; as such Red Hat is able to support these components and provide upgrades and fixes under our standard support terms and conditions.

Tier 2 - Tested Frameworks

These are third-party frameworks where Red Hat does not have sufficient influence and does not provide upgrades and fixes under our standard support terms and conditions. Commercially reasonable support is provided by Red Hat Global Support Services for these frameworks.

Tier 3 - Frameworks in Tested Examples

These are third-party frameworks where Red Hat does not have sufficient influence and does not provide upgrades and fixes under our standard support terms and conditions. Red Hat supports the examples these frameworks are used in and the generic use cases that these examples intend to demonstrate.

Tier 4 - Confirmed Frameworks

These are third-party frameworks that do not receive any support from Red Hat, but Red Hat verifies that the frameworks run successfully on Red Hat JBoss Enterprise Application Platform. Frameworks and versions not listed here have not been explicitly tested and certified, and thus may be subject to support limitations.

For a list of JBoss Web Framework Kit frameworks by tier, see https://access.redhat.com/site/articles/112543 on the Red Hat Customer Portal.

Report a bug

2.3. About the JBoss Web Framework Kit Distribution

The frameworks composing JBoss Web Framework Kit are distributed from a range of sources and in a variety of formats:

- The component frameworks are available from the Red Hat Customer Portal. They are distributed in two alternative formats: a binary for each framework or together as one Maven repository. In addition, the source code for each framework is provided for inspection.
- The third party frameworks are not distributed by Red Hat and each must be obtained from its own source.

A number of defined Maven JBoss stacks are provided as part of the JBoss Web Framework Kit distribution. All of the BOMs defining the JBoss stacks are available in the Maven repository `.zip` file available to download from the Red Hat Customer Portal or from http://www.jboss.org/developer-materials/ on the JBoss Developer Framework website.

An extensive set of examples are also provided as part of the JBoss Web Framework Kit distribution:

- TicketMonster is a moderately complex application demonstrating a number of the JBoss Web Framework Kit frameworks working together.
- Quickstarts and Maven archeypes illustrate subsets of the JBoss Web Framework Kit frameworks used to create simple applications.
- RichFaces, Snowdrop and Seam demonstrations showcase the power of each framework in web application development.

All of these examples are available from the Red Hat Customer Portal, with TicketMonster, the quickstarts, and the Maven archetypes also available from http://www.jboss.org/developer-materials/ on the JBoss Developer Framework website.

Report a bug

Chapter 3. Overview of Spring with Red Hat JBoss Enterprise Application Platform

3.1. About Spring Framework

Spring is an application development framework for enterprise Java. Spring is lightweight, has a modular architecture, helps you write easy to test applications, allows you to use Spring's configuration management services in any architectural layer, and provides an abstraction layer over transaction strategies. Use Spring framework to create high performing, easily testable, and reusable code.

Report a bug

3.2. About Spring and Red Hat JBoss Enterprise Application Platform

Spring directly and easily works on Red Hat JBoss Enterprise Application Platform with very little configuration change. The following table lists use cases of JBoss Enterprise Application Platform and Spring integration. These use cases are related to the Sportsclub example that is shipped with Web Framework Kit 2 distribution. The scenarios selected for this example are focused on using the Java EE 6 services, provided by JBoss Enterprise Application Platform, in Spring applications.

Table 3.1. Use cases

Category	Use case	How does this involve JBoss Enterprise Application Platform
Persistence	Spring/Hibernate integration	The application uses a Spring-configured Hibernate SessionFactory, using JTA transaction management and JTA-bound sessions. The Hibernate library is the one provided by JBoss Enterprise Application Platform.
	Spring/JPA integration	The Persistence Unit is deployed by JBoss Enterprise Application Platform and retrieved from JNDI to be injected into Spring beans. PersistenceContext is shared with surrounding EJBs (if any).

Category	Use case	How does this involve JBoss Enterprise Application Platform
Testing	Unit-testing components that have managed infrastructure dependencies	The DataSource and EntityManager are managed by JBoss Enterprise Application Platform and acquired from JNDI by Spring when the application is running. In the case of JBoss Enterprise Application Platform, developers can test their code in isolation using Spring-specific replacements that simulate the JBoss Enterprise Application Platform environment.
Business Logic	Spring-based service beans	The business services are Spring-managed and wrapped into transactions managed by Spring's interceptors. The TransactionManager in use is the JTATransactionManager using Transactions provided in JBoss Enterprise Application Platform.
	EJBs injected with Spring Beans	The application uses JBoss-deployed EJBs which are injected with Spring beans acquired from an application context bootstrapped by the Spring Deployer. Transactions are managed by EJBs.
User Interface	JSF/RichFaces and Spring integration	The application uses JSF support and RichFaces components provided by JBoss Enterprise Application Platform. The business services and UI-backing instances are Spring beans.
	Spring MVC and EJB integration	The application uses Spring MVC and the business logic is implemented using JBoss-deployed EJBs, which are injected into the Spring controllers.
	Conversation-oriented web application	The application uses Spring Web Flow for defining a conversational process for creating a reservation.

Category	Use case	How does this involve JBoss Enterprise Application Platform
JMS/JCA integration	JMS/Spring integration using JCA	The application uses Spring-configured message listeners for processing JMS messages from destinations managed by JBoss Enterprise Application Platform and Spring /JCA integration for receiving messages.
JMX	Spring beans are exposed as JMX beans	The JBoss Enterprise Application Platform MBean Server is used for registering the Spring-exported JMX beans. Consequently, the Spring beans can be managed from a management console.
Web Services	JAX-WS defined web-services are injected with Spring beans	The application uses JBoss Enterprise Application Platforms' support for JAX-WS through JBoss WS, but also Spring to define the underlying business logic, which is injected into the JBoss WS-deployed services.
	A Spring Web Services-based variant of the JAX-WS example	The application implements a web service based on Spring Web Services.
Security	Application-server pre-authentication	The application uses Spring Security for authorizing access to resources. Authentication is provided by the application server.

Report a bug

Chapter 4. Installing Spring

4.1. Prerequisites

Spring is certified for use with several Red Hat JBoss products. This chapter is intended to help you choose a certified distribution of the framework, and the most appropriate strategy of using it with your application and target platform.

System Requirements

The prerequisites for using Spring are as follows:

- Maven 3.0.4
- Red Hat JBoss Developer Studio 8
- Red Hat JBoss Enterprise Application Platform 6

Note

Throughout this guide, $EAP_HOME represents the top-level directory of the extracted distribution, which is usually jboss-eap-6.

Downloading Spring

A certified version of Spring can be obtained by downloading it from a public repository (Maven or Ivy). The following versions of Spring are certified for use with Red Hat JBoss products:

- 2.5.6.SEC03
- 3.0.7.RELEASE
- 3.1.4.RELEASE
- 3.2.13.RELEASE
- 4.0.9.RELEASE
- 4.1.4.RELEASE

Instructions on downloading and setting up repositories for the community versions of Spring are available at Spring Reference documentation. Though the instructions in the Spring Reference documentation are written taking Spring 3 into consideration, they are also applicable to Spring 2.5.6.SEC03 and Spring 4.

Warning

- Spring 2.5 is deprecated from the release of JBoss Web Framework Kit 2.4.0. This means that Spring 2.5 will continue to be tested and supported in JBoss Web Framework Kit 2.x but it will not be tested or supported in JBoss Web Framework Kit 3 and later. Note that subsequent Spring 2 releases will not be tested or supported in JBoss Web Framework Kit.
- Spring 3.0.x and 3.1.x are deprecated from the release of JBoss Web Framework Kit 2.5.0. This means that Spring 3.0.x and 3.1.x will continue to be tested and supported in JBoss Web Framework Kit 2.x but they will not be tested or supported in JBoss Web Framework Kit 3 and later. Spring 3.2.x and later are not deprecated and continue to be tested and supported in JBoss Web Framework Kit.
- Support for Spring 4 as a tested framework is limited to the features supported by Red Hat JBoss Enterprise Application Platform. Specifically, support is not provided for the new features in Spring 4 related to Java SE 8 and Java EE 7.

When using Spring in an application, the options for packaging and distributing the framework are:

- Installing the framework libraries as shared libraries on the platform
- Packaging the framework libraries together with the application (in application archives)

Report a bug

4.2. Spring as Shared Libraries

One method of using Spring inside Red Hat JBoss Enterprise Application Platform is by installing it as a shared library. This strategy is preferred when multiple applications using Spring are required to run on the same application server, as this strategy helps you to avoid including JARs in each deployed application. To install Spring as a shared library, create a Spring module in the **`EAP_HOME/modules/system/add-ons`** directory of the application server. A JBoss Enterprise Application Platform module is identified or referenced by its *identifier* and *slot*. This section, uses a module with the **`org.springframework.spring`** identifier, and the **`main`** slot. This is the default slot: if a slot name is unspecified, JBoss Enterprise Application Platform assumes a slot name of 'main' for identifying the module.

Procedure 4.1. Installing Spring as a Shared Library

1. Create **`EAP_HOME/modules/system/add-ons/org/springframework/spring/main`** directory.

   This path consists of a main directory (**`org/springframework/spring`**) which reflects the module identifier and a subdirectory (**`main`**) which reflects the slot name.

2. Copy the Spring JARs, to be shared, in the **`EAP_HOME/modules/system/add-ons/org/springframework/spring/main`** directory.

3. Create a **`module.xml`** descriptor, which includes references to the Spring JARs and application server dependencies to the module. Each JAR copied to the module folder must have a distinct **`<resource-root>`** entry in the module descriptor.

   Example 4.1. Example module.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0"
name="org.springframework.spring" slot="main">
    <resources>
        <resource-root path="aopalliance.jar"/>
        <resource-root path="aspectjrt.jar"/>
        <resource-root path="aspectjweaver.jar"/>
        <resource-root path="spring-aop.jar"/>
        <resource-root path="spring-beans.jar"/>
        <resource-root path="spring-context.jar"/>
        <resource-root path="spring-context-support.jar"/>
        <resource-root path="spring-core.jar"/>
        <resource-root path="spring-expression.jar"/>
        <resource-root path="spring-web.jar"/>
    </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
    ...
  </dependencies>
</module>
```

The module name and slot must match the directory structure of the module.

JBoss Enterprise Application Platform allows you to create multiple Spring modules, with different identifier or slot name combinations. Thus, your applications can share multiple Spring versions at the same time, and the Spring versions do not interfere with each other. Also, you applications can point to any of the Spring modules.

Determining the JARs That Need to Be Copied

The set of JARs that you must install as shared libraries are the Spring JARs on which your applications are dependent. Also include AOP Alliance APIs and AspectJ libraries, if necessary.

Implications of Packaging the Application

If you install Spring libraries as shared libraries inside the application server, do not included the libraries in the deployed applications as that may lead to classloading errors. For Ant builds, do not include the Spring libraries in the **<lib/>** element of the WAR or EAR tasks. In Maven builds, set the scope to *provided*.

Report a bug

4.3. Referencing the Spring Module

If you install Spring as a module, you can voluntarily included it in applications through the **/META-INF/jboss-deployment-structure.xml** or **/WEB-INF/jboss-deployment-structure.xml** descriptors as follows:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
 <deployment>
  <dependencies>
```

```
    <module name="org.springframework.spring" slot="<slot-name>">
     <imports>
      <include path="META-INF**"/>
      <include path="org**"/>
     </imports>
    </module>
   </dependencies>
  </deployment>
</jboss-deployment-structure>
```

The **<imports>** element in the descriptor is critical for the proper functioning of the Spring custom namespace capabilities. For applications that use custom namespaces and require a shared Spring module, you must use the JBoss Enterprise Application Platform deployment structure descriptors with the **<imports>** element. Applications can also reference the module through manifest by adding the following to the **META-INF/MANIFEST.MF** file:

```
Dependencies: org.springframework.spring:<slot-name>
```

However, changes to the **META-INF/MANIFEST.MF** file does not allow the usage of custom Spring namespaces.

Report a bug

4.4. Benefits and Drawbacks of Spring as Shared Library

Installing the Spring libraries as shared libraries is an option you must consider when multiple applications that use the same version of Spring are to be deployed on the same application server instance. However, this method has its advantages as well as disadvantages.

Advantages

- Reduces the size of the packaged applications(EARs, WARs, RARs)
- Reduces the number of times classloaders load Spring classes
- Provides a simple path to upgrade to a different version of Spring (a single place to do the replacement)

Disadvantages

- Flexibility considerations:

  The Spring module gets integrated with JBoss Enterprise Application Platform module system, and the dependencies of Spring modules are required to be reflected in the dependency system of the applications using it. As a result, the application becomes less isolated from rest of the application server. This is not necessarily a disadvantage but an item to consider when sharing Spring.

- Build Requirements:

  All applications deployed on the server must be built in a way that does not package Spring.

- Operational Issues:

  Installing the shared Spring libraries becomes part of the application server setup. The packaged applications do not contain the Spring libraries, and do not run on a server that does not contain the shared Spring libraries.

Report a bug

4.5. Spring in Application Archives

Packaging Spring in application archives is a more common approach than installing Spring as a shared library in the application server. In this method, the Spring files are required to be at the following locations inside the application archives:

In WAR

In the **`WEB-INF/lib`** directory.

In EAR

In the **`/lib`** folder of the archive, or another folder specified in the **`META-INF/application.xml`** descriptor.

In EARs that contain WARs, package Spring either in the shared library directory of the EAR, or in the **`WEB-INF/lib`** directory of individual WARs that use Spring. Unless the WARs use different versions of Spring (not a common case), Red Hat recommends you to package Spring as a shared library of the EARs.

In RAR

In the root directory of the archive.

Warning

The **`SpringContextResourceAdaptor`** present in Spring 3.0.7.RELEASE and Spring 2.5.6.SEC03 violates section 5.3.1 of the JCA 1.5 specification. As a consequence, RARs containing packaged Spring files from these versions cause deployment errors on Red Hat JBoss Enterprise Application Platform.

To work around this issue, disable archive validation in the JCA subsystem by adding the following code to the server configuration:

```
<subsystem xmlns="urn:jboss:domain:jca:1.1">
 <archive-validation enabled="false" />
</subsystem>
```

Report a bug

4.6. Benefits and Drawbacks of Spring in Application Archives

Packaging the Spring libraries in application archives has its advantages as well as disadvantages.

Advantages

- Every application deployed on the server can use a different version of Spring
- Build scripts are simpler

- Packaged applications can be deployed on different server instances, that is, the target server is not required to be prepared for a specific deployment

Disadvantages

- Size of the application archive increases
- Memory usage is not efficient as classes common to all applications are loaded multiple times

Report a bug

4.7. Using Spring With JBoss Spring Deployer

The Snowdrop distribution includes a Spring Deployer for Red Hat JBoss Enterprise Application Platform. The Spring Deployer requires Spring classes to instantiate ApplicationContexts. The Spring classes are made available to the Deployer in a separate **`org.springframework.spring`** module using the dedicated **`snowdrop`** slot. Refer the *Snowdrop User Guide* for more information on the Spring Deployer.

Report a bug

Chapter 5. Configuring Spring

5.1. Overview of Spring Configuration

Red Hat recommends the following configuration practices for using Spring with Red Hat JBoss Enterprise Application Platform.

This chapter details configuration strategy for the following:

- Datasources
- Hibernate SessionFactories
- JPA EntityManagers and EntityManagerFactories
- Transaction management

Report a bug

5.2. Setting Up Data Access

Spring applications either deploy their own data access infrastructure or rely on a managed infrastructure provided by the application server. For JBoss Enterprise Application Platform, Red Hat recommends you to use the infrastructure provided by the server or use a managed infrastructure using JTA transaction management.

Report a bug

5.2.1. Database Access Through Managed Datasources

On JBoss Enterprise Application Platform, you can configure managed datasources through the *datasource* subsystem and access it through JNDI. You can create a managed datasource by editing the **`EAP_HOME/standalone/configuration/standalone.xml`** configuration file either from the CLI or from the web console.

For example, when running in the standalone mode, you can add a datasource definition by including the following element inside the **`datasources`** subsystem.

Example 5.1. Managed Datasource Configuration in standalone.xml

```
<datasource jndi-name="java:jboss/datasources/ExampleDsJndiName" pool-
name="ExampleDS" enabled="true">
    <connection-url>jdbc:h2:mem:exampleDS</connection-url>
    <driver>h2</driver>
    <security>
        <user-name>sa</user-name>
    </security>
</datasource>
```

The datasource is bound in JNDI at **`java:jboss/datasources/ExampleDsJndiName`**. You can reference the datasource from a Spring ApplicationContext by using the definition in the example that follows. You can inject a datasource bean into any regular Spring bean (for example, a JBDS DAO).

Example 5.2. Defining a Managed Datasource Spring Bean

```
<jee:jndi-lookup id="dataSource" jndi-
name="java:jboss/datasources/ExampleDsJndiName" expected-
type="javax.sql.DataSource"/>
```

Report a bug

5.2.2. Hibernate SessionFactory

Applications that use Hibernate directly (as opposed to JPA) either use the Hibernate version included in the application server (for Spring versions which are compatible with Hibernate 4 and higher) by referencing it, or package a supported version of Hibernate 3 (for Spring versions that require Hibernate 3).

Spring applications can use one of Spring's SessionFactory-instantiating FactoryBeans, and a managed datasource to use Hibernate.

Example 5.3. SessionFactory Bean Definition

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="datasource"/>
       ...
</bean>
```

Red Hat recommends JTA transaction management and Hibernate-JTA session management integration. You can use these by setting the following properties:

Example 5.4. JTA session management setup properties with Hibernate

```
hibernate.current_session_context_class=jta
hibernate.transaction.jta.platform=org.hibernate.service.jta.platform.i
nternal.JBossAppServerJtaPlatform
hibernate.transaction.factory_class=org.hibernate.transaction.JTATransa
ctionFactory
```

Once you define a SessionFactory, you can inject it in component classes such as DAOs and use it directly as in Example 5.5, “Hibernate-based DAO: a SessionFactory is Injected Directly in the Bean”:

Example 5.5. Hibernate-based DAO: a SessionFactory is Injected Directly in the Bean

```
public class HibernateAccountDao {
  @Autowired SessionFactory sessionFactory;

  public List<Account> getAllAccounts() {
     return
```

```
sessionFactory.getCurrentSession().getCriteria(Account.class).list();
  }
  ...
}
```

The example is available in the Red Hat JBoss Web Framework Kit distribution, and the detailed instructions about using the example are available in the Snowdrop Sportsclub Example User Guide.

Report a bug

5.3. Using JPA

Spring provides the following three methods of implementing Java Persistence API:

- PersistenceUnit Deployed by the Container
- PersistenceUnit Created by Spring
- PersistenceContext and PersistenceUnit Injection

Report a bug

5.3.1. PersistenceUnit Deployed by the Container

Spring applications can retrieve the persistence units deployed by the container by looking them up in JNDI. In order to bind the entity manager or entity manager factory under a well-established name and subsequently look them up, applications can define them in the **`web.xml`** file.

Consider the following persistence unit defined in the **`persistence.xml`** file:

Example 5.6. Persistence Unit Definition

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
version="2.0">
    <persistence-unit name="sportsclubPU">
        <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
    </persistence-unit>
</persistence>
```

In this case, you can bind the persistence context in JNDI as follows:

Example 5.7. Persistence context binding in JNDI

```
<web-app version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    metadata-complete="true">
```

```
    <persistence-context-ref>
        <persistence-context-ref-name>example/em</persistence-context-
ref-name>
        <persistence-unit-name>sportsclubPU</persistence-unit-name>
    </persistence-context-ref>
</web-app>
```

Spring applications can use either container-managed entity managers or application-managed entity managers. In this case, entity managers are provided with the corresponding entity manager factories. You can access a container-managed entity manager as a Spring Bean, as shown in the following example.

Example 5.8. Spring Bean Representing a Container-managed Entity manager

```
<jee:jndi-lookup id="entityManager" jndi-
name="java:comp/env/example/em"/>
```

You can use this EntityManager directly when it is injected in component classes such as DAOs.

Example 5.9. Hibernate-based DAO: a SessionFactory is Injected Directly in the Bean

```
public class HibernateAccountDao {
  @Autowired EntityManager entityManager;

  public List<Account> getAllAccounts() {
     return entityManager.createQuery("SELECT a FROM
Account").getResultList();
  }
  ...
}
```

Alternatively, you can use an EntityManagerFactory directly by either relying on Spring's ability to perform @PersistenceContext injection with a transactional EntityManager or when the scenario requires an application-managed EntityManager.

In this case, you can bind the persistence unit in JNDI as follows:

Example 5.10. Persistence unit binding in JNDI

```
<web-app version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    metadata-complete="true">

    <persistence-unit-ref>
        <persistence-unit-ref-name>example/emf</persistence-unit-ref-
```

```
name>
        <persistence-unit-name>sportsclubPU</persistence-unit-name>
    </persistence-unit-ref>
</web-app>
```

Following is an example of acquiring the entityManagerFactory from JNDI.

Example 5.11. Spring Bean Representing an Entity Manager Factory

```
<jee:jndi-lookup id="entityManagerFactory" jndi-
name="java:comp/env/example/emf"/>
```

In general, for implementing transaction-aware components, you do not access EntityManagerFactories directly (for example a service that delegates to multiple DAOs that have to be enrolled in the same transaction). This is true even if Spring supports injection into fields annotated with @PersistenceContext when an EntityManagerFactory is provided. Rather, components must access the JNDI-bound EntityManager, as it is JTA-synchronized and is also shared with non-Spring components that use JPA (for example EJBs).

A comprehensive example of using JPA-driven data access in Spring-based applications is found in the Sportsclub example application, included in the JBoss Web Framework Kit distribution.

Report a bug

5.3.2. PersistenceUnit Created by Spring

When declaring transactions, Spring applications can specify that a transaction is intended to be read-only as shown in the following example.

Example 5.12. A Spring Method Declaring a Read-Only Transaction

```
@Transaction(readOnly = true)
public List<Account> getAllAccounts() {
    return entityManager.createQuery("SELECT a FROM
Account").getResultList();
}
```

A read-only transaction means that the persistence context is discarded at the end of the transaction. Spring normally tries to block the persistence context from being discarded, but the **`@Transactional(readOnly=true)`** flag causes Spring to allow it. Application performance increases as the persistence context is discarded at the end of the transaction. Support for this mode is not obligatory in Spring, and applications are not expected to rely on this behaviour at all times.

This behaviour is supported by Spring-defined EntityManagerFactories for certain JPA providers (Hibernate is one of them), but not when the EntityManager or EntityManagerFactory is provided from JNDI. This means that a JTA-integrated LocalContainerEntityManagerFactoryBean must be used for any application that requires support for Read-Only persistence contexts. The following is a bean definition for Red Hat JBoss Enterprise Application Platform:

Example 5.13. A Spring-defined JTA-based Entity Manager Factory

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an">
    <property name="jpaVendorAdapter">
        <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.transaction.jta.platform">

org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform
            </prop>
         </props>
    </property>
</bean>
```

This bean definition requires the **`META-INF/persistence.xml`** file, which contains the persistence unit definition, to be present in the deployment. JBoss Enterprise Application Platform automatically deploys the persistence unit from the **`persistence.xml`** file. However, if you require to create a Spring-based EntityManager rather than use the container-managed EntityManagerFactory, prevent JBoss Enterprise Application Platform from deploying the persistence unit. Use one of the following methods to achieve this:

- Configuring JBoss Enterprise Application Platform to not create EntityManagerFactory
- Renaming or moving the **`persistence.xml`** file

Configuring JBoss Enterprise Application Platform to not create EntityManagerFactory

In the **`persistence.xml`** file, set the value of **`jboss.as.jpa.managed`** property to false. This stops JBoss Enterprise Application Platform from creating the EntityManagerFactory. By default, this property is set to true, which enables container managed JPA access to the persistence unit.

Example 5.14. Configuring JBoss Enterprise Application Platform to not create EntityManagerFactory

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="bookingDatabase">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>org.springframework.webflow.samples.booking.User</class>

<class>org.springframework.webflow.samples.booking.Booking</class>
        <class>org.springframework.webflow.samples.booking.Hotel</class>
        <properties>
            <property name="jboss.as.jpa.managed" value="false"/>
```

```
            <!-- other properties (omitted) -->
        </properties>
    </persistence-unit>
</persistence>
```

Renaming or moving the Persistence.xml file

Rename or move the **`persistence.xml`** file and use the **`persistenceXmlLocation`** property as follows:

Example 5.15. Renaming or moving the Persistence.xml file

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an">
   <property name="persistenceXmlLocation" value="classpath*:META-
INF/persistence-booking.xml"/>
 </bean>
```

Irrespective of where the **`persistence.xml`** file is located or what it is called, the WEB-INF/classes location is not scanned for persistent entries when the LocalContainerManagedEntityFactoryBean is used. In this case, the list of entities must be provided explicitly through one of the following methods:

- By enumerating the persistent classes in the **`persistence.xml`** file (or other persistence unit configuration file name)
- By packaging the entities in a JAR file and providing a <jar-file/> configuration entry in the **`persistence.xml`** file
- Using a PersistenceUnitPostprocessor

The first two solutions are based on the standard functionality of JPA. The third solution is Spring-specific and involves implementing a PersistenceUnitPostprocessor class that adds the persistent classes directly to the PersistenceUnitInfo object, as in the following example:

Example 5.16. Example of a PersistenceUnitPostProcessor Implementation That Adds all Classes Annotated with @Entity

```
package org.springframework.webflow.samples.booking;

import java.io.IOException;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.ResourcePatternResolver;
import
org.springframework.core.type.classreading.CachingMetadataReaderFactory
;
import org.springframework.core.type.classreading.MetadataReader;
import
org.springframework.orm.jpa.persistenceunit.MutablePersistenceUnitInfo;
import
```

```
org.springframework.orm.jpa.persistenceunit.PersistenceUnitPostProcesso
r;

public class MyPersistenceUnitPostProcessor implements
PersistenceUnitPostProcessor
{

   @Autowired
   private ResourcePatternResolver resourceLoader;

   public void
postProcessPersistenceUnitInfo(MutablePersistenceUnitInfo
mutablePersistenceUnitInfo)
   {
      try
      {
         Resource[] resources =
resourceLoader.getResources("classpath:org/myexample/*.class");
         for (Resource resource : resources)
         {
            CachingMetadataReaderFactory cachingMetadataReaderFactory
= new CachingMetadataReaderFactory();
            MetadataReader metadataReader =
cachingMetadataReaderFactory.getMetadataReader(resource);
            if
(metadataReader.getAnnotationMetadata().isAnnotated(javax.persistence.
Entity.class.getName()))
            {

mutablePersistenceUnitInfo.addManagedClassName(metadataReader.getClassM
etadata().getClassName());
            }
         }
         mutablePersistenceUnitInfo.setExcludeUnlistedClasses(true);
      }
      catch (IOException e)
      {
         throw new RuntimeException(e);
      }
   }
}
```

You can inject a bean of PersistenceUnitPostprocessor class in the LocalContainerEntityManagerFactoryBean as follows:

Example 5.17. Adding the PersistenceUnitPostProcessor to the Context Definition

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an">
<!-- other properties -->
  <property name="persistenceUnitPostProcessors">
    <list>
      <bean
```

```
class="org.springframework.webflow.samples.booking.MyPersistenceUnitPost
Processor"/>
    </list>
  </property>
</bean>
```

Report a bug

5.3.3. PersistenceContext and PersistenceUnit Injection

Spring injects @PersistenceContext into Spring components by default. In order to do so, applications must have access to an EntityManagerFactory bean (either created by Spring or looked up in JNDI).

You can rename the **`persistence.xml`** file to use a Spring-based EntityManagerFactory. However, the presence of @PersistenceContext or @PersistenceUnit annotations in Spring can cause deployment errors when the **`persistence.xml`** file is renamed. The errors occur because in a Java EE 6 environment, the @PersistenceContext and @PersistenceUnit annotations are used for supporting the container-driven injection of container-managed persistence contexts and persistence units, respectively. During deployment, Red Hat JBoss Enterprise Application Platform scans the deployment classes and validates for the presence of such annotations, the corresponding managed persistence units exist as well. This is not the case if the **`persistence.xml`** file is renamed.

To resolve this, either disable the scanning of deployment classes, or use the @Autowire injection.

In case of Spring-based deployments, the injection of components and resources is done by Spring and not by JBoss Enterprise Application Platform, so in most cases it is not necessary for JBoss Enterprise Application Platform to perform the scanning. For web applications conforming to the Servlet 2.5 specification and higher, this can be set up through the metadata-complete attribute (applications that use the Servlet 2.4 standard do not exhibit this problem, as annotation-based injection is not supported by the container).

Example 5.18. Metadata-complete Flag to Disable Class Scanning by Red Hat JBoss Enterprise Application Platform

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    metadata-complete="true">
```

An alternative is to use @Autowired instead of @PersistenceUnit/@PersistenceContext. In the case of @PersistenceUnit/EntityManagerFactory injection you are required to only replace the annotation, but @PersistenceContext requires an extra step.

If you are using a container-deployed EntityManagerFactory, bind the EntityManager in JNDI and use a JNDI lookup bean to retrieve it.

If you are using a Spring-deployed EntityManagerFactory, autowire a transaction-aware EntityManager by adding a SharedEntityManagerBean definition as in the following example.

Example 5.19. SharedEntityManager Bean to Create an Injectable Transaction-aware EntityManager

```
<bean id="entityManagerWrapper"
class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
   <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

Report a bug

5.4. Transaction Management

Spring provides a declarative transaction model including transaction propagation semantics, to allow applications to declare transaction boundaries around specific methods either through annotation or through XML. The model is the same whether the transactions being used are local transactions or JTA transactions. Spring wraps components in a transactional proxy which delegates to a transaction manager; which is declared separately as a Spring bean. Through PlatformTransactionManager abstraction, Spring gives you a choice between using resource-local transactions and delegating to the transaction manager provided by the application server.

Transaction support in JBoss Enterprise Application Platform is provided by JBoss Transaction Service, a highly configurable transaction manager. To use the JBoss Transaction Service in a Spring application, use a Spring JTA transaction manager definition.

Example 5.20. JTA Transaction Manager Definition in Spring

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName"
value="java:jboss/TransactionManager"/>
</bean>
```

The use of the transaction manager allows Spring to create JTA transactions (for example, through Spring declarative transaction management), or to enroll in existing transactions, if any. This requires the use of managed datasources, JTA-integrated session factories, or container-deployed persistence units. It ensures that the underlying database connections, sessions, and persistence contexts are managed transparently and shared with other components.

Report a bug

5.5. Messaging (JMS) Integration

Spring applications have two distinct mechanisms of integrating with JMS. One is by using the JmsTemplate which simplifies the usage of the JMS API for sending and receiving messages, and the other is by implementing message-driven POJOs.

The recommended practice in Red Hat JBoss Enterprise Application Platform is to use the JmsTemplate for sending messages, but not for receiving them.

Spring applications can implement message-driven POJOs by wiring MessageListener beans (or POJOs through a MessageListenerAdapter) into a MessageListenerContainer.

A typical message-driven POJO is shown in the following example.

Example 5.21. A message-driven POJO as a simple Java class with a single-argument method

```
public class MessageDrivenPojo
{
@Autowire PaymentProcessor paymentProcessor;

public void pojoHandlerMethod(PaymentNotification paymentNotification)
{

paymentProcessor.processPayment(paymentNotification.getAccountNumber(),
paymentNotification.getAmount());
}
}
```

Spring provides two different types of MessageListenerContainers; native JMS and JCA-based. JBoss Enterprise Application Platform uses the JCA MessageListenerContainer for better integration with the application server for server message and delivery session, connection, and message consumer management. In order to minimize the amount of proprietary code, you can use Snowdrop's namespace support for JMS/JCA integration.

Example 5.22. JCA Message Listener Containers and Namespace Support in Red Hat JBoss Enterprise Application Platform

```
<jms:jca-listener-container resource-adapter="resourceAdapter"
acknowledge="auto"
activation-spec-factory="activationSpecFactory">
  <jms:listener destination="/someDestination" ref="messageDrivenPojo"
  method="pojoHandlerMethod"/>
</jms:jca-listener-container>

<jboss:activation-spec-factory id="activationSpecFactory"/>
<jboss:resource-adapter id="resourceAdapter"/>
<bean id="messageDrivenPojo" class="example.MessageDrivenPojo"/>
```

When a message arrives, the container invokes the messageDrivenPojo bean. The message is converted to the argument type of the pojoHandlerMethod inside the bean. You can use any regular bean as a message-driven POJO; the only restriction is that the handling method must have a single argument. Spring handles the conversion of message content to the expected argument type. For a JCA-based container, the invocation is automatically enrolled in a JTA transaction.

Report a bug

5.6. JTA and Messaging Integration

The recommended integration method for receiving JMS messages is a JCA-endpoint based MessageListenerContainer. The processing of a message (during the call to the message handling method) is wrapped in a JTA transaction. As a result, other JTA-aware resources (datasources, entity manager factories, session factories, JMS sessions) participate in the same transaction.

Spring's JMS utility classes such as JmsTemplate and DefaultMessageListenerContainer support injection with a Spring transaction manager abstraction. By injecting the classes with the JTA-based implementation you ensure that the JMS-based operations are enrolled in JTA transactions as well.

Report a bug

5.7. EJB Integration

Although there is overlap between EJB and Spring, combining the two component models is common. It consists of components of one type delegating functionality to components of another type. The recommended practice is to provide the delegate components through injection, which can happen in any direction (Spring to EJB and EJB to Spring).

Report a bug

5.7.1. Injecting Spring Beans into EJBs

Red Hat JBoss Enterprise Application Platform 6 provides the following two major options of injecting Spring beans into EJBs:

- Using Spring's native support for EJB integration
- Using Snowdrop with Spring

Report a bug

5.7.1.1. Using Spring's Native Support for EJB Integration

Spring supports injection into EJBs through SpringBeanAutowiringInterceptor, which honors the @Autowired annotation.

Example 5.23. Injecting Spring Bean into EJB Using Spring's Native Support

```
@Stateless
@Interceptors(SpringBeanAutowiringInterceptor.class)
public class InjectedEjbImpl implements InjectedEjb {

    @Autowired
    private SpringBean springBean;

}
```

The injected Spring beans are retrieved from an ApplicationContext located using a ContextSingletonBeanFactoryLocator, which uses a two-step method for locating contexts. The locator relies on the existence of one or more files named **`beanRefContext.xml`** on the classpath (that is, it performs a 'classpath*:beanRefContext.xml' lookup), which contains a single application context definition.

Example 5.24. Simple beanRefContext.xml File Used by a ContextSingletonBeanFactoryLocator and the Corresponding simpleContext.xml

```
<beans>
    <bean
class="org.springframework.context.support.ClassPathXmlApplicationConte
xt">
        <constructor-arg value="classpath*:simpleContext.xml" />
    </bean>
</beans
```

```
<beans>
  <bean id="springBean" class="example.SpringBean"/>
</beans>
```

Report a bug

5.7.1.2. Using Snowdrop with Spring

Snowdrop is a package of Red Hat JBoss Enterprise Application Platform specific extensions to Spring, which are included in JBoss Web Framework Kit. This section provides an overview of injecting Spring beans into EJBs using Snowdrop support. For details and a more elaborate example, refer the *Snowdrop User Guide* and the *Sportsclub Example*.

Snowdrop supports the bootstrapping of Spring application contexts through JBoss Enterprise Application Platform specific Spring deployer. The deployer identifies Spring bean configuration files, (regular application context XML definitions) which are deployed in the META-INF directory of a deployable module (EAR, WAR, or EJB-JAR), and matches a specific pattern (by default, *-spring.xml). It bootstraps ApplicationContexts, which are further registered in JNDI under a name that can be configured from within the context definition.

Example 5.25. Spring Beans Configuration File (example-spring.xml)

```
<beans>
  <description>BeanFactory=(MyApp)</description>
  <bean id="springBean" class="example.SpringBean"/>
</beans>
```

The Spring deployer bootstraps a context that contains a springBean bean and registers it in JNDI under the *MyApp* name.

You can inject beans defined in such contexts into EJBs, by using the Snowdrop specific SpringLifecycleInterceptor and @Spring annotation.

Example 5.26. Injecting Spring Bean into EJB Using Snowdrop

```
@Stateless
@Interceptors(SpringLifecycleInterceptor.class)
public class InjectedEjbImpl implements InjectedEjb
{
```

```
 @Spring(bean = "springBean", jndiName = "MyApp")
 private SpringBean springBean;

 /* rest of the class definition omitted */
}
```

Report a bug

5.7.2. Accessing EJBs from Spring Beans

You can inject stateless EJBs into Spring components in the following two ways:

- Using EJB reference bean definitions
- Using the @EJB annotation in Spring

You can define EJB references as Spring beans using the **<jee:local-slsb>/<jee:remote-slsb>** elements.

Example 5.27. Defining EJB Reference as Spring Bean

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee.xsd">

  <jee:local-slsb id="ejbReference"
  jndi-name="java:global/example/example-ejb/ExampleImpl"
  business-interface="example.ExampleEjb"/>

  <bean id="consumerBean" class="example.ConsumerBean">
      <property name="ejbReference" ref="ejbReference"/>
  </bean>

</beans>
```

The EJB references are regular Spring beans and can be injected as any other Spring bean.

Example 5.28. Injecting EJB Reference Defined as Spring Bean

```
public class ConsumerBean {

  @Autowire ExampleEjb exampleEJB;

}
```

Spring also supports injection directly with EJB references, as shown in the following example:

Example 5.29. Injecting an EJB Using Annotations

```
public class ConsumerBean {

  @EJB(mappedName="java:global/example/example-ejb/ExampleImpl")
  ExampleEjb exampleEJB;

}
```

Unlike @EJB references in EJBs, which may not specify a name or mappedName and relies upon the container to resolve the reference automatically, @EJB references in Spring beans must specify the JNDI location where the EJB is expected to be found.

Report a bug

5.8. Exceptions and Workarounds

This section describes compatibility exceptions that occur while using Spring, and solutions to work around the exceptions.

Report a bug

5.8.1. Compatibility of Spring SAAJ, Oracle SAAJ, and IBM JDK

The Oracle SOAP with Attachments API for Java (SAAJ) implementation uses internal Java Developer Kit (JDK) classes that are not provided with IBM JDKs. This causes compilation errors in applications using Spring SAAJ support together with the Oracle SAAJ implementation and an IBM JDK.

The recommended workaround is to remove Oracle SAAJ JARs from the class path of the application so that the bundled IBM implementation is used. The following is an example of the SAAJ JAR dependency that must be removed:

```
<dependency>
 <groupId>com.sun.xml.messaging.saaj</groupId>
 <artifactId>saaj-impl</artifactId>
 <version>1.3.10</version>
</dependency>
```

A second workaround, which may not be functional on IBM JVM 7, is to add the Oracle JAXP implementation to the class path of the application as follows:

```
<dependency>
 <groupId>com.sun.xml.parsers</groupId>
 <artifactId>jaxp-ri</artifactId>
 <version>1.4.5</version>
</dependency>
```

Report a bug

Chapter 6. Spring in Action

6.1. About Spring Quickstarts

A quickstart is a sample application that serves as a working example of the features and functionalities provided by the framework. Multiple Spring quickstarts are packaged in the **`Quickstarts`** directory of every Red Hat JBoss Web Framework Kit release, for example:

- spring-greeter
- spring-kitchensink-basic
- spring-petclinic
- spring-resteasy

Each quickstart contains a **`README.md`** file that details the prerequisites of the sample application, build and deploy instructions, and other related information. This chapter discusses the Spring Archetype example that uses the spring-kitchensink quickstart packaged in the JBoss Web Framework Kit distribution.

Report a bug

6.2. About the Spring Archetype example

The Spring Archetype example allows you to create a Spring 4.1 compliant application using Spring MVC, JPA 2.0, and Bean Validation 1.0. It includes a persistence unit, and some sample persistence and transaction code to introduce you to database access in enterprise Java. The Spring Archetype example demonstrates the following:

- The general project structure of a Spring based web application
- A simple set of domain classes, MVC controllers, and web pages
- The data infrastructure configuration (including database, JPA, and transactions) for running Spring applications on JBoss Enterprise Application Platform

The following tasks are explored in the Spring Archetype example:

- Creating a project
- Adding source code
- Adding Spring configuration files
- Running the example

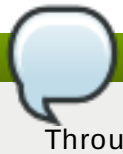
The following section demonstrates the Spring Archetype example using Red Hat JBoss Developer Studio to develop a simple Spring Model-View-Controller (MVC) web application on Red Hat JBoss Enterprise Application Platform.

Report a bug

6.3. Prerequisites of the Spring Archetype example

The Spring Archetype example uses the following frameworks and tools:

- Maven 3.0.4
- Spring 4.1.4.RELEASE
- Red Hat JBoss Developer Studio 8
- Red Hat JBoss Enterprise Application Platform 6



Note

Throughout the guide, *$EAP_HOME* represents the top-level directory of the extracted distribution, which is usually **`jboss-eap-6`**.

Report a bug

6.4. Creating the Spring Archetype Project

Use the following procedure to create an archetype Spring project using Red Hat JBoss Developer Studio.

Procedure 6.1. Creating a Maven Project

1. Start JBoss Developer Studio.
2. Ensure you are using the **`JBoss`** perspective.
3. Click **File** → **New** → **Spring MVC Project**.
4. Enter the **`Target Runtime`** and click **`Next`**.
5. Enter the **`Project name`** and **`Package`**.

   Project name

   jboss-springmvc-webapp (that is the default)

   Package

   org.jboss.tools.example.springmvc (that is the default)

   Use default Workspace location

   (leave checked)
6. Click **`Next`**. Confirm the **`Group Id`**, **`Artifact Id`**, **`Version`**, and **`Package`**.

   Group Id

   Automatically set to org.jboss.tools.example.springmvc

   Artifact Id

   jboss-springmvc-webapp

   Version

   0.0.1-SNAPSHOT

Package

Automatically set to org.jboss.tools.example.springmvc

7. Click **Finish**.

Result

Your project is created.

Spring Project Nature

Adding the Spring Project nature is not a requirement, but it enables integration with Spring IDE. This includes visualizing your application context definition, and syntactical and semantical checks.

Procedure 6.2. Adding tool support for Spring

1. In the **JBoss Central** view, select the **Software/Update** tab.
2. In the **Features Available** list, select the **Spring IDE** plug-in.
3. Click **Install**.

Report a bug

6.5. Running the Spring Archetype Project

Use the following procedures to run the Spring Archetype project from the command line interface and Red Hat JBoss Developer Studio respectively.

Procedure 6.3. Run the Example from the Command Line

1. In the example directory, build the example with Maven.

```
cd $EXAMPLE_HOME
mvn clean package
```

2. Copy the example into your JBoss Enterprise Application Platform instance.

```
cp target/jboss-springmvc-webapp.war
$EAP_HOME/standalone/deployments
```

3. Run the JBoss Enterprise Application Platform instance.

```
$EAP_HOME/bin/standalone.sh
```

4. After you have deployed the example, you can access it at http://localhost:8080/jboss-springmvc-webapp.

Procedure 6.4. Run the example from JBoss Developer Studio

1. In the **Project Explorer** view, right-click the Spring Archetype project and click **Run As→Run on Server**. The **Run on Server** window opens.
2. From the **Select the server type** list, select a JBoss Enterprise Application Platform 6 instance.

3. Click **Finish**.
4. After the example is deployed, you can accessed it at http://localhost:8080/jboss-springmvc-webapp.

Report a bug

Chapter 7. Migrating Spring Applications

7.1. Migrating Spring Archives to Red Hat JBoss Enterprise Application Platform 6 from a Previous Version

Overview

This section covers the main points you must considered when you plan to migrate existing Spring applications from JBoss Enterprise Application Platform 5 to JBoss Enterprise Application Platform 6. The changes required depend on the strategy and features used by the application.

Important

This example is intended to help you get your application running on JBoss Enterprise Application Platform 6 as a first step. Be aware that this configuration is not supported, so upgrade your application to use Hibernate 4 as your next migration step.

Update Spring applications that use Hibernate

Applications that use the Hibernate API directly with Spring, through either LocalSessionFactoryBean or AnnotationSessionFactoryBean, may use a version of Hibernate 3 packaged inside the application. Hibernate 4, which is provided through the org.hibernate module of JBoss Enterprise Application Platform 6, is not supported by Spring 3.0, but it is supported by Spring 3.1 or higher.

Update Spring applications that use JPA

The changes required in the application depend on whether the Spring JPA based application uses a server-deployed persistence unit or a Spring-managed persistence unit.

Report a bug

7.1.1. Server-deployed Persistence Unit

Applications that use a server-deployed persistence unit must observe the typical Java EE rules with respect to dependency management. The **`javax.persistence`** classes and the persistence provider, in this case Hibernate, are contained in modules which are added automatically by the application when the persistence unit is deployed. To use the server-deployed persistence units from within Spring, do one of the following:

- Register the JNDI persistence context in the **`web.xml`** file:

```
<persistence-context-ref>
    <persistence-context-ref-name>persistence/petclinic-
em</persistence-unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-context-ref>
```

The persistence context is then available for JNDI lookup as follows:

```
<jee:jndi-lookup id="entityManager" jndi-
name="java:comp/env/persistence/petclinic-em" expected-
type="javax.persistence.EntityManager"/>
```

- Register the JNDI persistence unit in the **web.xml** file:

```
<persistence-unit-ref>
    <persistence-unit-ref-name>persistence/petclinic-emf</persistence-
unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
<persistence-unit-ref>
    <persistence-unit-ref-name>persistence/petclinic-emf</persistence-
unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
```

The persistence unit is then available for JNDI lookup as follows:

```
<jee:jndi-lookup id="entityManagerFactory" jndi-
name="java:comp/env/persistence/petclinic-emf" expected-
type="javax.persistence.EntityManagerFactory"/>
```

Note

JNDI binding using the **persistence.xml** file properties is not supported in JBoss Enterprise Application Platform 6.

Report a bug

7.1.2. Spring-managed Persistence Unit

Spring applications running on JBoss Enterprise Application Platform 6 may also create persistence units using the **LocalContainerEntityManagerFactoryBean** class. For such applications consider the following:

- **Change the location of persistence unit definitions**

  When a deployment contains a **META-INF/persistence.xml** file or a **WEB-INF/classes/META-INF/persistence.xml** file, the application server attempts to create a persistence unit based on the data in that file. In most cases, these definition files are not compliant with the Java EE requirements because the required elements (for example the datasource of the persistence unit) are expected to be provided by the Spring context definitions. This results in failure of deployment of the persistence unit and consequently of the entire deployment.

  You can avoid this problem by using a feature of the **LocalContainerEntityManagerFactoryBean** that is designed for this purpose. Persistence unit definition files can exist in other locations than the **META-INF/persistence.xml** file and

the location can be indicated through the **persistenceXmlLocation** property of the factory bean class.

The following example assumes the persistence unit is in the **META-INF/jpa-persistence.xml** file:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryB
ean">
    <property name="persistenceXmlLocation" value="classpath*:META-
INF/jpa-persistence.xml"/>
    <!-- other definitions -->
</bean>
```

- **Manage dependencies**

  If using Hibernate with the application, the Hibernate 3 JARs must be included in the deployment. However, due to the presence of @PersistenceUnit or @PersistenceContext annotations on the application classes, the application server automatically adds the Hibernate 4 **org.hibernate** module as a dependency. You can instruct the server to exclude this module by creating a **META-INF/jboss-deployment-structure.xml** file (or a **WEB-INF/jboss-deployment-structure.xml** file for web applications) with the following content:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
 <deployment>
  <exclusions>
   <module name="org.hibernate"/>
  </exclusions>
 </deployment>
</jboss-deployment-structure>
```

Report a bug

7.2. Migrating Spring to Red Hat JBoss Enterprise Application Platform from a Servlet Container

This section explains how Spring applications developed for servlet containers can be migrated to take advantage of the Red Hat JBoss middleware service support and infrastructure. This is explained by using sample applications provided by Spring.

Migrating applications from a servlet container to JBoss Enterprise Application Platform addresses the following concerns:

- Ensures that applications are compatible with the requirements of Java EE 6 standard
- Enables integration with the Red Hat JBoss middleware services provided by the container
- Avoids the inclusion of libraries that are provided by the application server

Except for rare situations, migrating applications does not involve changes to the Java code or the Spring bean configuration definitions that define the business logic of the application (bean wiring, aspect definitions, and controllers). The definitions that relate to the infrastructure are:

- Session factories

- Datasources
- Entity managers

Report a bug

7.2.1. Avoiding the Inclusion of Server-provided Dependencies

Red Hat JBoss Enterprise Application Platform already provides a number of dependencies that are required by applications. This is unlike servlet containers, where applications need to package a significant number of libraries in order to provide access to certain Java EE 6 technologies. During migration, change the deployable build by removing the libraries provided by the application server. This is not only an improvement that reduces the size of the final build, but also a requirement, as the inclusion of those libraries results in classloading errors.

Some examples of libraries provided by JBoss Enterprise Application Platform and that do not need to be included in the application are:

- The Java EE 6 APIs (like JPA, JSTL, and JMS)
- Hibernate (including Hibernate as a JPA provider)
- JSF

These components may be required as compile-time dependencies, so include them in the project, but do not package them in the final build. In Maven builds, you can achieve this by setting the scope to *provided* and in Red Hat JBoss Developer Studio by unchecking the **Exported** flag for the dependency. Other build systems have their own respective mechanisms of achieving the same goal.

However, components and frameworks that are not part of the Java EE 6 implementation but are certified for use with JBoss Enterprise Application Platform 6 must be included in the application or installed in the application server as subsystems. These components include:

- Spring Framework
- Snowdrop
- Facelets
- RichFaces

Report a bug

7.2.2. Migrating Datasource Definitions

Most servlet containers support JNDI-bound resources, and binding Datasources in JNDI allows managing connectivity parameters such as URLs, credentials, and pool sizes independent of the application code. However, Spring applications often rely on independent connection pool bean definitions, like those used by commons-dbcp or c3po (see Example 7.1, “Example commons-dbcp DataSource definition in a servlet container”).

Example 7.1. Example commons-dbcp DataSource definition in a servlet container

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
 <property name="driverClassName" value="org.postgresql.Driver" />
 <property name="url"
```

```
value="jdbc:postgresql://exampleHost/exampleDatabase" />
 <property name="username" value="user" />
 <property name="password" value="password" />
</bean>
```

Red Hat JBoss Enterprise Application Platform provides an efficient way of deploying and maintaining managed datasources (see Example 7.2, “Using a JBoss Enterprise Application Platform managed datasource in Spring” for more details, and *JBoss Enterprise Application Platform Administration and Configuration Guide* for information on deploying them) that are JNDI-accessible. In order to replace the datasource defined in the commons-dbcp example with a managed JNDI-accessible datasource, replace the bean definition with a JNDI object reference as in the following example.

Example 7.2. Using a JBoss Enterprise Application Platform managed datasource in Spring

```
<jee:jndi-lookup id="dataSource" jndi-
name="java:jboss/datasources/ExampleDS" expected-
type="javax.sql.DataSource"/>
```

Preserving the bean id is important when the bean is injected by name. Indicating the expected type specifically is important for @Autowired scenarios.

Report a bug

7.2.3. Migrating Hibernate SessionFactories to JTA

To run applications on JBoss Enterprise Application Platform use JTA for transaction management. The infrastructure changes consist of altering the session factory definition to allow JTA-backed session context and replacing the local Spring-based transaction manager with a HibernateTransactionManager.

Listing Example 7.3, “SessionFactory and transaction manager definitions in a servlet environment” contains typical bean definitions for SessionFactory and transaction manager when used in a servlet container with local Hibernate-based transactions enabled.

Example 7.3. SessionFactory and transaction manager definitions in a servlet environment

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="mappingLocations" value="classpath:**/*.hbm.xml"/>
 <property name="hibernateProperties">
  <value>
   hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
   hibernate.show_sql=true
  </value>
 </property>
</bean>
```

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate4.HibernateTransactionManager">
 <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

You can migrate these definitions to use JTA (the datasource is assumed to be a managed datasource, as shown previously).

Example 7.4. JTA-based SessionFactory and transaction manager

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="mappingLocations" value="classpath:**/*.hbm.xml"/>
 <property name="hibernateProperties">
  <value>
   hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
   hibernate.show_sql=true
   hibernate.current_session_context_class=jta

hibernate.transaction.jta.platform=org.hibernate.service.jta.platform.i
nternal.JBossAppServerJtaPlatform

hibernate.transaction.factory_class=org.hibernate.transaction.JTATransa
ctionFactory
  </value>
 </property>
</bean>

<tx:jta-transaction-manager id="transactionManager"/>
```

Note

Definitions of the sessionFactory bean are virtually unchanged; the only difference is the addition of the properties required for setting up JTA-based context management and the transaction manager change.

Report a bug

7.2.4. Migrating JPA-based Applications

Java EE 6 environments have restrictions regarding what is deployable by a container that fully supports the JPA API. Specifically, a persistence unit definition contained in a **`META-INF/persistence.xml`** file is automatically deployed by the container, but the container cannot declare a RESOURCE_LOCAL transaction type, and must include a JTA datasource reference. Also, the container may not specify a transaction type (which is equivalent to setting a JTA transaction type, but is ignored when the persistence unit is initialized by Spring and a resource-local model is used instead).

However, it is common for servlet-container based applications to use RESOURCE_LOCAL transactions and JTA datasources as shown in example Example 7.5, “A sample persistence unit definition for a servlet-container based application”.

Example 7.5. A sample persistence unit definition for a servlet-container based application

```
<persistence-unit name="examplePU" transaction-type="RESOURCE_LOCAL">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <properties>
   <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
   <property name="hibernate.show_sql" value="true"/>
   <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider"/>
 </properties>
</persistence-unit>
```

A servlet container does not deploy a persistence unit automatically, as a Java EE 6 application does. Therefore, Spring applications using JPA rely on Spring to create the persistence unit by using one of its JPA support factory beans. Also, in a Java EE 6 application, server enrollment in JTA transactions is a requirement, Spring applications running outside Java EE are required to set up a resource-local transaction manager as described in Example 7.6, “JPA EntityManagerFactory and transaction setup in a servlet-container based application”.

Example 7.6. JPA EntityManagerFactory and transaction setup in a servlet-container based application

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an"
p:dataSource-ref="dataSource">
 <property name="jpaVendorAdapter">
  <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
   p:database="${jpa.database}" p:showSql="${jpa.showSql}"/>
 </property>
</bean>

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager"
p:entityManagerFactory-ref="entityManagerFactory"/>
```

This persistence unit definition in Example 7.6, “JPA EntityManagerFactory and transaction setup in a servlet-container based application” is not compatible with the requirements of Java EE 6 for application containers. It leads to deployment failure if found in the **`META-INF/persistence.xml`** file.

There are two ways to solve this problem and to make the application deployable on Red Hat JBoss Enterprise Application Platform:

- rename the persistence unit definition file

- leave persistence-unit deployment to JBoss Enterprise Application Platform and use JNDI lookup for retrieving entity managers and entity manager factories.

To rename the persistence unit, you can provide the alternate location as a property to the LocalContainerEntityManagerFactoryBean as described in Example 7.7, “LocalContainerEntityManagerFactoryBean with alternate persistence.xml location”.

Example 7.7. LocalContainerEntityManagerFactoryBean with alternate persistence.xml location

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an">
 <property name="persistenceXmlLocation" value="classpath*:META-
INF/jpa-persistence.xml"/>
  <!-- other properties (ommitted) -->
</bean>
```

A more effective approach is converting the **`persistence.xml`** file definition to a JTA-based model and using JNDI lookup for retrieving the entity manager. For this, convert the persistence unit definition as in the following example (note that it is not necessary to provide values for both *jboss.entity.manager.jndi.name* and *jboss.entity.manager.factory.jndi.name*, but one must be specified).

Example 7.8. Changing the persistence.xml definition to be Java EE 6 compatible

```
<persistence-unit name="examplePU">
 <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <properties>
   <property name="hibernate.show_sql" value="true"/>
   <property name="jboss.entity.manager.jndi.name"
value="java:jboss/example/EntityManager"/>
   <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/example/EntityManagerFactory"/>
  </properties>
</persistence-unit>
```

You can retrieve the EntityManagerFactory or JTA-synchronized EntityManager from JNDI as follows:

Example 7.9. EntityManager retrieved by JNDI lookup and JTA transaction manager (works with @Autowired)

```
<jee:jndi-lookup id="entityManager" jndi-
name="java:jboss/example/EntityManager" expected-
type="javax.persistence.EntityManager"/>

<tx:jta-transaction-manager/>
```

If the original application relied on Spring to inject the EntityManager using the @PersistenceContext annotation into services and DAOs, change the annotation to @Autowired (as the bean is an EntityManager). In such cases, look for the EntityManagerFactory instead, and allow Spring to create the EntityManager.

Example 7.10. EntityManagerFactory retrieved by JNDI lookup and JTA transaction manager (works with @PersistenceContext)

```
<jee:jndi-lookup id="entityManagerFactory" jndi-
name="java:jboss/example/EntityManagerFactory" expected-
type="javax.persistence.EntityManagerFactory"/>

<tx:jta-transaction-manager/>
```

A particular concern when migrating JPA-based application is the choice of a JPA provider. Servlet-container-based applications include a JPA implementation and specify the nature of the provider explicitly, whereas, JBoss Enterprise Application Platform uses Hibernate as a JPA provider.

Report a bug

7.3. Migrating Spring PetClinic Example

This section explains the migration of Spring PetClinic sample application. Download the application from Spring Framework's github repository at https://github.com/spring-projects/spring-petclinic.

The example provides a number of alternative data access implementations, to use these implementations change the **web.xml** file. This shows the migration strategies in each case, noting that only one of these strategies is used at a time.

To help you with the migration process, a migrated example named **spring-petclinic** is distributed with this release of JBoss Web Framework Kit. It can be obtained from the JBoss Web Framework Kit **quickstarts.zip** file which is available to download from the Red Hat Customer Portal at https://access.redhat.com/ or from the JBoss Developer website at http://www.jboss.org/.

Report a bug

7.3.1. Preliminary Changes in Spring PetClinic Example

Before starting the actual migration process, make a few adjustments to the **pom.xml** file. Spring Petclinic uses Hibernate 4.3.8.Final, which is not supported on Red Hat JBoss Enterprise Application Platform 6.x. So, downgrade the version of Hibernate to a supported version. Change Hibernate version in the **pom.xml** file to 4.2.14.SP4-redhat-1.

Note

You can also use a BOM to reference the supported versions of Hibernate. There is a Hibernate BOM org.jboss.bom.eap:jboss-javaee-6.0-with-hibernate.

Report a bug

7.3.2. Build and Deploy the Spring PetClinic Application

Execute the following steps to build and deploy the Spring Petclinic application downloaded from the github repository:

1. Run an instance of JBoss Enterprise Application Platform.

```
$EAP_HOME/bin/standalone.sh
```

2. Navigate to the root directory of the Spring Petclinic application.

```
cd $spring-petclinic
```

3. Build and deploy the application using Maven.

```
mvn clean package
```

4. Copy **`petclinic.war`** file into the JBoss Enterprise Application Platform instance.

```
cp petclinic-spring/target/petclinic.war
$EAP_HOME/standalone/deployments/
```

After the example has been deployed, you can access it at http://localhost:8080/petclinic/.

Report a bug

Revision History

Revision 2.7.0-3	**Thu Jan 22 2015**	**Michelle Murray**
JBQA-11344: QE feedback incorporated		
Revision 2.7.0-1	**Tues Jan 06 2015**	**Michelle Murray**
Generated for WFK 2.7 release		