Oracle® Fusion Middleware

Developing and Administering Spring Applications for Oracle WebLogic Server

12*c* (12.2.1)

E55176-01

October 2015

This document describes Spring support in WebLogic Server, tells how to enable the Spring extension and the Spring console-extension, and provides information about developing Spring applications for WebLogic Server.

ORACLE

Oracle Fusion Middleware Developing and Administering Spring Applications for Oracle WebLogic Server, 12*c* (12.2.1)

E55176-01

Contents

Preface

This preface describes the document accessibility features and conventions used in this guide—*Developing and Administering Spring Applications for Oracle WebLogic Server*.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction and Roadmap

This chapter describes the contents and organization of this guide - *Developing and Administering Spring Applications for Oracle WebLogic Server*.

WebLogic Server supports the open source Spring projects when they are used in Java EE applications. This document describes the Spring features that WebLogic supports for use inside Java EE applications. WebLogic Server does not support any commercial products by SpringSource.

This chapter includes the following sections:

1.1 Document Scope and Audience

This document is written for developers who develop Spring applications and for administrators who configure and monitor those applications.

It is assumed that the reader is familiar with WebLogic Server and Java EE application development.

1.2 Guide to This Document

- This section, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Simplified Configuration for Spring Applications," discusses the preconfigured MBeans that you can use for dependency injection, with no special configuration required.
- Chapter 3, "WebLogic Spring Security Integration," tells how to use the Spring security framework with the WebLogic Server security framework.
- Chapter 4, "Spring Console Extension in WebLogic Server," tells how to enable the Spring extension to the WebLogic Server Administration Console.
- Chapter 5, "Using WebLogic Server Clustering," tells how to take advantage of WebLogic Server clustering for Spring applications.

- Chapter 6, "Spring Dependency Injection Extension to WebLogic Server," tells how to enable Spring support in WebLogic Server.
- Chapter 7, "Developing Spring-Based Applications for Oracle WebLogic Server," provides examples of how to develop Spring applications for WebLogic Server.

1.3 Related Documentation

For Spring documentation and other information about the Spring Framework, see http://www.springsource.com/.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- *Tuning Performance of Oracle WebLogic Server* contains information on monitoring and improving the performance of WebLogic Server applications.

1.4 Examples for Spring Application Development

In addition to this document, Oracle provides a sample application developed using Spring. It is a Spring version of Avitek Medical Records (or MedRec) and is based on the original MedRec sample application, but it is reimplemented using the Spring framework. It is a Oracle WebLogic Server sample application suite that concisely demonstrates many aspects of Spring and WebLogic.

In this release of WebLogic Server, the Code Examples and MedRec sample applications are not installed by default. To run the examples and sample applications, you must select a custom installation of WebLogic Server and select to install the Server Examples.

1.5 Support for Spring Framework on WebLogic Server

WebLogic Server supports the WebLogic Server/Spring integration features described in this document with Spring Framework versions 3.0.*x*, 3.1.*x*, and 4.0.*x* (note that version 3.2.*x* is not supported).

Other versions of Spring may be used in WebLogic Server applications without using the integration features described in this document, similar to the way other open source technologies may be used in WebLogic Server applications. In such cases, Oracle does not support the WebLogic Server/Spring integration features, but does provide support for WebLogic Server itself, and does provide problem resolution assistance if WebLogic Server is not providing documented capabilities.

1.6 New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server 12.2.1*.

2

Simplified Configuration for Spring Applications

This chapter describes how to configure Spring applications to use WebLogic Server resources.

To use WebLogic Server resources for a Spring application, you must specify a number of configurations for the application. WebLogic Server detects that an application is a Spring application and automatically creates these configurations for optimal performance. When the application context is created on application startup, WebLogic Server automatically creates a parent context containing the predefined resources. By making the application's context a child of this context, all of these resources become available to the application, and no explicit configuration is required.

The Spring configuration file *WL_HOME*`\server\lib\SpringServerApplicationContext.xml` specifies the default settings, which you can review to see if they satisfy your needs. You can override these settings in your own application context, if desired.

The defaults include:

- The default transaction manager is `WebLogicJtaTransactionManager`.
- The WebLogic Server runtime MBean server is automatically configured for Spring JMX
- Server resources defined in WebLogic Server are automatically exposed as Spring beans.

3

WebLogic Spring Security Integration

This chapter describes how to integrate the WebLogic Server security system with the Spring security framework.

The WebLogic Server security system supports and extends Java EE security while providing a rich set of security providers that you can be customize to integrate with different security databases or security policies.

As described at the Spring Security Web site (http://static.springsource.org/spring-security/site/), Acegi Security is now Spring Security, the official security project of the Spring Portfolio. The Spring Security (acegi) framework provides security to a Spring application and includes a rich set of security providers.

The question then becomes how to integrate the two security frameworks.

For a combined Java EE and Spring application, rather than require authentication with both security frameworks, WLS security and Spring security work together. WLS security handles the authentication via the default Authentication provider for the security realm, and converts WLS principals to Spring GrantedAuthority principals through a mapper class. Once authenticated by WLS security, a user is authenticated for Spring security. You can then decide how to secure the objects in the application. One common practice is to secure Java EE resource with WebLogic security and secure Spring resource with Spring security.

3.1 Spring Container Adapter Provides Integration

As described in the Spring Security Reference, Container Adapters enable Spring Security to integrate directly with the containers used to host end user applications, in this case WebLogic Server.

The integration between a container and Spring Security is achieved through an adapter. The adapter provides a container-compatible user authentication provider, and needs to return a container-compatible user object.

`applicationContext-acegi-security.xml` is the configuration file for Spring security. For WebLogic Server, `WeblogicAuthenticationFilter` is added to the list of filters in `applicationContext-acegi-security.xml`. This filter is responsible for converting the WebLogic principals to Spring GrantedAuthority subjects, based on the mapper. The mapper is configured as a property for the `WeblogicAuthenticationFilter`, and it is injected at creation time.

The following is an example of the mapper class.

```
public class MyAuthorityGranter implements AuthorityGranter {
public Set grant(Principal principal) {
Set rtnSet = new HashSet();
```

```
if (principal.getName().equals("fred@oracle.com")) {
rtnSet.add("ROLE_SUPERVISOR");
rtnSet.add("IS_AUTHENTICATED_ANONYMOUSLY");
}
return rtnSet;
}
}
```

In this example, user `fred@oracle.com` in the WebLogic domain is mapped to `ROLE_SUPERVISOR` and `IS_AUTHENTICATED_ANONYMOUSLY`.

3.2 How applicationContext-acegi-security.xml Is Plugged Into web.xml

The following code is added to `web.xml` to plug in the `applicationContext-acegi-security.xml` file:

```
<filter>
  <filter-name>Acegi Filter Chain Proxy</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.util.FilterChainProxy</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>Acegi Filter Chain Proxy</filter-name>
  <url-pattern>/main/secure/*</url-pattern>
  </filter-mapping>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext-acegi-security.xml
  </param-value>
</context-param>
```

4

Spring Console Extension in WebLogic Server

This chapter describes the Spring console extension, which is based on Runtime MBeans registered using the WebLogic Server infrastructure. The console extension displays configuration and runtime information for deployed Spring beans.

> **Limitation:** To enable the Spring console extension, you must have a Web component as part of your application. This is because most of the applications that use Spring are Web applications.

To use the Spring console extension, you must turn on support for Spring beans and enable the Spring console extension, as explained in steps 1 and 2, below. You only have to do this once. In addition, you must enable your Spring applications to make use of the Spring console extension. This requires making simple changes to your Spring application configuration, as explained in step 3, below. You have to do this for each of your Spring applications.

1. Deploy *WL_HOME*`/server/lib/weblogic-spring.jar` to WebLogic Server, where *WL_HOME* refers to the main WebLogic Server installation directory, such as `Oracle\Middleware\Oracle_Home\wlserver`. You only need to perform this step once for your WebLogic Server instance.

   `weblogic-spring.jar` file is a Java EE optional package used by an application (packaged as an EAR or WAR file). It creates the MBeans for your application during its deployment.

   Deploy `weblogic-spring.jar` either of the following ways:

   - Use the WebLogic Server Administration Console, as described in "Deploy applications and modules" in *Oracle WebLogic Server Administration Console Online Help*. The JAR file is located in *WL_HOME*`\server\lib`.
   - Issue the following command at the command line:

     ```
     java weblogic.Deployer -library -deploy - source
        WL_HOME/server/lib/weblogic-spring.jar - targets
        server_name -adminurl server_URL -user
     user_name -password password
     ```

2. Enable the Spring console extension in the WebLogic Server Administration Console. (The Spring console extension is disabled by default.) You only need to perform this step once for your domain. Do the following:

   a. Log into the WebLogic Server Administration Console.

   b. In the banner toolbar region at the top of the right pane of the Console, click **Preferences**.

c. On the Preferences page, click **Extensions**.

d. Select the check box next to **spring-console**, then click **Enable**.

e. Stop the server, then restart it for the change to take effect.

3. Change the manifest of your application (packaged as an EAR or WAR file) so it includes `weblogic-spring.jar` as a Java EE optional package. Do this to each Spring application you want to make use of the Spring runtime MBeans or the Spring console extension

   Do this by adding the following lines to your `META-INF/Manifest.mf`:

```
Extension-List: WeblogicSpring
WeblogicSpring-Extension-Name: weblogic-spring
WeblogicSpring-Specification-Version: 10.3.0.0
WeblogicSpring-Implementation-Version: 10.3.0.0
```

After you deploy your application, you can monitor it in the WebLogic Server Administration Console. The Spring configuration and monitoring pages are under the Spring Framework tab. See "Spring Bean Task Overview" in *Oracle WebLogic Server Administration Console Online Help*

5

Using WebLogic Server Clustering

This chapter describes how Spring applications can take advantage of WebLogic Server's clustering features. Because most Spring applications are packaged as Web applications (`.war` files), you do not need to do anything special in order to take advantage of WebLogic Server clusters. All you need to do is deploy your Spring application to the servers in a WebLogic Server cluster. For information on which Spring versions are supported with this and other WebLogic Server/Spring integration features, see Section 1.5, "Support for Spring Framework on WebLogic Server."

WebLogic Server extends the Spring `JndiRmiProxyFactoryBean` and its associated service exporter so that it supports proxying with any Java EE RMI implementation. To use the extension to the `JndiRmiProxyFactoryBean` and its exporter:

1. Configure client support by implementing code such as the following:

```
<bean id="proProxy"
    class="org.springframework.remoting.rmi.JndiRmiProxyFactoryBean">
  <property name="jndiName" value="t3://${serverName}:${rmiPort}/order"/>
  </property>
  <property name="jndiEnvironment">
    <props>
      <prop key="java.naming.factory.url.pkgs">weblogic.jndi.factories</prop>
    </props>
  </property>
  <property name="serviceInterface"
    value="org.springframework.samples.jpetstore.domain.logic.OrderService"/>
</bean>
```

2. Configure the service exporter by implementing code such as the following:

```
<bean id="order-pro"
    class="org.springframework.remoting.rmi.JndiRmiServiceExporter">
  <property name="service" ref="petStore"/>
  <property name="serviceInterface"
    value="org.springframework.samples.jpetstore.domain.logic.OrderService"/>
  <property name="jndiName" value="order"/>
</bean>
```

6

Spring Dependency Injection Extension to WebLogic Server

This chapter describes how to enable the Spring Framework extension that provides enhanced dependency-injection with WebLogic Server.

A standard WebLogic Server installation provides standard Java EE 7 dependency injection (DI) and interceptors (a form of aspect-oriented programming) in the WebLogic Server Java EE container. WebLogic Server also supports a Spring Framework extension that provides enhanced dependency injection and aspect-oriented programming features in the container. This extension uses Pitchfork, a Spring Framework add-on that provides JSR-250 (Common Annotations), dependency injection, and EJB 3.0 style interception. (See `http://oss.oracle.com/projects/pitchfork/`.) The extension provides dependency injection and aspect-oriented programming to EJB instances and Web components that include the servlet listener and filter.

> **Note:** JSP tag handlers do not support the Spring extension in this release of WebLogic Server.

To enable the Spring extension with WebLogic Server, do the following:

1. Download a version of Spring and its dependencies from `http://www.springsource.com/download`. Download the version of Spring that is certified by Oracle. For that information, see the Oracle Fusion Middleware Supported System Configurations page on the Oracle Technology Network.

   You must have at least the following JAR files:

   - spring.jar
   - aspectjweaver.jar
   - commons-logging.jar
   - log4j-1.2.14.jar
   - pitchfork.jar

   Also download `pitchfork.jar` from `http://oss.oracle.com/projects/pitchfork/`

   You can add other JAR files if necessary.

2. Add the JARs listed above to the WebLogic Server classpath. See Adding JARs to the System Classpath in *Developing Applications for Oracle WebLogic Server*.

The WebLogic Server Web container and EJB container use these JARs to provide container services (dependency injection and interceptor).

Applications (packaged as EAR, WAR, or JAR files) can use these JARs because they are in the server classpath. You can configure your application to use the version of JARs packaged with the application if you enable certain of the deployment descriptors.

3. Enable the Spring extension by setting the `<component-factory-class-name>` element to `org.springframework.jee.interfaces.SpringComponentFactory`. This element exists in EJB, Web, and application descriptors. A module level descriptor overwrites an application level descriptor. If the tag is set to null (default), the Spring extension is disabled.

4. Provide the standard Spring bean definition file with the name `spring-ejb-jar.xml` or `spring-web.xml`, and place it in the /WEB-INF/classes/META-INF directory of your application (or put the META-INF directory in a JAR file). These are the standard Spring bean definition files with the names that the Weblogic container searches for. For the Spring container to be aware of the EJB or servlet instance, the `<id>` tag of the Spring bean must be set to the `ejb-name` for EJB or the `class-name` of the Web components.

   For example, for the following stateless EJB...

```
@Stateless

public class TraderImpl
  implements Trader {
  private List symbolList;
  public void setSymbolList(List l) {
    symbolList = l;
  }

  ...
}
```

   ...you can add the following `spring-ejb-jar.xml` to the application...

```
<beans>
  <!-- id corresponds to ejb-name. -->
  <bean id="Trader">
     <property name="symbolList">
       <list>
         <value>ORCL</value>
         <value>MSFT</value>
       </list>
     </property>
</beans>
```

   ...to inject a `symbolList` to the EJB, which is not available using the standard Java EE specification.

7

Developing Spring-Based Applications for Oracle WebLogic Server

This chapter describes the MedRec-Spring sample application, the Spring version of WebLogic Server's Java EE-based Avitek Medical Records (MedRec) sample application.

Included with WebLogic Server is a sample application, called MedRec (Spring Version), called MedRec-Spring for short. In MedRec-Spring, the Java EE-based Medrec's components are replaced with Spring components, as described in the following sections:

1. Configure Spring Inversion of Control.
2. Enable the Spring Web Services Client Service. Spring offers a JAX-WC factory to produce a proxy for Web Services
3. Make JMS Services Available to the Application at Runtime.
4. Use JPA Data Access.
5. Use the Spring Transaction Abstraction Layer for Transaction Management.

The sample code in the following sections are from MedRec-Spring.

> **Note:** MedRec-Spring is not installed by default when you install WebLogic Server. You must choose Custom installation, then select Server Examples from the Choose Products and Component page.
>
> If you have already installed WebLogic Server, rerun the installer, select the Middleware home where WebLogic Server is installed, choose Custom installation, then select Server Examples from the Choose Products and Component page.
>
> Included with Medrec-Spring is documentation which discusses its design and implementation. That documentation is available at *ORACLE_HOME*\wlserver\samples\server\docs\, where *ORACLE_HOME* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see "Sample Applications and Code Examples" in *Understanding Oracle WebLogic Server*.

7.1 Configure Spring Inversion of Control

In Spring, references to other beans (injected properties) are configured via the Spring configuration file `applicationContext-web.xml`.

Spring 2.5 annotation-driven configuration is used in MedRec-Spring. The application context is configured to have Spring automatically scan the Spring beans detecting Spring-specific annotations like `@Service`, so it is not necessary to declare every Spring bean in the XML configuration files. The configuration, in *ORACLE_HOME*`\wlserver\samples\server\medrec-spring\modules\medrec\web\war\WEB-INF\applicationContext.xml`, is as follows:

```
<context:component-scan base-package="com.oracle.medrec"/>
```

The dependency injection is mainly configured via the `@Autowired` annotation. For example, the *ORACLE_HOME*`\wlserver\samples\server\medrec-spring\modules\medrec\domain\src\com\oracle\medrec\service\impl\RecordServiceImpl.java` includes the following:

```
@Service("recordService")
@Transactional
public class RecordServiceImpl implements RecordService {

    @Autowired
    private RecordRepository recordRepository;

    @Autowired
    private PatientRepository patientRepository;

    @Autowired
    private PhysicianRepository physicianRepository;
```

All of these provide similar development experience as that of EJB 3.0. For more information, see "Annotation Type Autowired" in the Spring documentation at http://www.springsource.org/documentation/.

7.2 Enable the Spring Web Services Client Service

MedRec-Spring uses the Spring `JaxWsPortProxyFactoryBean` to expose a dynamic proxy for Web Services, as shown in the following example from *ORACLE_HOME*`\wlserver\samples\server\medrec-spring\modules\physician\web\war\WEB-INF\applicationContext.xml`:

```
<bean id="patientService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
   <property name="serviceInterface" value="com.oracle.physician.service.PatientService"/>
   <property name="wsdlDocumentUrl"
         value="http://localhost:7011/medrec/webservices/PatientFacadeService?WSDL"/>
   <property name="namespaceUri" value="http://www.oracle.com/medrec"/>
   <property name="serviceName" value="PatientFacadeService"/>
   <property name="portName" value="PatientFacadePort"/>
</bean>

<bean id="physicianService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
   <property name="serviceInterface" value="com.oracle.physician.service.PhysicianService"/>
   <property name="wsdlDocumentUrl"
         value="http://localhost:7011/medrec/webservices/PhysicianFacadeService?WSDL"/>
  <property name="namespaceUri" value="http://www.oracle.com/medrec"/>
  <property name="serviceName" value="PhysicianFacadeService"/>
  <property name="portName" value="PhysicianFacadePort"/>
</bean>

<bean id="recordService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
  <property name="serviceInterface" value="com.oracle.physician.service.RecordService"/>
  <property name="wsdlDocumentUrl"
```

```
        value="http://localhost:7011/medrec/webservices/RecordFacadeService?WSDL"/>
  <property name="namespaceUri" value="http://www.oracle.com/medrec"/>
  <property name="serviceName" value="RecordFacadeService"/>
  <property name="portName" value="RecordFacadePort"/>
</bean>
```

With this approach, you do not have to use the tool-generated Web Services stubs. For more information, see `JaxWsPortProxyFactoryBean` in the Spring documentation at http://www.springsource.org/documentation/.

7.3 Make JMS Services Available to the Application at Runtime

In Spring, you must configure JMS services so they are provided to the application during runtime. In MedRec-Spring, Oracle made JMS services available to the application at runtime by implementing the following code in the Spring configuration file *ORACLE_HOME*\wlserver\samples\server\medrec-spring\modules\medrec\web\war\WEB-INF\applicationContext.xml:

```
<!--- Messaging ********************************** -->
   <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory"/>
   </bean>
   <jee:jndi-lookup id="connectionFactory" jndi-name="weblogic.jms.XAConnectionFactory"/>

   <jee:jndi-lookup id="patientNotificationQueue"
      jndi-name="com.oracle.medrec.jms.PatientNotificationQueue"/>
   <bean id="messageListener"
   class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
      <property name="delegate" ref="patientNotifierBroker"/>
      <property name="defaultListenerMethod" value="notifyPatient"/>
   </bean>

   <bean id="messageListenerContainer"
         class="org.springframework.jms.listener.DefaultMessageListenerContainer">
      <!--<property name="taskExecutor" ref="taskExecutor"/>-->
      <property name="connectionFactory" ref="connectionFactory"/>
      <property name="messageListener" ref="messageListener"/>
      <property name="destination" ref="patientNotificationQueue"/>
      <!-- no need to use XA transaction now -->
      <property name="sessionTransacted" value="true"/>
</bean>
```

Here, various beans from the Spring framework are declared. In particular, the `jmsTemplate` wraps the underlying JMS APIs and is used to send messages. `messageListenerContainer` provides similar functionality as the Message-Driven Bean container, and the JMS listener in MedRec-Spring is registered to it.

7.4 Use JPA Data Access

MedRec-Spring use the standard Java Persistence API (JPA) to manage data sources. The configuration is in *ORACLE_HOME*\wlserver\samples\server\medrec-spring\modules\medrec\domain\src\META-INF\persistence.xml:

```
    <persistence-unit name="MedRec" transaction-type="JTA">
        <jta-data-source>jdbc/MedRecGlobalDataSourceXA</jta-data-source>
        <class>com.oracle.medrec.model.Address</class>
```

```
        <class>com.oracle.medrec.model.Administrator</class>
        <class>com.oracle.medrec.model.BaseEntity</class>
        <class>com.oracle.medrec.model.PersonName</class>
        <class>com.oracle.medrec.model.Patient</class>
        <class>com.oracle.medrec.model.Physician</class>
        <class>com.oracle.medrec.model.Prescription</class>
        <class>com.oracle.medrec.model.Record</class>
        <class>com.oracle.medrec.model.RegularUser</class>
        <class>com.oracle.medrec.model.User</class>
        <class>com.oracle.medrec.model.VersionedEntity</class>
        <class>com.oracle.medrec.model.VitalSigns</class>
        <properties>
            <property name="kodo.jdbc.SynchronizeMappings"
                      value="buildSchema"/>
        </properties>
    </persistence-unit>
```

MedRec-Spring does not declare a Data Access Object (DAO) in the Spring configuration file, because MedRec-Spring uses the annotation-driven approach (in this case `@Repository` for all the DAOs), so all of them will be automatically managed by Spring. For example, see the following in `RecordRepositoryImpl.java`:

```
@Repository
public class RecordRepositoryImpl
         extends EntityRepositorySupport<Record, Long> implements RecordRepository {

      public List<Record> findRecordsByPatientId(Long patientId) {
         return findByProperty("Record.findRecordsByPatientId", patientId);
      }
}
```

7.5 Use the Spring Transaction Abstraction Layer for Transaction Management

MedRec-Spring uses Spring 2.5 configuration to use the WebLogic JTA transaction manager and to enable annotation-based declarative transaction management. See the configuration in *ORACLE_HOME*`\wlserver\samples\server\medrec-spring\modules\medrec\web\war\WEB-INF\applicationContext.xml`:

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

   <tx:annotation-driven/>
```

All the transaction demarcation is implemented via Spring's `@Transactional` annotation. This is similar to what is done in EJB 3.0 applications. See *ORACLE_HOME*`\wlserver\samples\server\medrec-spring\modules\medrec\domain\src\com\oracle\medrec\service\impl\RecordServiceImpl.java`:

```
@Service("recordService")
@Transactional
public class RecordServiceImpl implements RecordService {

…
   public void createRecord(Record record, Long physicianId, Long patientId) {

   }

   @Transactional(readOnly = true)
```

```
    public List<Record> getRecordsByPatientId(Long patientId) {
       }

    @Transactional(readOnly = true)
    public Record getRecord(Long id) {
    }
}
```