Integration

Table of Contents

This part of the reference documentation covers the Spring Framework's integration with a number of Java EE (and related) technologies.

Chapter 1. Remoting and web services using Spring

1.1. Introduction

Spring features integration classes for remoting support using various technologies. The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs. Currently, Spring supports the following remoting technologies:

- *Remote Method Invocation (RMI)*. Through the use of the `RmiProxyFactoryBean` and the `RmiServiceExporter` Spring supports both traditional RMI (with `java.rmi.Remote` interfaces and `java.rmi.RemoteException`) and transparent remoting via RMI invokers (with any Java interface).

- *Spring's HTTP invoker*. Spring provides a special remoting strategy which allows for Java serialization via HTTP, supporting any Java interface (just like the RMI invoker). The corresponding support classes are `HttpInvokerProxyFactoryBean` and `HttpInvokerServiceExporter`.

- *Hessian*. By using Spring's `HessianProxyFactoryBean` and the `HessianServiceExporter` you can transparently expose your services using the lightweight binary HTTP-based protocol provided by Caucho.

- *JAX-WS*. Spring provides remoting support for web services via JAX-WS (the successor of JAX-RPC, as introduced in Java EE 5 and Java 6).

- *JMS*. Remoting using JMS as the underlying protocol is supported via the `JmsInvokerServiceExporter` and `JmsInvokerProxyFactoryBean` classes.

- *AMQP*. Remoting using AMQP as the underlying protocol is supported by the Spring AMQP project.

While discussing the remoting capabilities of Spring, we'll use the following domain model and corresponding services:

```
public class Account implements Serializable{

    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

```java
public interface AccountService {

    public void insertAccount(Account account);

    public List<Account> getAccounts(String name);

}
```

```java
// the implementation doing nothing at the moment
public class AccountServiceImpl implements AccountService {

    public void insertAccount(Account acc) {
        // do something...
    }

    public List<Account> getAccounts(String name) {
        // do something...
    }

}
```

We will start exposing the service to a remote client by using RMI and talk a bit about the drawbacks of using RMI. We'll then continue to show an example using Hessian as the protocol.

1.2. Exposing services using RMI

Using Spring's support for RMI, you can transparently expose your services through the RMI infrastructure. After having this set up, you basically have a configuration similar to remote EJBs, except for the fact that there is no standard support for security context propagation or remote transaction propagation. Spring does provide hooks for such additional invocation context when using the RMI invoker, so you can for example plug in security frameworks or custom security credentials here.

1.2.1. Exporting the service using the RmiServiceExporter

Using the `RmiServiceExporter`, we can expose the interface of our AccountService object as RMI object. The interface can be accessed by using `RmiProxyFactoryBean`, or via plain RMI in case of a traditional RMI service. The `RmiServiceExporter` explicitly supports the exposing of any non-RMI services via RMI invokers.

Of course, we first have to set up our service in the Spring container:

```xml
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

Next we'll have to expose our service using the `RmiServiceExporter`:

```xml
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- does not necessarily have to be the same name as the bean to be exported -->
    <property name="serviceName" value="AccountService"/>
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

As you can see, we're overriding the port for the RMI registry. Often, your application server also maintains an RMI registry and it is wise to not interfere with that one. Furthermore, the service name is used to bind the service under. So right now, the service will be bound at `'rmi://HOST:1199/AccountService'`. We'll use the URL later on to link in the service at the client side.

> 🛈 The `servicePort` property has been omitted (it defaults to 0). This means that an anonymous port will be used to communicate with the service.

1.2.2. Linking in the service at the client

Our client is a simple object using the `AccountService` to manage accounts:

```java
public class SimpleObject {

    private AccountService accountService;

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    // additional methods using the accountService

}
```

To link in the service on the client, we'll create a separate Spring container, containing the simple object and the service linking configuration bits:

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean
">
    <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

That's all we need to do to support the remote account service on the client. Spring will transparently create an invoker and remotely enable the account service through the `RmiServiceExporter`. At the client we're linking it in using the `RmiProxyFactoryBean`.

1.3. Using Hessian to remotely call services via HTTP

Hessian offers a binary HTTP-based remoting protocol. It is developed by Caucho and more information about Hessian itself can be found at http://www.caucho.com.

1.3.1. Wiring up the DispatcherServlet for Hessian and co.

Hessian communicates via HTTP and does so using a custom servlet. Using Spring's `DispatcherServlet` principles, as known from Spring Web MVC usage, you can easily wire up such a servlet exposing your services. First we'll have to create a new servlet in your application (this is an excerpt from `'web.xml'`):

```
<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

You're probably familiar with Spring's `DispatcherServlet` principles and if so, you know that now you'll have to create a Spring container configuration resource named `'remoting-servlet.xml'` (after the name of your servlet) in the `'WEB-INF'` directory. The application context will be used in the next section.

Alternatively, consider the use of Spring's simpler `HttpRequestHandlerServlet`. This allows you to embed the remote exporter definitions in your root application context (by default in `'WEB-INF/applicationContext.xml'`), with individual servlet definitions pointing to specific exporter beans. Each servlet name needs to match the bean name of its target exporter in this case.

1.3.2. Exposing your beans by using the HessianServiceExporter

In the newly created application context called `remoting-servlet.xml`, we'll create a `HessianServiceExporter` exporting your services:

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class=
"org.springframework.remoting.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

Now we're ready to link in the service at the client. No explicit handler mapping is specified, mapping request URLs onto services, so `BeanNameUrlHandlerMapping` will be used: Hence, the service will be exported at the URL indicated through its bean name within the containing `DispatcherServlet`'s mapping (as defined above): `'http://HOST:8080/remoting/AccountService'`.

Alternatively, create a `HessianServiceExporter` in your root application context (e.g. in `'WEB-INF/applicationContext.xml'`):

```
<bean name="accountExporter" class=
"org.springframework.remoting.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

In the latter case, define a corresponding servlet for this exporter in `'web.xml'`, with the same end result: The exporter getting mapped to the request path `/remoting/AccountService`. Note that the servlet name needs to match the bean name of the target exporter.

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>
org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

1.3.3. Linking in the service on the client

Using the `HessianProxyFactoryBean` we can link in the service at the client. The same principles apply

as with the RMI example. We'll create a separate bean factory or application context and mention the following beans where the `SimpleObject` is using the `AccountService` to manage accounts:

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class=
"org.springframework.remoting.caucho.HessianProxyFactoryBean">
    <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService
"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

1.3.4. Applying HTTP basic authentication to a service exposed through Hessian

One of the advantages of Hessian is that we can easily apply HTTP basic authentication, because both protocols are HTTP-based. Your normal HTTP server security mechanism can easily be applied through using the `web.xml` security features, for example. Usually, you don't use per-user security credentials here, but rather shared credentials defined at the `HessianProxyFactoryBean` level (similar to a JDBC `DataSource`).

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors" ref="authorizationInterceptor"/>
</bean>

<bean id="authorizationInterceptor"
        class=
"org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
    <property name="authorizedRoles" value="administrator,operator"/>
</bean>
```

This is an example where we explicitly mention the `BeanNameUrlHandlerMapping` and set an interceptor allowing only administrators and operators to call the beans mentioned in this application context.

> Of course, this example doesn't show a flexible kind of security infrastructure. For more options as far as security is concerned, have a look at the Spring Security project at http://projects.spring.io/spring-security/.

1.4. Exposing services using HTTP invokers

As opposed to Hessian, which are both lightweight protocols using their own slim serialization mechanisms, Spring HTTP invokers use the standard Java serialization mechanism to expose services through HTTP. This has a huge advantage if your arguments and return types are complex

types that cannot be serialized using the serialization mechanisms Hessian uses (refer to the next section for more considerations when choosing a remoting technology).

Under the hood, Spring uses either the standard facilities provided by the JDK or Apache `HttpComponents` to perform HTTP calls. Use the latter if you need more advanced and easier-to-use functionality. Refer to hc.apache.org/httpcomponents-client-ga/ for more information.

> ⚠️ Be aware of vulnerabilities due to unsafe Java deserialization: Manipulated input streams could lead to unwanted code execution on the server during the deserialization step. As a consequence, do not expose HTTP invoker endpoints to untrusted clients but rather just between your own services. In general, we strongly recommend any other message format (e.g. JSON) instead.
>
> If you are concerned about security vulnerabilities due to Java serialization, consider the general-purpose serialization filter mechanism at the core JVM level, originally developed for JDK 9 but backported to JDK 8, 7 and 6 in the meantime: https://blogs.oracle.com/java-platform-group/entry/incoming_filter_serialization_data_a http://openjdk.java.net/jeps/290

1.4.1. Exposing the service object

Setting up the HTTP invoker infrastructure for a service object resembles closely the way you would do the same using Hessian. Just as Hessian support provides the `HessianServiceExporter`, Spring's HttpInvoker support provides the `org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter`.

To expose the `AccountService` (mentioned above) within a Spring Web MVC `DispatcherServlet`, the following configuration needs to be in place in the dispatcher's application context:

```
<bean name="/AccountService" class=
"org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

Such an exporter definition will be exposed through the `DispatcherServlet`'s standard mapping facilities, as explained in the section on Hessian.

Alternatively, create an `HttpInvokerServiceExporter` in your root application context (e.g. in `'WEB-INF/applicationContext.xml'`):

```
<bean name="accountExporter" class=
"org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

In addition, define a corresponding servlet for this exporter in `'web.xml'`, with the servlet name matching the bean name of the target exporter:

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>
org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

If you are running outside of a servlet container and are using Oracle's Java 6, then you can use the built-in HTTP server implementation. You can configure the `SimpleHttpServerFactoryBean` together with a `SimpleHttpInvokerServiceExporter` as is shown in this example:

```
<bean name="accountExporter"
        class=
"org.springframework.remoting.httpinvoker.SimpleHttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>

<bean id="httpServer"
        class="org.springframework.remoting.support.SimpleHttpServerFactoryBean">
    <property name="contexts">
        <util:map>
            <entry key="/remoting/AccountService" value-ref="accountExporter"/>
        </util:map>
    </property>
    <property name="port" value="8080" />
</bean>
```

1.4.2. Linking in the service at the client

Again, linking in the service from the client much resembles the way you would do it when using Hessian. Using a proxy, Spring will be able to translate your calls to HTTP POST requests to the URL pointing to the exported service.

```
<bean id="httpInvokerProxy" class=
"org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService
"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

As mentioned before, you can choose what HTTP client you want to use. By default, the `HttpInvokerProxy` uses the JDK's HTTP functionality, but you can also use the Apache `HttpComponents` client by setting the `httpInvokerRequestExecutor` property:

```
<property name="httpInvokerRequestExecutor">
    <bean class=
"org.springframework.remoting.httpinvoker.HttpComponentsHttpInvokerRequestExecutor"/>
</property>
```

1.5. Web services

Spring provides full support for standard Java web services APIs:

- Exposing web services using JAX-WS

- Accessing web services using JAX-WS

In addition to stock support for JAX-WS in Spring Core, the Spring portfolio also features Spring Web Services, a solution for contract-first, document-driven web services - highly recommended for building modern, future-proof web services.

1.5.1. Exposing servlet-based web services using JAX-WS

Spring provides a convenient base class for JAX-WS servlet endpoint implementations - `SpringBeanAutowiringSupport`. To expose our `AccountService` we extend Spring's `SpringBeanAutowiringSupport` class and implement our business logic here, usually delegating the call to the business layer. We'll simply use Spring's `@Autowired` annotation for expressing such dependencies on Spring-managed beans.

```java
/**
 * JAX-WS compliant AccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
 *
 * This wrapper class is necessary because JAX-WS requires working with dedicated
 * endpoint classes. If an existing service needs to be exported, a wrapper that
 * extends SpringBeanAutowiringSupport for simple Spring bean autowiring (through
 * the @Autowired annotation) is the simplest JAX-WS compliant way.
 *
 * This is the class registered with the server-side JAX-WS implementation.
 * In the case of a Java EE 5 server, this would simply be defined as a servlet
 * in web.xml, with the server detecting that this is a JAX-WS endpoint and reacting
 * accordingly. The servlet name usually needs to match the specified WS service name.
 *
 * The web service engine manages the lifecycle of instances of this class.
 * Spring bean references will just be wired in here.
 */
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

@WebService(serviceName="AccountService")
public class AccountServiceEndpoint extends SpringBeanAutowiringSupport {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public Account[] getAccounts(String name) {
        return biz.getAccounts(name);
    }

}
```

Our `AccountServiceEndpoint` needs to run in the same web application as the Spring context to allow for access to Spring's facilities. This is the case by default in Java EE 5 environments, using the standard contract for JAX-WS servlet endpoint deployment. See Java EE 5 web service tutorials for details.

1.5.2. Exporting standalone web services using JAX-WS

The built-in JAX-WS provider that comes with Oracle's JDK supports exposure of web services using the built-in HTTP server that's included in the JDK as well. Spring's `SimpleJaxWsServiceExporter` detects all `@WebService` annotated beans in the Spring application context, exporting them through the default JAX-WS server (the JDK HTTP server).

In this scenario, the endpoint instances are defined and managed as Spring beans themselves; they will be registered with the JAX-WS engine but their lifecycle will be up to the Spring application context. This means that Spring functionality like explicit dependency injection may be applied to the endpoint instances. Of course, annotation-driven injection through @Autowired will work as well.

```
<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">
    <property name="baseAddress" value="http://localhost:8080/"/>
</bean>

<bean id="accountServiceEndpoint" class="example.AccountServiceEndpoint">
    ...
</bean>

...
```

The AccountServiceEndpoint may derive from Spring's SpringBeanAutowiringSupport but doesn't have to since the endpoint is a fully Spring-managed bean here. This means that the endpoint implementation may look like as follows, without any superclass declared - and Spring's @Autowired configuration annotation still being honored:

```
@WebService(serviceName="AccountService")
public class AccountServiceEndpoint {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public List<Account> getAccounts(String name) {
        return biz.getAccounts(name);
    }

}
```

1.5.3. Exporting web services using the JAX-WS RI's Spring support

Oracle's JAX-WS RI, developed as part of the GlassFish project, ships Spring support as part of its JAX-WS Commons project. This allows for defining JAX-WS endpoints as Spring-managed beans, similar to the standalone mode discussed in the previous section - but this time in a Servlet environment. *Note that this is not portable in a Java EE 5 environment; it is mainly intended for non-EE environments such as Tomcat, embedding the JAX-WS RI as part of the web application.*

The difference to the standard style of exporting servlet-based endpoints is that the lifecycle of the endpoint instances themselves will be managed by Spring here, and that there will be only one JAX-

WS servlet defined in `web.xml`. With the standard Java EE 5 style (as illustrated above), you'll have one servlet definition per service endpoint, with each endpoint typically delegating to Spring beans (through the use of `@Autowired`, as shown above).

Check out https://jax-ws-commons.java.net/spring/ for details on setup and usage style.

1.5.4. Accessing web services using JAX-WS

Spring provides two factory beans to create JAX-WS web service proxies, namely `LocalJaxWsServiceFactoryBean` and `JaxWsPortProxyFactoryBean`. The former can only return a JAX-WS service class for us to work with. The latter is the full-fledged version that can return a proxy that implements our business service interface. In this example we use the latter to create a proxy for the `AccountService` endpoint (again):

```
<bean id="accountWebService" class=
"org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="example.AccountService"/>
    <property name="wsdlDocumentUrl" value=
"http://localhost:8888/AccountServiceEndpoint?WSDL"/>
    <property name="namespaceUri" value="http://example/"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountServiceEndpointPort"/>
</bean>
```

Where `serviceInterface` is our business interface the clients will use. `wsdlDocumentUrl` is the URL for the WSDL file. Spring needs this a startup time to create the JAX-WS Service. `namespaceUri` corresponds to the targetNamespace in the .wsdl file. `serviceName` corresponds to the service name in the .wsdl file. `portName` corresponds to the port name in the .wsdl file.

Accessing the web service is now very easy as we have a bean factory for it that will expose it as `AccountService` interface. We can wire this up in Spring:

```
<bean id="client" class="example.AccountClientImpl">
    ...
    <property name="service" ref="accountWebService"/>
</bean>
```

From the client code we can access the web service just as if it was a normal class:

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
    }
}
```

The above is slightly simplified in that JAX-WS requires endpoint interfaces and implementation classes to be annotated with @WebService, @SOAPBinding etc annotations. This means that you cannot (easily) use plain Java interfaces and implementation classes as JAX-WS endpoint artifacts; you need to annotate them accordingly first. Check the JAX-WS documentation for details on those requirements.

1.6. JMS

It is also possible to expose services transparently using JMS as the underlying communication protocol. The JMS remoting support in the Spring Framework is pretty basic - it sends and receives on the `same thread` and in the *same non-transactional* `Session`, and as such throughput will be very implementation dependent. Note that these single-threaded and non-transactional constraints apply only to Spring's JMS *remoting* support. See JMS (Java Message Service) for information on Spring's rich support for JMS-based *messaging*.

The following interface is used on both the server and the client side.

```
package com.foo;

public interface CheckingAccountService {

    public void cancelAccount(Long accountId);

}
```

The following simple implementation of the above interface is used on the server-side.

```java
package com.foo;

public class SimpleCheckingAccountService implements CheckingAccountService {

    public void cancelAccount(Long accountId) {
        System.out.println("Cancelling account [" + accountId + "]");
    }


}
```

This configuration file contains the JMS-infrastructure beans that are shared on both the client and server.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory
">
        <property name="brokerURL" value="tcp://ep-t43:61616"/>
    </bean>

    <bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="mmm"/>
    </bean>

</beans>
```

1.6.1. Server-side configuration

On the server, you just need to expose the service object using the `JmsInvokerServiceExporter`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="checkingAccountService"
            class="org.springframework.jms.remoting.JmsInvokerServiceExporter">
        <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
        <property name="service">
            <bean class="com.foo.SimpleCheckingAccountService"/>
        </property>
    </bean>

    <bean class="org.springframework.jms.listener.SimpleMessageListenerContainer">
        <property name="connectionFactory" ref="connectionFactory"/>
        <property name="destination" ref="queue"/>
        <property name="concurrentConsumers" value="3"/>
        <property name="messageListener" ref="checkingAccountService"/>
    </bean>

</beans>
```

```java
package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext(new String[]{"com/foo/server.xml",
"com/foo/jms.xml"});
    }

}
```

1.6.2. Client-side configuration

The client merely needs to create a client-side proxy that will implement the agreed upon interface
(CheckingAccountService). The resulting object created off the back of the following bean definition
can be injected into other client side objects, and the proxy will take care of forwarding the call to
the server-side object via JMS.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="checkingAccountService"
            class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
        <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
        <property name="connectionFactory" ref="connectionFactory"/>
        <property name="queue" ref="queue"/>
    </bean>

</beans>
```

```java
package com.foo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
                new String[] {"com/foo/client.xml", "com/foo/jms.xml"});
        CheckingAccountService service = (CheckingAccountService) ctx.getBean(
"checkingAccountService");
        service.cancelAccount(new Long(10));
    }

}
```

1.7. AMQP

Refer to the Spring AMQP Reference Document 'Spring Remoting with AMQP' section for more information.

1.8. Auto-detection is not implemented for remote interfaces

The main reason why auto-detection of implemented interfaces does not occur for remote interfaces is to avoid opening too many doors to remote callers. The target object might implement internal callback interfaces like `InitializingBean` or `DisposableBean` which one would not want to expose to callers.

Offering a proxy with all interfaces implemented by the target usually does not matter in the local

case. But when exporting a remote service, you should expose a specific service interface, with specific operations intended for remote usage. Besides internal callback interfaces, the target might implement multiple business interfaces, with just one of them intended for remote exposure. For these reasons, we *require* such a service interface to be specified.

This is a trade-off between configuration convenience and the risk of accidental exposure of internal methods. Always specifying a service interface is not too much effort, and puts you on the safe side regarding controlled exposure of specific methods.

1.9. Considerations when choosing a technology

Each and every technology presented here has its drawbacks. You should carefully consider your needs, the services you are exposing and the objects you'll be sending over the wire when choosing a technology.

When using RMI, it's not possible to access the objects through the HTTP protocol, unless you're tunneling the RMI traffic. RMI is a fairly heavy-weight protocol in that it supports full-object serialization which is important when using a complex data model that needs serialization over the wire. However, RMI-JRMP is tied to Java clients: It is a Java-to-Java remoting solution.

Spring's HTTP invoker is a good choice if you need HTTP-based remoting but also rely on Java serialization. It shares the basic infrastructure with RMI invokers, just using HTTP as transport. Note that HTTP invokers are not only limited to Java-to-Java remoting but also to Spring on both the client and server side. (The latter also applies to Spring's RMI invoker for non-RMI interfaces.)

Hessian might provide significant value when operating in a heterogeneous environment, because they explicitly allow for non-Java clients. However, non-Java support is still limited. Known issues include the serialization of Hibernate objects in combination with lazily-initialized collections. If you have such a data model, consider using RMI or HTTP invokers instead of Hessian.

JMS can be useful for providing clusters of services and allowing the JMS broker to take care of load balancing, discovery and auto-failover. By default: Java serialization is used when using JMS remoting but the JMS provider could use a different mechanism for the wire formatting, such as XStream to allow servers to be implemented in other technologies.

Last but not least, EJB has an advantage over RMI in that it supports standard role-based authentication and authorization and remote transaction propagation. It is possible to get RMI invokers or HTTP invokers to support security context propagation as well, although this is not provided by core Spring: There are just appropriate hooks for plugging in third-party or custom solutions here.

1.10. Accessing REST endpoints

The Spring Framework has two choices for client-side access to REST endpoints:

- RestTemplate — the original Spring REST client with an API similar to other template classes in Spring such as `JdbcTemplate`, `JmsTemplate` and others. `RestTemplate` has a synchronous API and relies on blocking I/O which is okay for client scenarios with low concurrency.

- **WebClient** — reactive client with a functional, fluent API from the `spring-webflux` module. It relies on non-blocking I/O which allows it to support high concurrency more efficiently (i.e. using a small number of threads) than the `RestTemplate`. `WebClient` is a natural fit for streaming scenarios.

1.10.1. RestTemplate

The `RestTemplate` provides a higher level API over HTTP client libraries with methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and enforce REST best practices.

Table 1. Overview of RestTemplate methods

HTTP Method	RestTemplate Method
DELETE	delete
GET	getForObject getForEntity
HEAD	headForHeaders(String url, String… uriVariables)
OPTIONS	optionsForAllow(String url, String… uriVariables)
POST	postForLocation(String url, Object request, String… uriVariables) postForObject(String url, Object request, Class<T> responseType, String… uriVariables)
PUT	put(String url, Object request, String…uriVariables)
PATCH and others	exchange execute

The names of `RestTemplate` methods follow a naming convention, the first part indicates what HTTP method is being invoked and the second part indicates what is returned. For example, the method `getForObject()` will perform a GET, convert the HTTP response into an object type of your choice and return that object. The method `postForLocation()` will do a POST, converting the given object into a HTTP request and return the response HTTP Location header where the newly created object can be found. In case of an exception processing the HTTP request, an exception of the type `RestClientException` will be thrown; this behavior can be changed by plugging in another `ResponseErrorHandler` implementation into the `RestTemplate`.

The `exchange` and `execute` methods are generalized versions of the more specific methods listed above them and can support additional combinations and methods, like HTTP PATCH. However, note that the underlying HTTP library must also support the desired combination. The JDK `HttpURLConnection` does not support the `PATCH` method, but Apache HttpComponents HttpClient version 4.2 or later does. They also enable `RestTemplate` to read an HTTP response to a generic type (e.g. `List<Account>`), using a `ParameterizedTypeReference`, a new class that enables capturing and passing generic type info.

Objects passed to and returned from these methods are converted to and from HTTP messages by `HttpMessageConverter` instances. Converters for the main mime types are registered by default, but you can also write your own converter and register it via the `messageConverters()` bean property. The default converter instances registered with the template are `ByteArrayHttpMessageConverter`, `StringHttpMessageConverter`, `FormHttpMessageConverter` and `SourceHttpMessageConverter`. You can override these defaults using the `messageConverters()` bean property as would be required if using

the `MarshallingHttpMessageConverter` or `MappingJackson2HttpMessageConverter`.

Each method takes URI template arguments in two forms, either as a `String` variable-length argument or a `Map<String,String>`. For example,

```
String result = restTemplate.getForObject(
        "http://example.com/hotels/{hotel}/bookings/{booking}", String.class,"42", "
21");
```

using variable-length arguments and

```
Map<String, String> vars = Collections.singletonMap("hotel", "42");
String result = restTemplate.getForObject(
        "http://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);
```

using a `Map<String,String>`.

To create an instance of `RestTemplate` you can simply call the default no-arg constructor. This will use standard Java classes from the `java.net` package as the underlying implementation to create HTTP requests. This can be overridden by specifying an implementation of `ClientHttpRequestFactory`. Spring provides the implementation `HttpComponentsClientHttpRequestFactory` that uses the Apache HttpComponents `HttpClient` to create requests. `HttpComponentsClientHttpRequestFactory` is configured using an instance of `org.apache.http.client.HttpClient` which can in turn be configured with credentials information or connection pooling functionality.

> 💡 Note that the `java.net` implementation for HTTP requests may raise an exception when accessing the status of a response that represents an error (e.g. 401). If this is an issue, switch to `HttpComponentsClientHttpRequestFactory` instead.

The previous example using Apache HttpComponents `HttpClient` directly rewritten to use the `RestTemplate` is shown below

```
uri = "http://example.com/hotels/{id}/bookings";

RestTemplate template = new RestTemplate();

Booking booking = // create booking object

URI location = template.postForLocation(uri, booking, "1");
```

To use Apache HttpComponents instead of the native `java.net` functionality, construct the `RestTemplate` as follows:

```
RestTemplate template = new RestTemplate(new HttpComponentsClientHttpRequestFactory()
);
```

> 💡 Apache HttpClient supports gzip encoding. To use it, construct a `HttpComponentsClientHttpRequestFactory` like so:
>
> ```
> HttpClient httpClient = HttpClientBuilder.create().build();
> ClientHttpRequestFactory requestFactory = new
> HttpComponentsClientHttpRequestFactory(httpClient);
> RestTemplate restTemplate = new RestTemplate(requestFactory);
> ```

The general callback interface is `RequestCallback` and is called when the execute method is invoked.

```
public <T> T execute(String url, HttpMethod method, RequestCallback requestCallback,
        ResponseExtractor<T> responseExtractor, String... uriVariables)

// also has an overload with uriVariables as a Map<String, String>.
```

The `RequestCallback` interface is defined as

```
public interface RequestCallback {
 void doWithRequest(ClientHttpRequest request) throws IOException;
}
```

and allows you to manipulate the request headers and write to the request body. When using the execute method you do not have to worry about any resource management, the template will always close the request and handle any errors. Refer to the API documentation for more information on using the execute method and the meaning of its other method arguments.

Working with the URI

For each of the main HTTP methods, the `RestTemplate` provides two variants that take either a String URI template or `java.net.URI` as the first argument. When using a String URI template, encoding is automatically applied:

```
restTemplate.getForObject("http://example.com/hotel list", String.class);
```

The resulting target URI is "http://example.com/hotel%20list". Alternatively you can provide an already prepared `java.net.URI` that will be used as is. For more information on preparing URIs or customizing how the `RestTemplate` expands URI templates, see URI Links in the "Web Servlet" section.

Dealing with request and response headers

Besides the methods described above, the `RestTemplate` also has the `exchange()` method, which can be used for arbitrary HTTP method execution based on the `HttpEntity` class.

Perhaps most importantly, the `exchange()` method can be used to add request headers and read response headers. For example:

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.set("MyRequestHeader", "MyValue");
HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);

HttpEntity<String> response = template.exchange(
        "http://example.com/hotels/{hotel}",
        HttpMethod.GET, requestEntity, String.class, "42");

String responseHeader = response.getHeaders().getFirst("MyResponseHeader");
String body = response.getBody();
```

In the above example, we first prepare a request entity that contains the `MyRequestHeader` header. We then retrieve the response, and read the `MyResponseHeader` and body.

Jackson JSON Views support

It is possible to specify a Jackson JSON View to serialize only a subset of the object properties. For example:

```
MappingJacksonValue value = new MappingJacksonValue(new User("eric", "7!jd#h23"));
value.setSerializationView(User.WithoutPasswordView.class);
HttpEntity<MappingJacksonValue> entity = new HttpEntity<MappingJacksonValue>(value);
String s = template.postForObject("http://example.com/user", entity, String.class);
```

1.10.2. HTTP Message Converters

Same in Spring WebFlux

The `spring-web` module contains the `HttpMessageConverter` contract for reading and writing the body of HTTP requests and responses via `InputStream` and `OutputStream`. `HttpMessageConverter`'s are used on the client side, e.g. in the `RestTemplate`, and also on the server side, e.g. in Spring MVC REST controllers.

Concrete implementations for the main media (mime) types are provided in the framework and are registered by default with the `RestTemplate` on the client-side and with `RequestMethodHandlerAdapter` on the server-side (see Configuring Message Converters).

The implementations of `HttpMessageConverter`s are described in the following sections. For all converters a default media type is used but can be overridden by setting the `supportedMediaTypes` bean property

StringHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write Strings from the HTTP request and response. By default, this converter supports all text media types (`text/*`), and writes with a `Content-Type` of `text/plain`.

FormHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write form data from the HTTP request and response. By default, this converter reads and writes the media type `application/x-www-form-urlencoded`. Form data is read from and written into a `MultiValueMap<String, String>`.

ByteArrayHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write byte arrays from the HTTP request and response. By default, this converter supports all media types (`*/*`), and writes with a `Content-Type` of `application/octet-stream`. This can be overridden by setting the `supportedMediaTypes` property, and overriding `getContentType(byte[])`.

MarshallingHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write XML using Spring's `Marshaller` and `Unmarshaller` abstractions from the `org.springframework.oxm` package. This converter requires a `Marshaller` and `Unmarshaller` before it can be used. These can be injected via constructor or bean properties. By default this converter supports (`text/xml`) and (`application/xml`).

MappingJackson2HttpMessageConverter

An `HttpMessageConverter` implementation that can read and write JSON using Jackson's `ObjectMapper`. JSON mapping can be customized as needed through the use of Jackson's provided annotations. When further control is needed, a custom `ObjectMapper` can be injected through the `ObjectMapper` property for cases where custom JSON serializers/deserializers need to be provided for specific types. By default this converter supports (`application/json`).

MappingJackson2XmlHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write XML using [Jackson XML](#) extension's `XmlMapper`. XML mapping can be customized as needed through the use of JAXB or Jackson's provided annotations. When further control is needed, a custom `XmlMapper` can be injected through the `ObjectMapper` property for cases where custom XML serializers/deserializers need to be provided for specific types. By default this converter supports (`application/xml`).

SourceHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write `javax.xml.transform.Source` from the HTTP request and response. Only `DOMSource`, `SAXSource`, and `StreamSource` are supported. By default, this converter supports (`text/xml`) and (`application/xml`).

BufferedImageHttpMessageConverter

An `HttpMessageConverter` implementation that can read and write `java.awt.image.BufferedImage` from the HTTP request and response. This converter reads and writes the media type supported by the Java I/O API.

1.10.3. Async RestTemplate

The `AsyncRestTemplate` is deprecated. Please use the WebClient instead.

Chapter 2. Enterprise JavaBeans (EJB) integration

2.1. Introduction

As a lightweight container, Spring is often considered an EJB replacement. We do believe that for many if not most applications and use cases, Spring as a container, combined with its rich supporting functionality in the area of transactions, ORM and JDBC access, is a better choice than implementing equivalent functionality via an EJB container and EJBs.

However, it is important to note that using Spring does not prevent you from using EJBs. In fact, Spring makes it much easier to access EJBs and implement EJBs and functionality within them. Additionally, using Spring to access services provided by EJBs allows the implementation of those services to later transparently be switched between local EJB, remote EJB, or POJO (plain old Java object) variants, without the client code having to be changed.

In this chapter, we look at how Spring can help you access and implement EJBs. Spring provides particular value when accessing stateless session beans (SLSBs), so we'll begin by discussing this.

2.2. Accessing EJBs

2.2.1. Concepts

To invoke a method on a local or remote stateless session bean, client code must normally perform a JNDI lookup to obtain the (local or remote) EJB Home object, then use a 'create' method call on that object to obtain the actual (local or remote) EJB object. One or more methods are then invoked on the EJB.

To avoid repeated low-level code, many EJB applications use the Service Locator and Business Delegate patterns. These are better than spraying JNDI lookups throughout client code, but their usual implementations have significant disadvantages. For example:

- Typically code using EJBs depends on Service Locator or Business Delegate singletons, making it hard to test.

- In the case of the Service Locator pattern used without a Business Delegate, application code still ends up having to invoke the create() method on an EJB home, and deal with the resulting exceptions. Thus it remains tied to the EJB API and the complexity of the EJB programming model.

- Implementing the Business Delegate pattern typically results in significant code duplication, where we have to write numerous methods that simply call the same method on the EJB.

The Spring approach is to allow the creation and use of proxy objects, normally configured inside a Spring container, which act as codeless business delegates. You do not need to write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you are actually adding real value in such code.

2.2.2. Accessing local SLSBs

Assume that we have a web controller that needs to use a local EJB. We'll follow best practice and use the EJB Business Methods Interface pattern, so that the EJB's local interface extends a non EJB-specific business methods interface. Let's call this business methods interface `MyComponent`.

```java
public interface MyComponent {
    ...
}
```

One of the main reasons to use the Business Methods Interface pattern is to ensure that synchronization between method signatures in local interface and bean implementation class is automatic. Another reason is that it later makes it much easier for us to switch to a POJO (plain old Java object) implementation of the service if it makes sense to do so. Of course we'll also need to implement the local home interface and provide an implementation class that implements `SessionBean` and the `MyComponent` business methods interface. Now the only Java coding we'll need to do to hook up our web tier controller to the EJB implementation is to expose a setter method of type `MyComponent` on the controller. This will save the reference as an instance variable in the controller:

```java
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

We can subsequently use this instance variable in any business method in the controller. Now assuming we are obtaining our controller object out of a Spring container, we can (in the same context) configure a `LocalStatelessSessionProxyFactoryBean` instance, which will be the EJB proxy object. The configuration of the proxy, and setting of the `myComponent` property of the controller is done with a configuration entry such as:

```xml
<bean id="myComponent"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/myBean"/>
    <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

There's a lot of work happening behind the scenes, courtesy of the Spring AOP framework, although you aren't forced to work with AOP concepts to enjoy the results. The `myComponent` bean definition creates a proxy for the EJB, which implements the business method interface. The EJB local home is cached on startup, so there's only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the `classname` method on the local EJB and invokes the corresponding business method on

the EJB.

The `myController` bean definition sets the `myComponent` property of the controller class to the EJB proxy.

Alternatively (and preferably in case of many such proxy definitions), consider using the `<jee:local-slsb>` configuration element in Spring's "jee" namespace:

```xml
<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"
        business-interface="com.mycom.MyComponent"/>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

This EJB access mechanism delivers huge simplification of application code: the web tier code (or other EJB client code) has no dependence on the use of EJB. If we want to replace this EJB reference with a POJO or a mock object or other test stub, we could simply change the `myComponent` bean definition without changing a line of Java code. Additionally, we haven't had to write a single line of JNDI lookup or other EJB plumbing code as part of our application.

Benchmarks and experience in real applications indicate that the performance overhead of this approach (which involves reflective invocation of the target EJB) is minimal, and is typically undetectable in typical use. Remember that we don't want to make fine-grained calls to EJBs anyway, as there's a cost associated with the EJB infrastructure in the application server.

There is one caveat with regards to the JNDI lookup. In a bean container, this class is normally best used as a singleton (there simply is no reason to make it a prototype). However, if that bean container pre-instantiates singletons (as do the various XML `ApplicationContext` variants) you may have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup will be performed in the `init()` method of this class and then cached, but the EJB will not have been bound at the target location yet. The solution is to not pre-instantiate this factory object, but allow it to be created on first use. In the XML containers, this is controlled via the `lazy-init` attribute.

Although this will not be of interest to the majority of Spring users, those doing programmatic AOP work with EJBs may want to look at `LocalSlsbInvokerInterceptor`.

2.2.3. Accessing remote SLSBs

Accessing remote EJBs is essentially identical to accessing local EJBs, except that the `SimpleRemoteStatelessSessionProxyFactoryBean` or `<jee:remote-slsb>` configuration element is used. Of course, with or without Spring, remote invocation semantics apply; a call to a method on an object in another VM in another computer does sometimes have to be treated differently in terms of usage scenarios and failure handling.

Spring's EJB client support adds one more advantage over the non-Spring approach. Normally it is problematic for EJB client code to be easily switched back and forth between calling EJBs locally or remotely. This is because the remote interface methods must declare that they throw

`RemoteException`, and client code must deal with this, while the local interface methods don't. Client code written for local EJBs which needs to be moved to remote EJBs typically has to be modified to add handling for the remote exceptions, and client code written for remote EJBs which needs to be moved to local EJBs, can either stay the same but do a lot of unnecessary handling of remote exceptions, or needs to be modified to remove that code. With the Spring remote EJB proxy, you can instead not declare any thrown `RemoteException` in your Business Method Interface and implementing EJB code, have a remote interface which is identical except that it does throw `RemoteException`, and rely on the proxy to dynamically treat the two interfaces as if they were the same. That is, client code does not have to deal with the checked `RemoteException` class. Any actual `RemoteException` that is thrown during the EJB invocation will be re-thrown as the non-checked `RemoteAccessException` class, which is a subclass of `RuntimeException`. The target service can then be switched at will between a local EJB or remote EJB (or even plain Java object) implementation, without the client code knowing or caring. Of course, this is optional; there is nothing stopping you from declaring `RemoteExceptions` in your business interface.

2.2.4. Accessing EJB 2.x SLSBs versus EJB 3 SLSBs

Accessing EJB 2.x Session Beans and EJB 3 Session Beans via Spring is largely transparent. Spring's EJB accessors, including the `<jee:local-slsb>` and `<jee:remote-slsb>` facilities, transparently adapt to the actual component at runtime. They handle a home interface if found (EJB 2.x style), or perform straight component invocations if no home interface is available (EJB 3 style).

Note: For EJB 3 Session Beans, you could effectively use a `JndiObjectFactoryBean` / `<jee:jndi-lookup>` as well, since fully usable component references are exposed for plain JNDI lookups there. Defining explicit `<jee:local-slsb>` / `<jee:remote-slsb>` lookups simply provides consistent and more explicit EJB access configuration.

Chapter 3. JMS (Java Message Service)

3.1. Introduction

Spring provides a JMS integration framework that simplifies the use of the JMS API much like Spring's integration does for the JDBC API.

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. The `JmsTemplate` class is used for message production and synchronous message reception. For asynchronous reception similar to Java EE's message-driven bean style, Spring provides a number of message listener containers that are used to create Message-Driven POJOs (MDPs). Spring also provides a declarative way of creating message listeners.

The package `org.springframework.jms.core` provides the core functionality for using JMS. It contains JMS template classes that simplify the use of the JMS by handling the creation and release of resources, much like the `JdbcTemplate` does for JDBC. The design principle common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The JMS template follows the same design. The classes offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the JMS session and message producer to the user.

The package `org.springframework.jms.support` provides `JMSException` translation functionality. The translation converts the checked `JMSException` hierarchy to a mirrored hierarchy of unchecked exceptions. If there are any provider specific subclasses of the checked `javax.jms.JMSException`, this exception is wrapped in the unchecked `UncategorizedJmsException`.

The package `org.springframework.jms.support.converter` provides a `MessageConverter` abstraction to convert between Java objects and JMS messages.

The package `org.springframework.jms.support.destination` provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

The package `org.springframework.jms.annotation` provides the necessary infrastructure to support annotation-driven listener endpoints using `@JmsListener`.

The package `org.springframework.jms.config` provides the parser implementation for the `jms` namespace as well the java config support to configure listener containers and create listener endpoints.

Finally, the package `org.springframework.jms.connection` provides an implementation of the `ConnectionFactory` suitable for use in standalone applications. It also contains an implementation of Spring's `PlatformTransactionManager` for JMS (the cunningly named `JmsTransactionManager`). This allows for seamless integration of JMS as a transactional resource into Spring's transaction management mechanisms.

3.2. Using Spring JMS

3.2.1. JmsTemplate

The `JmsTemplate` class is the central class in the JMS core package. It simplifies the use of JMS since it handles the creation and release of resources when sending or synchronously receiving messages.

Code that uses the `JmsTemplate` only needs to implement callback interfaces giving them a clearly defined high level contract. The `MessageCreator` callback interface creates a message given a `Session` provided by the calling code in `JmsTemplate`. In order to allow for more complex usage of the JMS API, the callback `SessionCallback` provides the user with the JMS session and the callback `ProducerCallback` exposes a `Session` and `MessageProducer` pair.

The JMS API exposes two types of send methods, one that takes delivery mode, priority, and time-to-live as Quality of Service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many send methods in `JmsTemplate`, the setting of the QOS parameters have been exposed as bean properties to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property `setReceiveTimeout`.

Some JMS providers allow the setting of default QOS values administratively through the configuration of the `ConnectionFactory`. This has the effect that a call to `MessageProducer`'s send method `send(Destination destination, Message message)` will use different QOS default values than those specified in the JMS specification. In order to provide consistent management of QOS values, the `JmsTemplate` must therefore be specifically enabled to use its own QOS values by setting the boolean property `isExplicitQosEnabled` to `true`.

For convenience, `JmsTemplate` also exposes a basic request-reply operation that allows to send a message and wait for a reply on a temporary queue that is created as part of the operation.

> Instances of the `JmsTemplate` class are *thread-safe once configured*. This is important because it means that you can configure a single instance of a `JmsTemplate` and then safely inject this *shared* reference into multiple collaborators. To be clear, the `JmsTemplate` is stateful, in that it maintains a reference to a `ConnectionFactory`, but this state is *not* conversational state.

As of Spring Framework 4.1, `JmsMessagingTemplate` is built on top of `JmsTemplate` and provides an integration with the messaging abstraction, i.e. `org.springframework.messaging.Message`. This allows you to create the message to send in generic manner.

3.2.2. Connections

The `JmsTemplate` requires a reference to a `ConnectionFactory`. The `ConnectionFactory` is part of the JMS specification and serves as the entry point for working with JMS. It is used by the client application as a factory to create connections with the JMS provider and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

When using JMS inside an EJB, the vendor provides implementations of the JMS interfaces so that they can participate in declarative transaction management and perform pooling of connections

and sessions. In order to use this implementation, Java EE containers typically require that you declare a JMS connection factory as a `resource-ref` inside the EJB or servlet deployment descriptors. To ensure the use of these features with the `JmsTemplate` inside an EJB, the client application should ensure that it references the managed implementation of the `ConnectionFactory`.

Caching Messaging Resources

The standard API involves creating many intermediate objects. To send a message the following 'API' walk is performed

```
ConnectionFactory->Connection->Session->MessageProducer->send
```

Between the ConnectionFactory and the Send operation there are three intermediate objects that are created and destroyed. To optimise the resource usage and increase performance two implementations of `ConnectionFactory` are provided.

SingleConnectionFactory

Spring provides an implementation of the `ConnectionFactory` interface, `SingleConnectionFactory`, that will return the same `Connection` on all `createConnection()` calls and ignore calls to `close()`. This is useful for testing and standalone environments so that the same connection can be used for multiple `JmsTemplate` calls that may span any number of transactions. `SingleConnectionFactory` takes a reference to a standard `ConnectionFactory` that would typically come from JNDI.

CachingConnectionFactory

The `CachingConnectionFactory` extends the functionality of `SingleConnectionFactory` and adds the caching of Sessions, MessageProducers, and MessageConsumers. The initial cache size is set to 1, use the property `sessionCacheSize` to increase the number of cached sessions. Note that the number of actual cached sessions will be more than that number as sessions are cached based on their acknowledgment mode, so there can be up to 4 cached session instances when `sessionCacheSize` is set to one, one for each acknowledgment mode. MessageProducers and MessageConsumers are cached within their owning session and also take into account the unique properties of the producers and consumers when caching. MessageProducers are cached based on their destination. MessageConsumers are cached based on a key composed of the destination, selector, noLocal delivery flag, and the durable subscription name (if creating durable consumers).

3.2.3. Destination Management

Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI. When configuring a Spring application context you can use the JNDI factory class `JndiObjectFactoryBean` / `<jee:jndi-lookup>` to perform dependency injection on your object's references to JMS destinations. However, often this strategy is cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the JMS provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The `JmsTemplate` delegates the resolution of a destination name to a JMS destination object to an implementation of the interface `DestinationResolver`. `DynamicDestinationResolver` is the default

implementation used by `JmsTemplate` and accommodates resolving dynamic destinations. A `JndiDestinationResolver` is also provided that acts as a service locator for destinations contained in JNDI and optionally falls back to the behavior contained in `DynamicDestinationResolver`.

Quite often the destinations used in a JMS application are only known at runtime and therefore cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations is not part of the JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in JNDI. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to disregard the warnings in the JMS specification and to use the `TopicSession` method `createTopic(String topicName)` or the `QueueSession` method `createQueue(String queueName)` to create a new destination with default destination properties. Depending on the vendor implementation, `DynamicDestinationResolver` may then also create a physical destination instead of only resolving one.

The boolean property `pubSubDomain` is used to configure the `JmsTemplate` with knowledge of what JMS domain is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used. This property used by `JmsTemplate` determines the behavior of dynamic destination resolution via implementations of the `DestinationResolver` interface.

You can also configure the `JmsTemplate` with a default destination via the property `defaultDestination`. The default destination will be used with send and receive operations that do not refer to a specific destination.

3.2.4. Message Listener Containers

One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (MDBs). Spring offers a solution to create message-driven POJOs (MDPs) in a way that does not tie a user to an EJB container. (See Asynchronous reception: Message-Driven POJOs for detailed coverage of Spring's MDP support.) As from Spring Framework 4.1, endpoint methods can be simply annotated using `@JmsListener` see Annotation-driven listener endpoints for more details.

A message listener container is used to receive messages from a JMS message queue and drive the `MessageListener` that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegates boilerplate JMS infrastructure concerns to the framework.

There are two standard JMS message listener containers packaged with Spring, each with its specialised feature set.

SimpleMessageListenerContainer

This message listener container is the simpler of the two standard flavors. It creates a fixed number of JMS sessions and consumers at startup, registers the listener using the standard JMS `MessageConsumer.setMessageListener()` method, and leaves it up the JMS provider to perform listener callbacks. This variant does not allow for dynamic adaption to runtime demands or for participation in externally managed transactions. Compatibility-wise, it stays very close to the spirit of the standalone JMS specification - but is generally not compatible with Java EE's JMS restrictions.

While `SimpleMessageListenerContainer` does not allow for the participation in externally managed transactions, it does support native JMS transactions: simply switch the 'sessionTransacted' flag to 'true' or, in the namespace, set the 'acknowledge' attribute to 'transacted': Exceptions thrown from your listener will lead to a rollback then, with the message getting redelivered. Alternatively, consider using 'CLIENT_ACKNOWLEDGE' mode which provides redelivery in case of an exception as well but does not use transacted Sessions and therefore does not include any other Session operations (such as sending response messages) in the transaction protocol.

The default 'AUTO_ACKNOWLEDGE' mode does not provide proper reliability guarantees. Messages may get lost when listener execution fails (since the provider will automatically acknowledge each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (this may be configured through the 'acceptMessagesWhileStopping' flag). Make sure to use transacted sessions in case of reliability needs, e.g. for reliable queue handling and durable topic subscriptions.

DefaultMessageListenerContainer

This message listener container is the one used in most cases. In contrast to `SimpleMessageListenerContainer`, this container variant allows for dynamic adaptation to runtime demands and is able to participate in externally managed transactions. Each received message is registered with an XA transaction when configured with a `JtaTransactionManager`; so processing may take advantage of XA transaction semantics. This listener container strikes a good balance between low requirements on the JMS provider, advanced functionality such as the participation in externally managed transactions, and compatibility with Java EE environments.

The cache level of the container can be customized. Note that when no caching is enabled, a new connection and a new session is created for each message reception. Combining this with a non durable subscription with high loads may lead to message lost. Make sure to use a proper cache level in such case.

This container also has recoverable capabilities when the broker goes down. By default, a simple `BackOff` implementation retries every 5 seconds. It is possible to specify a custom `BackOff` implementation for more fine-grained recovery options, see `ExponentialBackOff` for an example.

Like its sibling `SimpleMessageListenerContainer`, `DefaultMessageListenerContainer` supports native JMS transactions and also allows for customizing the acknowledgment mode. This is strongly recommended over externally managed transactions if feasible for your scenario: that is, if you can live with occasional duplicate messages in case of the JVM dying. Custom duplicate message detection steps in your business logic may cover such situations, e.g. in the form of a business entity existence check or a protocol table check. Any such arrangements will be significantly more efficient than the alternative: wrapping your entire processing with an XA transaction (through configuring your `DefaultMessageListenerContainer` with an `JtaTransactionManager`), covering the reception of the JMS message as well as the execution of the business logic in your message listener (including database operations etc).

The default 'AUTO_ACKNOWLEDGE' mode does not provide proper reliability guarantees. Messages may get lost when listener execution fails (since the provider will automatically acknowledge each message before listener invocation) or when the listener container shuts down (this may be configured through the 'acceptMessagesWhileStopping' flag). Make sure to use transacted sessions in case of reliability needs, e.g. for reliable queue handling and durable topic subscriptions.

3.2.5. Transaction management

Spring provides a `JmsTransactionManager` that manages transactions for a single JMS `ConnectionFactory`. This allows JMS applications to leverage the managed transaction features of Spring as described in Transaction Management. The `JmsTransactionManager` performs local resource transactions, binding a JMS Connection/Session pair from the specified `ConnectionFactory` to the thread. `JmsTemplate` automatically detects such transactional resources and operates on them accordingly.

In a Java EE environment, the `ConnectionFactory` will pool Connections and Sessions, so those resources are efficiently reused across transactions. In a standalone environment, using Spring's `SingleConnectionFactory` will result in a shared JMS `Connection`, with each transaction having its own independent `Session`. Alternatively, consider the use of a provider-specific pooling adapter such as ActiveMQ's `PooledConnectionFactory` class.

`JmsTemplate` can also be used with the `JtaTransactionManager` and an XA-capable JMS `ConnectionFactory` for performing distributed transactions. Note that this requires the use of a JTA transaction manager as well as a properly XA-configured ConnectionFactory! (Check your Java EE server's / JMS provider's documentation.)

Reusing code across a managed and unmanaged transactional environment can be confusing when using the JMS API to create a `Session` from a `Connection`. This is because the JMS API has only one factory method to create a `Session` and it requires values for the transaction and acknowledgment modes. In a managed environment, setting these values is the responsibility of the environment's transactional infrastructure, so these values are ignored by the vendor's wrapper to the JMS Connection. When using the `JmsTemplate` in an unmanaged environment you can specify these values through the use of the properties `sessionTransacted` and `sessionAcknowledgeMode`. When using

a `PlatformTransactionManager` with `JmsTemplate`, the template will always be given a transactional JMS `Session`.

3.3. Sending a Message

The `JmsTemplate` contains many convenience methods to send a message. There are send methods that specify the destination using a `javax.jms.Destination` object and those that specify the destination using a string for use in a JNDI lookup. The send method that takes no destination argument uses the default destination.

```java
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

    private JmsTemplate jmsTemplate;
    private Queue queue;

    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    public void setQueue(Queue queue) {
        this.queue = queue;
    }

    public void simpleSend() {
        this.jmsTemplate.send(this.queue, new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage("hello queue world");
            }
        });
    }
}
```

This example uses the `MessageCreator` callback to create a text message from the supplied `Session` object. The `JmsTemplate` is constructed by passing a reference to a `ConnectionFactory`. As an alternative, a zero argument constructor and `connectionFactory` is provided and can be used for constructing the instance in JavaBean style (using a BeanFactory or plain Java code). Alternatively, consider deriving from Spring's `JmsGatewaySupport` convenience base class, which provides pre-built bean properties for JMS configuration.

The method `send(String destinationName, MessageCreator creator)` lets you send a message using

the string name of the destination. If these names are registered in JNDI, you should set the `destinationResolver` property of the template to an instance of `JndiDestinationResolver`.

If you created the `JmsTemplate` and specified a default destination, the `send(MessageCreator c)` sends a message to that destination.

3.3.1. Using Message Converters

In order to facilitate the sending of domain model objects, the `JmsTemplate` has various send methods that take a Java object as an argument for a message's data content. The overloaded methods `convertAndSend()` and `receiveAndConvert()` in `JmsTemplate` delegate the conversion process to an instance of the `MessageConverter` interface. This interface defines a simple contract to convert between Java objects and JMS messages. The default implementation `SimpleMessageConverter` supports conversion between `String` and `TextMessage`, `byte[]` and `BytesMesssage`, and `java.util.Map` and `MapMessage`. By using the converter, you and your application code can focus on the business object that is being sent or received via JMS and not be concerned with the details of how it is represented as a JMS message.

The sandbox currently includes a `MapMessageConverter` which uses reflection to convert between a JavaBean and a `MapMessage`. Other popular implementation choices you might implement yourself are Converters that use an existing XML marshalling package, such as JAXB, Castor or XStream, to create a `TextMessage` representing the object.

To accommodate the setting of a message's properties, headers, and body that can not be generically encapsulated inside a converter class, the `MessagePostProcessor` interface gives you access to the message after it has been converted, but before it is sent. The example below demonstrates how to modify a message header and a property after a `java.util.Map` is converted to a message.

```java
public void sendWithConversion() {
    Map map = new HashMap();
    map.put("Name", "Mark");
    map.put("Age", new Integer(47));
    jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {
        public Message postProcessMessage(Message message) throws JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");
            return message;
        }
    });
}
```

This results in a message of the form:

```
MapMessage={
    Header={
        ... standard headers ...
        CorrelationID={123-00001}
    }
    Properties={
        AccountID={Integer:1234}
    }
    Fields={
        Name={String:Mark}
        Age={Integer:47}
    }
}
```

3.3.2. SessionCallback and ProducerCallback

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS `Session` or `MessageProducer`. The `SessionCallback` and `ProducerCallback` expose the JMS `Session` and `Session` / `MessageProducer` pair respectively. The `execute()` methods on `JmsTemplate` execute these callback methods.

3.4. Receiving a message

3.4.1. Synchronous reception

While JMS is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded `receive(..)` methods provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property `receiveTimeout` specifies how long the receiver should wait before giving up waiting for a message.

3.4.2. Asynchronous reception: Message-Driven POJOs

> ℹ️ Spring also supports annotated-listener endpoints through the use of the `@JmsListener` annotation and provides an open infrastructure to register endpoints programmatically. This is by far the most convenient way to setup an asynchronous receiver, see Enable listener endpoint annotations for more details.

In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages. The one restriction (but see also below for the discussion of the `MessageListenerAdapter` class) on an MDP is that it must implement the `javax.jms.MessageListener` interface. Please also be aware that in the case where your POJO will be receiving messages on multiple threads, it is important to ensure that your implementation is thread-safe.

Below is a simple implementation of an MDP:

```java
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                System.out.println(((TextMessage) message).getText());
            }
            catch (JMSException ex) {
                throw new RuntimeException(ex);
            }
        }
        else {
            throw new IllegalArgumentException("Message must be of type TextMessage");
        }
    }
}
```

Once you've implemented your `MessageListener`, it's time to create a message listener container.

Find below an example of how to define and configure one of the message listener containers that ships with Spring (in this case the `DefaultMessageListenerContainer`).

```xml
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the message listener container -->
<bean id="jmsContainer" class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <strong><property name="messageListener" ref="messageListener" /></strong>
</bean>
```

Please refer to the Spring javadocs of the various message listener containers for a full description of the features supported by each implementation.

3.4.3. SessionAwareMessageListener interface

The `SessionAwareMessageListener` interface is a Spring-specific interface that provides a similar contract to the JMS `MessageListener` interface, but also provides the message handling method with access to the JMS `Session` from which the `Message` was received.

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

You can choose to have your MDPs implement this interface (in preference to the standard JMS `MessageListener` interface) if you want your MDPs to be able to respond to any received messages (using the `Session` supplied in the `onMessage(Message, Session)` method). All of the message listener container implementations that ship with Spring have support for MDPs that implement either the `MessageListener` or `SessionAwareMessageListener` interface. Classes that implement the `SessionAwareMessageListener` come with the caveat that they are then tied to Spring through the interface. The choice of whether or not to use it is left entirely up to you as an application developer or architect.

Please note that the `'onMessage(..)'` method of the `SessionAwareMessageListener` interface throws `JMSException`. In contrast to the standard JMS `MessageListener` interface, when using the `SessionAwareMessageListener` interface, it is the responsibility of the client code to handle any exceptions thrown.

3.4.4. MessageListenerAdapter

The `MessageListenerAdapter` class is the final component in Spring's asynchronous messaging support: in a nutshell, it allows you to expose almost *any* class as a MDP (there are of course some constraints).

Consider the following interface definition. Notice that although the interface extends neither the `MessageListener` nor `SessionAwareMessageListener` interfaces, it can still be used as a MDP via the use of the `MessageListenerAdapter` class. Notice also how the various message handling methods are strongly typed according to the *contents* of the various `Message` types that they can receive and handle.

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

```java
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

In particular, note how the above implementation of the `MessageDelegate` interface (the above `DefaultMessageDelegate` class) has *no* JMS dependencies at all. It truly is a POJO that we will make into an MDP via the following configuration.

```xml
<!-- this is the Message Driven POJO (MDP) -->
<strong><bean id="messageListener" class=
"org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean></strong>

<!-- and this is the message listener container... -->
<bean id="jmsContainer" class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <strong><property name="messageListener" ref="messageListener" /></strong>
</bean>
```

Below is an example of another MDP that can only handle the receiving of JMS `TextMessage` messages. Notice how the message handling method is actually called `'receive'` (the name of the message handling method in a `MessageListenerAdapter` defaults to `'handleMessage'`), but it is configurable (as you will see below). Notice also how the `'receive(..)'` method is strongly typed to receive and respond only to JMS `TextMessage` messages.

```java
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

```java
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

The configuration of the attendant `MessageListenerAdapter` would look like this:

```
<bean id="messageListener" class=
"org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

Please note that if the above `'messageListener'` receives a JMS `Message` of a type other than `TextMessage`, an `IllegalStateException` will be thrown (and subsequently swallowed). Another of the capabilities of the `MessageListenerAdapter` class is the ability to automatically send back a response `Message` if a handler method returns a non-void value. Consider the interface and class:

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements
ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

If the above `DefaultResponsiveTextMessageDelegate` is used in conjunction with a `MessageListenerAdapter` then any non-null value that is returned from the execution of the `'receive(..)'` method will (in the default configuration) be converted into a `TextMessage`. The resulting `TextMessage` will then be sent to the `Destination` (if one exists) defined in the JMS Reply-To property of the original `Message`, or the default `Destination` set on the `MessageListenerAdapter` (if one has been configured); if no `Destination` is found then an `InvalidDestinationException` will be thrown (and please note that this exception *will not* be swallowed and *will* propagate up the call stack).

3.4.5. Processing messages within transactions

Invoking a message listener within a transaction only requires reconfiguration of the listener container.

Local resource transactions can simply be activated through the `sessionTransacted` flag on the listener container definition. Each message listener invocation will then operate within an active JMS transaction, with message reception rolled back in case of listener execution failure. Sending a response message (via `SessionAwareMessageListener`) will be part of the same local transaction, but

any other resource operations (such as database access) will operate independently. This usually requires duplicate message detection in the listener implementation, covering the case where database processing has committed but message processing failed to commit.

```xml
<bean id="jmsContainer" class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <strong><property name="sessionTransacted" value="true"/></strong>
</bean>
```

For participating in an externally managed transaction, you will need to configure a transaction manager and use a listener container which supports externally managed transactions: typically `DefaultMessageListenerContainer`.

To configure a message listener container for XA transaction participation, you'll want to configure a `JtaTransactionManager` (which, by default, delegates to the Java EE server's transaction subsystem). Note that the underlying JMS ConnectionFactory needs to be XA-capable and properly registered with your JTA transaction coordinator! (Check your Java EE server's configuration of JNDI resources.) This allows message reception as well as e.g. database access to be part of the same transaction (with unified commit semantics, at the expense of XA transaction log overhead).

```xml
<bean id="transactionManager" class=
"org.springframework.transaction.jta.JtaTransactionManager"/>
```

Then you just need to add it to our earlier container configuration. The container will take care of the rest.

```xml
<bean id="jmsContainer" class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <strong><property name="transactionManager" ref="transactionManager"/></strong>
</bean>
```

3.5. Support for JCA Message Endpoints

Beginning with version 2.5, Spring also provides support for a JCA-based `MessageListener` container. The `JmsMessageEndpointManager` will attempt to automatically determine the `ActivationSpec` class name from the provider's `ResourceAdapter` class name. Therefore, it is typically possible to just provide Spring's generic `JmsActivationSpecConfig` as shown in the following example.

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpecConfig">
        <bean class="
org.springframework.jms.listener.endpoint.JmsActivationSpecConfig">
            <property name="destinationName" value="myQueue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

Alternatively, you may set up a `JmsMessageEndpointManager` with a given `ActivationSpec` object. The `ActivationSpec` object may also come from a JNDI lookup (using `<jee:jndi-lookup>`).

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpec">
        <bean class="org.apache.activemq.ra.ActiveMQActivationSpec">
            <property name="destination" value="myQueue"/>
            <property name="destinationType" value="javax.jms.Queue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

Using Spring's `ResourceAdapterFactoryBean`, the target `ResourceAdapter` may be configured locally as depicted in the following example.

```
<bean id="resourceAdapter" class=
"org.springframework.jca.support.ResourceAdapterFactoryBean">
    <property name="resourceAdapter">
        <bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl" value="tcp://localhost:61616"/>
        </bean>
    </property>
    <property name="workManager">
        <bean class="org.springframework.jca.work.SimpleTaskWorkManager"/>
    </property>
</bean>
```

The specified `WorkManager` may also point to an environment-specific thread pool - typically through `SimpleTaskWorkManager`'s "asyncTaskExecutor" property. Consider defining a shared thread pool for all your `ResourceAdapter` instances if you happen to use multiple adapters.

In some environments (e.g. WebLogic 9 or above), the entire `ResourceAdapter` object may be obtained from JNDI instead (using `<jee:jndi-lookup>`). The Spring-based message listeners can then interact with the server-hosted `ResourceAdapter`, also using the server's built-in `WorkManager`.

Please consult the javadoc for `JmsMessageEndpointManager`, `JmsActivationSpecConfig`, and `ResourceAdapterFactoryBean` for more details.

Spring also provides a generic JCA message endpoint manager which is not tied to JMS: `org.springframework.jca.endpoint.GenericMessageEndpointManager`. This component allows for using any message listener type (e.g. a CCI MessageListener) and any provider-specific ActivationSpec object. Check out your JCA provider's documentation to find out about the actual capabilities of your connector, and consult `GenericMessageEndpointManager`'s javadoc for the Spring-specific configuration details.

> JCA-based message endpoint management is very analogous to EJB 2.1 Message-Driven Beans; it uses the same underlying resource provider contract. Like with EJB 2.1 MDBs, any message listener interface supported by your JCA provider can be used in the Spring context as well. Spring nevertheless provides explicit 'convenience' support for JMS, simply because JMS is the most common endpoint API used with the JCA endpoint management contract.

3.6. Annotation-driven listener endpoints

The easiest way to receive a message asynchronously is to use the annotated listener endpoint infrastructure. In a nutshell, it allows you to expose a method of a managed bean as a JMS listener endpoint.

```java
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(String data) { ... }
}
```

The idea of the example above is that whenever a message is available on the `javax.jms.Destination` "myDestination", the `processOrder` method is invoked accordingly (in this case, with the content of the JMS message similarly to what the `MessageListenerAdapter` provides).

The annotated endpoint infrastructure creates a message listener container behind the scenes for each annotated method, using a `JmsListenerContainerFactory`. Such a container is not registered against the application context but can be easily located for management purposes using the `JmsListenerEndpointRegistry` bean.

> `@JmsListener` is a *repeatable* annotation on Java 8, so it is possible to associate several JMS destinations to the same method by adding additional `@JmsListener` declarations to it.

3.6.1. Enable listener endpoint annotations

To enable support for `@JmsListener` annotations add `@EnableJms` to one of your `@Configuration` classes.

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        factory.setDestinationResolver(destinationResolver());
        factory.setSessionTransacted(true);
        factory.setConcurrency("3-10");
        return factory;
    }
}
```

By default, the infrastructure looks for a bean named `jmsListenerContainerFactory` as the source for the factory to use to create message listener containers. In this case, and ignoring the JMS infrastructure setup, the `processOrder` method can be invoked with a core poll size of 3 threads and a maximum pool size of 10 threads.

It is possible to customize the listener container factory to use per annotation or an explicit default can be configured by implementing the `JmsListenerConfigurer` interface. The default is only required if at least one endpoint is registered without a specific container factory. See the javadoc for full details and examples.

If you prefer XML configuration use the `<jms:annotation-driven>` element.

```
<jms:annotation-driven/>

<bean id="jmsListenerContainerFactory"
        class="org.springframework.jms.config.DefaultJmsListenerContainerFactory">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destinationResolver" ref="destinationResolver"/>
    <property name="sessionTransacted" value="true"/>
    <property name="concurrency" value="3-10"/>
</bean>
```

3.6.2. Programmatic endpoints registration

`JmsListenerEndpoint` provides a model of an JMS endpoint and is responsible for configuring the container for that model. The infrastructure allows you to configure endpoints programmatically in addition to the ones that are detected by the `JmsListener` annotation.

```java
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
        SimpleJmsListenerEndpoint endpoint = new SimpleJmsListenerEndpoint();
        endpoint.setId("myJmsEndpoint");
        endpoint.setDestination("anotherQueue");
        endpoint.setMessageListener(message -> {
            // processing
        });
        registrar.registerEndpoint(endpoint);
    }
}
```

In the example above, we used `SimpleJmsListenerEndpoint` which provides the actual `MessageListener` to invoke but you could just as well build your own endpoint variant describing a custom invocation mechanism.

It should be noted that you could just as well skip the use of `@JmsListener` altogether and only register your endpoints programmatically through `JmsListenerConfigurer`.

3.6.3. Annotated endpoint method signature

So far, we have been injecting a simple `String` in our endpoint but it can actually have a very flexible method signature. Let's rewrite it to inject the `Order` with a custom header:

```java
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(Order order, @Header("order_type") String orderType) {
        ...
    }
}
```

These are the main elements you can inject in JMS listener endpoints:

- The raw `javax.jms.Message` or any of its subclasses (provided of course that it matches the incoming message type).

- The `javax.jms.Session` for optional access to the native JMS API e.g. for sending a custom reply.

- The `org.springframework.messaging.Message` representing the incoming JMS message. Note that this message holds both the custom and the standard headers (as defined by `JmsHeaders`).

- `@Header`-annotated method arguments to extract a specific header value, including standard JMS headers.

- `@Headers`-annotated argument that must also be assignable to `java.util.Map` for getting access to all headers.
- A non-annotated element that is not one of the supported types (i.e. `Message` and `Session`) is considered to be the payload. You can make that explicit by annotating the parameter with `@Payload`. You can also turn on validation by adding an extra `@Valid`.

The ability to inject Spring's `Message` abstraction is particularly useful to benefit from all the information stored in the transport-specific message without relying on transport-specific API.

```
@JmsListener(destination = "myDestination")
public void processOrder(Message<Order> order) { ... }
```

Handling of method arguments is provided by `DefaultMessageHandlerMethodFactory` which can be further customized to support additional method arguments. The conversion and validation support can be customized there as well.

For instance, if we want to make sure our `Order` is valid before processing it, we can annotate the payload with `@Valid` and configure the necessary validator as follows:

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
        registrar.setMessageHandlerMethodFactory(myJmsHandlerMethodFactory());
    }

    @Bean
    public DefaultMessageHandlerMethodFactory myHandlerMethodFactory() {
        DefaultMessageHandlerMethodFactory factory = new
DefaultMessageHandlerMethodFactory();
        factory.setValidator(myValidator());
        return factory;
    }
}
```

3.6.4. Response management

The existing support in MessageListenerAdapter already allows your method to have a non-`void` return type. When that's the case, the result of the invocation is encapsulated in a `javax.jms.Message` sent either in the destination specified in the `JMSReplyTo` header of the original message or in the default destination configured on the listener. That default destination can now be set using the `@SendTo` annotation of the messaging abstraction.

Assuming our `processOrder` method should now return an `OrderStatus`, it is possible to write it as follow to automatically send a response:

```java
@JmsListener(destination = "myDestination")
@SendTo("status")
public OrderStatus processOrder(Order order) {
    // order processing
    return status;
}
```

> 💡 If you have several `@JmsListener`-annotated methods, you can also place the `@SendTo` annotation at the class level to share a default reply destination.

If you need to set additional headers in a transport-independent manner, you could return a `Message` instead, something like:

```java
@JmsListener(destination = "myDestination")
@SendTo("status")
public Message<OrderStatus> processOrder(Order order) {
    // order processing
    return MessageBuilder
            .withPayload(status)
            .setHeader("code", 1234)
            .build();
}
```

If you need to compute the response destination at runtime, you can encapsulate your response in a `JmsResponse` instance that also provides the destination to use at runtime. The previous example can be rewritten as follows:

```java
@JmsListener(destination = "myDestination")
public JmsResponse<Message<OrderStatus>> processOrder(Order order) {
    // order processing
    Message<OrderStatus> response = MessageBuilder
            .withPayload(status)
            .setHeader("code", 1234)
            .build();
    return JmsResponse.forQueue(response, "status");
}
```

Finally if you need to specify some QoS values for the response such as the priority or the time to live, you can configure the `JmsListenerContainerFactory` accordingly:

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        QosSettings replyQosSettings = new ReplyQosSettings();
        replyQosSettings.setPriority(2);
        replyQosSettings.setTimeToLive(10000);
        factory.setReplyQosSettings(replyQosSettings);
        return factory;
    }
}
```

3.7. JMS namespace support

Spring provides an XML namespace for simplifying JMS configuration. To use the JMS namespace elements you will need to reference the JMS schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       <strong>xmlns:jms="http://www.springframework.org/schema/jms"</strong>
       xsi:schemaLocation="
          http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
          <strong>http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms.xsd</strong>">

    <!-- bean definitions here -->

</beans>
```

The namespace consists of three top-level elements: `<annotation-driven/>`, `<listener-container/>` and `<jca-listener-container/>`. `<annotation-driven` enables the use of annotation-driven listener endpoints. `<listener-container/>` and `<jca-listener-container/>` defines shared listener container configuration and may contain `<listener/>` child elements. Here is an example of a basic configuration for two listeners.

```
<jms:listener-container>

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger" method=
"log"/>

</jms:listener-container>
```

The example above is equivalent to creating two distinct listener container bean definitions and two distinct `MessageListenerAdapter` bean definitions as demonstrated in MessageListenerAdapter. In addition to the attributes shown above, the `listener` element may contain several optional ones. The following table describes all available attributes:

Table 2. Attributes of the JMS <listener> element

Attribute	Description
id	A bean name for the hosting listener container. If not specified, a bean name will be automatically generated.
destination *(required)*	The destination name for this listener, resolved through the `DestinationResolver` strategy.
ref *(required)*	The bean name of the handler object.
method	The name of the handler method to invoke. If the `ref` points to a `MessageListener` or Spring `SessionAwareMessageListener`, this attribute may be omitted.
response-destination	The name of the default response destination to send response messages to. This will be applied in case of a request message that does not carry a "JMSReplyTo" field. The type of this destination will be determined by the listener-container's "response-destination-type" attribute. Note: This only applies to a listener method with a return value, for which each result object will be converted into a response message.
subscription	The name of the durable subscription, if any.
selector	An optional message selector for this listener.
concurrency	The number of concurrent sessions/consumers to start for this listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and might be ignored at runtime. Default is the value provided by the container

The `<listener-container/>` element also accepts several optional attributes. This allows for customization of the various strategies (for example, `taskExecutor` and `destinationResolver`) as well as basic JMS settings and resource references. Using these attributes, it is possible to define highly-customized listener containers while still benefiting from the convenience of the namespace.

Such settings can be automatically exposed as a `JmsListenerContainerFactory` by specifying the id of the bean to expose through the `factory-id` attribute.

```
<jms:listener-container connection-factory="myConnectionFactory"
        task-executor="myTaskExecutor"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger" method=
"log"/>

</jms:listener-container>
```

The following table describes all available attributes. Consult the class-level javadocs of the `AbstractMessageListenerContainer` and its concrete subclasses for more details on the individual properties. The javadocs also provide a discussion of transaction choices and message redelivery scenarios.

Table 3. Attributes of the JMS <listener-container> element

Attribute	Description
container-type	The type of this listener container. Available options are: `default`, `simple`, `default102`, or `simple102` (the default value is `'default'`).
container-class	A custom listener container implementation class as fully qualified class name. Default is Spring's standard `DefaultMessageListenerContainer` or `SimpleMessageListenerContainer`, according to the "container-type" attribute.
factory-id	Exposes the settings defined by this element as a `JmsListenerContainerFactory` with the specified id so that they can be reused with other endpoints.
connection-factory	A reference to the JMS `ConnectionFactory` bean (the default bean name is `'connectionFactory'`).
task-executor	A reference to the Spring `TaskExecutor` for the JMS listener invokers.
destination-resolver	A reference to the `DestinationResolver` strategy for resolving JMS `Destinations`.
message-converter	A reference to the `MessageConverter` strategy for converting JMS Messages to listener method arguments. Default is a `SimpleMessageConverter`.
error-handler	A reference to an `ErrorHandler` strategy for handling any uncaught Exceptions that may occur during the execution of the `MessageListener`.
destination-type	The JMS destination type for this listener: `queue`, `topic`, `durableTopic`, `sharedTopic` or `sharedDurableTopic`. This enables potentially the `pubSubDomain`, `subscriptionDurable` and `subscriptionShared` properties of the container. The default is `queue` (i.e. disabling those 3 properties).
response-destination-type	The JMS destination type for responses: "queue", "topic". Default is the value of the "destination-type" attribute.

Attribute	Description
client-id	The JMS client id for this listener container. Needs to be specified when using durable subscriptions.
cache	The cache level for JMS resources: `none`, `connection`, `session`, `consumer` or `auto`. By default (`auto`), the cache level will effectively be "consumer", unless an external transaction manager has been specified - in which case the effective default will be `none` (assuming Java EE-style transaction management where the given ConnectionFactory is an XA-aware pool).
acknowledge	The native JMS acknowledge mode: `auto`, `client`, `dups-ok` or `transacted`. A value of `transacted` activates a locally transacted `Session`. As an alternative, specify the `transaction-manager` attribute described below. Default is `auto`.
transaction-manager	A reference to an external `PlatformTransactionManager` (typically an XA-based transaction coordinator, e.g. Spring's `JtaTransactionManager`). If not specified, native acknowledging will be used (see "acknowledge" attribute).
concurrency	The number of concurrent sessions/consumers to start for each listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and might be ignored at runtime. Default is 1; keep concurrency limited to 1 in case of a topic listener or if queue ordering is important; consider raising it for general queues.
prefetch	The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers!
receive-timeout	The timeout to use for receive calls (in milliseconds). The default is `1000` ms (1 sec); `-1` indicates no timeout at all.
back-off	Specify the `BackOff` instance to use to compute the interval between recovery attempts. If the `BackOffExecution` implementation returns `BackOffExecution#STOP`, the listener container will not further attempt to recover. The `recovery-interval` value is ignored when this property is set. The default is a `FixedBackOff` with an interval of 5000 ms, that is 5 seconds.
recovery-interval	Specify the interval between recovery attempts, in milliseconds. Convenience way to create a `FixedBackOff` with the specified interval. For more recovery options, consider specifying a BackOff instance instead. The default is 5000 ms, that is 5 seconds.
phase	The lifecycle phase within which this container should start and stop. The lower the value the earlier this container will start and the later it will stop. The default is `Integer.MAX_VALUE` meaning the container will start as late as possible and stop as soon as possible.

Configuring a JCA-based listener container with the "jms" schema support is very similar.

```
<jms:jca-listener-container resource-adapter="myResourceAdapter"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="myMessageListener"/>

</jms:jca-listener-container>
```

The available configuration options for the JCA variant are described in the following table:

Table 4. Attributes of the JMS <jca-listener-container/> element

Attribute	Description
factory-id	Exposes the settings defined by this element as a `JmsListenerContainerFactory` with the specified id so that they can be reused with other endpoints.
resource-adapter	A reference to the JCA `ResourceAdapter` bean (the default bean name is `'resourceAdapter'`).
activation-spec-factory	A reference to the `JmsActivationSpecFactory`. The default is to autodetect the JMS provider and its `ActivationSpec` class (see `DefaultJmsActivationSpecFactory`)
destination-resolver	A reference to the `DestinationResolver` strategy for resolving JMS `Destinations`.
message-converter	A reference to the `MessageConverter` strategy for converting JMS Messages to listener method arguments. Default is a `SimpleMessageConverter`.
destination-type	The JMS destination type for this listener: `queue`, `topic`, `durableTopic`, `sharedTopic` or `sharedDurableTopic`. This enables potentially the `pubSubDomain`, `subscriptionDurable` and `subscriptionShared` properties of the container. The default is `queue` (i.e. disabling those 3 properties).
response-destination-type	The JMS destination type for responses: "queue", "topic". Default is the value of the "destination-type" attribute.
client-id	The JMS client id for this listener container. Needs to be specified when using durable subscriptions.
acknowledge	The native JMS acknowledge mode: `auto`, `client`, `dups-ok` or `transacted`. A value of `transacted` activates a locally transacted `Session`. As an alternative, specify the `transaction-manager` attribute described below. Default is `auto`.
transaction-manager	A reference to a Spring `JtaTransactionManager` or a `javax.transaction.TransactionManager` for kicking off an XA transaction for each incoming message. If not specified, native acknowledging will be used (see the "acknowledge" attribute).
concurrency	The number of concurrent sessions/consumers to start for each listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and will typically be ignored at runtime when using a JCA listener container. Default is 1.

Attribute	Description
prefetch	The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers!

Chapter 4. JMX

4.1. Introduction

The JMX support in Spring provides you with the features to easily and transparently integrate your Spring application into a JMX infrastructure.

> **JMX?**
>
> This chapter is not an introduction to JMX… it doesn't try to explain the motivations of why one might want to use JMX (or indeed what the letters JMX actually stand for). If you are new to JMX, check out Further resources at the end of this chapter.

Specifically, Spring's JMX support provides four core features:

- The automatic registration of *any* Spring bean as a JMX MBean

- A flexible mechanism for controlling the management interface of your beans

- The declarative exposure of MBeans over remote, JSR-160 connectors

- The simple proxying of both local and remote MBean resources

These features are designed to work without coupling your application components to either Spring or JMX interfaces and classes. Indeed, for the most part your application classes need not be aware of either Spring or JMX in order to take advantage of the Spring JMX features.

4.2. Exporting your beans to JMX

The core class in Spring's JMX framework is the `MBeanExporter`. This class is responsible for taking your Spring beans and registering them with a JMX `MBeanServer`. For example, consider the following class:

```java
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

To expose the properties and methods of this bean as attributes and operations of an MBean you simply configure an instance of the `MBeanExporter` class in your configuration file and pass in the bean as shown below:

```xml
<beans>
    <!-- this bean must not be lazily initialized if the exporting is to happen -->
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-
init="false">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
    </bean>
    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

The pertinent bean definition from the above configuration snippet is the `exporter` bean. The `beans` property tells the `MBeanExporter` exactly which of your beans must be exported to the JMX `MBeanServer`. In the default configuration, the key of each entry in the `beans Map` is used as the `ObjectName` for the bean referenced by the corresponding entry value. This behavior can be changed as described in Controlling the ObjectNames for your beans.

With this configuration the `testBean` bean is exposed as an MBean under the `ObjectName` `bean:name=testBean1`. By default, all *public* properties of the bean are exposed as attributes and all *public* methods (bar those inherited from the `Object` class) are exposed as operations.

> ℹ️ `MBeanExporter` is a `Lifecycle` bean (see Startup and shutdown callbacks) and MBeans are exported as late as possible during the application lifecycle by default. It is possible to configure the `phase` at which the export happens or disable automatic registration by setting the `autoStartup` flag.

4.2.1. Creating an MBeanServer

The above configuration assumes that the application is running in an environment that has one (and only one) `MBeanServer` already running. In this case, Spring will attempt to locate the running `MBeanServer` and register your beans with that server (if any). This behavior is useful when your application is running inside a container such as Tomcat or IBM WebSphere that has its own `MBeanServer`.

However, this approach is of no use in a standalone environment, or when running inside a container that does not provide an `MBeanServer`. To address this you can create an `MBeanServer` instance declaratively by adding an instance of the `org.springframework.jmx.support.MBeanServerFactoryBean` class to your configuration. You can also ensure that a specific `MBeanServer` is used by setting the value of the `MBeanExporter`'s `server` property to the `MBeanServer` value returned by an `MBeanServerFactoryBean`; for example:

```xml
<beans>

    <bean id="mbeanServer" class=
"org.springframework.jmx.support.MBeanServerFactoryBean"/>

    <!--
    this bean needs to be eagerly pre-instantiated in order for the exporting to
occur;
    this means that it must not be marked as lazily initialized
    -->
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="server" ref="mbeanServer"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

Here an instance of `MBeanServer` is created by the `MBeanServerFactoryBean` and is supplied to the `MBeanExporter` via the server property. When you supply your own `MBeanServer` instance, the `MBeanExporter` will not attempt to locate a running `MBeanServer` and will use the supplied `MBeanServer` instance. For this to work correctly, you must (of course) have a JMX implementation on your classpath.

4.2.2. Reusing an existing MBeanServer

If no server is specified, the `MBeanExporter` tries to automatically detect a running `MBeanServer`. This works in most environment where only one `MBeanServer` instance is used, however when multiple instances exist, the exporter might pick the wrong server. In such cases, one should use the `MBeanServer agentId` to indicate which instance to be used:

```
<beans>
    <bean id="mbeanServer" class=
"org.springframework.jmx.support.MBeanServerFactoryBean">
        <!-- indicate to first look for a server -->
        <property name="locateExistingServerIfPossible" value="true"/>
        <!-- search for the MBeanServer instance with the given agentId -->
        <property name="agentId" value="MBeanServer_instance_agentId>"/>
    </bean>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server" ref="mbeanServer"/>
        ...
    </bean>
</beans>
```

For platforms/cases where the existing `MBeanServer` has a dynamic (or unknown) `agentId` which is retrieved through lookup methods, one should use factory-method:

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server">
            <!-- Custom MBeanServerLocator -->
            <bean class="platform.package.MBeanServerLocator" factory-method=
"locateMBeanServer"/>
        </property>
    </bean>

    <!-- other beans here -->

</beans>
```

4.2.3. Lazy-initialized MBeans

If you configure a bean with the `MBeanExporter` that is also configured for lazy initialization, then the `MBeanExporter` will *not* break this contract and will avoid instantiating the bean. Instead, it will register a proxy with the `MBeanServer` and will defer obtaining the bean from the container until the first invocation on the proxy occurs.

4.2.4. Automatic registration of MBeans

Any beans that are exported through the `MBeanExporter` and are already valid MBeans are registered as-is with the `MBeanServer` without further intervention from Spring. MBeans can be automatically detected by the `MBeanExporter` by setting the `autodetect` property to `true`:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean
"/>
```

Here, the bean called `spring:mbean=true` is already a valid JMX MBean and will be automatically registered by Spring. By default, beans that are autodetected for JMX registration have their bean name used as the `ObjectName`. This behavior can be overridden as detailed in Controlling the ObjectNames for your beans.

4.2.5. Controlling the registration behavior

Consider the scenario where a Spring `MBeanExporter` attempts to register an `MBean` with an `MBeanServer` using the `ObjectName` `'bean:name=testBean1'`. If an `MBean` instance has already been registered under that same `ObjectName`, the default behavior is to fail (and throw an `InstanceAlreadyExistsException`).

It is possible to control the behavior of exactly what happens when an `MBean` is registered with an `MBeanServer`. Spring's JMX support allows for three different registration behaviors to control the registration behavior when the registration process finds that an `MBean` has already been registered under the same `ObjectName`; these registration behaviors are summarized on the following table:

Table 5. Registration Behaviors

Registration behavior	Explanation
`REGISTRATION_FAIL_ON_EXISTING`	This is the default registration behavior. If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered will not be registered and an `InstanceAlreadyExistsException` will be thrown. The existing `MBean` is unaffected.
`REGISTRATION_IGNORE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered will *not* be registered. The existing `MBean` is unaffected, and no `Exception` will be thrown. This is useful in settings where multiple applications want to share a common `MBean` in a shared `MBeanServer`.
`REGISTRATION_REPLACE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the existing `MBean` that was previously registered will be unregistered and the new `MBean` will be registered in its place (the new `MBean` effectively replaces the previous instance).

The above values are defined as constants on the `MBeanRegistrationSupport` class (the `MBeanExporter` class derives from this superclass). If you want to change the default registration behavior, you simply need to set the value of the `registrationBehaviorName` property on your `MBeanExporter` definition to one of those values.

The following example illustrates how to effect a change from the default registration behavior to the `REGISTRATION_REPLACE_EXISTING` behavior:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationBehaviorName" value="
REGISTRATION_REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

4.3. Controlling the management interface of your beans

In the previous example, you had little control over the management interface of your bean; *all* of the *public* properties and methods of each exported bean was exposed as JMX attributes and operations respectively. To exercise finer-grained control over exactly which properties and methods of your exported beans are actually exposed as JMX attributes and operations, Spring JMX provides a comprehensive and extensible mechanism for controlling the management interfaces of your beans.

4.3.1. MBeanInfoAssembler interface

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `org.springframework.jmx.export.assembler.MBeanInfoAssembler` interface which is responsible for defining the management interface of each bean that is being exposed. The default implementation, `org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler`, simply defines a management interface that exposes all public properties and methods (as you saw in the previous examples). Spring provides two additional implementations of the `MBeanInfoAssembler` interface that allow you to control the generated management interface using either source-level metadata or any arbitrary interface.

4.3.2. Using source-level metadata: Java annotations

Using the `MetadataMBeanInfoAssembler` you can define the management interfaces for your beans using source level metadata. The reading of metadata is encapsulated by the `org.springframework.jmx.export.metadata.JmxAttributeSource` interface. Spring JMX provides a default implementation which uses Java annotations, namely `org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource`. The

MetadataMBeanInfoAssembler *must* be configured with an implementation instance of the JmxAttributeSource interface for it to function correctly (there is *no* default).

To mark a bean for export to JMX, you should annotate the bean class with the ManagedResource annotation. Each method you wish to expose as an operation must be marked with the ManagedOperation annotation and each property you wish to expose must be marked with the ManagedAttribute annotation. When marking properties you can omit either the annotation of the getter or the setter to create a write-only or read-only attribute respectively.

> **ℹ** A ManagedResource annotated bean must be public as well as the methods exposing an operation or an attribute.

The example below shows the annotated version of the JmxTestBean class that you saw earlier:

```java
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(
        objectName="bean:name=testBean4",
        description="My Managed Bean",
        log=true,
        logFile="jmx.log",
        currencyTimeLimit=15,
        persistPolicy="OnUpdate",
        persistPeriod=200,
        persistLocation="foo",
        persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

    private String name;
    private int age;

    @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @ManagedAttribute(description="The Name Attribute",
            currencyTimeLimit=20,
            defaultValue="bar",
            persistPolicy="OnUpdate")
    public void setName(String name) {
        this.name = name;
```

```
    }

    @ManagedAttribute(defaultValue="foo", persistPeriod=300)
    public String getName() {
        return name;
    }

    @ManagedOperation(description="Add two numbers")
    @ManagedOperationParameters({
        @ManagedOperationParameter(name = "x", description = "The first number"),
        @ManagedOperationParameter(name = "y", description = "The second number")})
    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

}
```

Here you can see that the `JmxTestBean` class is marked with the `ManagedResource` annotation and that this `ManagedResource` annotation is configured with a set of properties. These properties can be used to configure various aspects of the MBean that is generated by the `MBeanExporter`, and are explained in greater detail later in section entitled Source-level metadata types.

You will also notice that both the `age` and `name` properties are annotated with the `ManagedAttribute` annotation, but in the case of the `age` property, only the getter is marked. This will cause both of these properties to be included in the management interface as attributes, but the `age` attribute will be read-only.

Finally, you will notice that the `add(int, int)` method is marked with the `ManagedOperation` attribute whereas the `dontExposeMe()` method is not. This will cause the management interface to contain only one operation, `add(int, int)`, when using the `MetadataMBeanInfoAssembler`.

The configuration below shows how you configure the `MBeanExporter` to use the `MetadataMBeanInfoAssembler`:

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
            class=
    "org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"
            class="
    org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up the ObjectName from the annotation -->
    <bean id="namingStrategy"
            class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

Here you can see that an `MetadataMBeanInfoAssembler` bean has been configured with an instance of
the `AnnotationJmxAttributeSource` class and passed to the `MBeanExporter` through the assembler
property. This is all that is required to take advantage of metadata-driven management interfaces
for your Spring-exposed MBeans.

4.3.3. Source-level metadata types

The following source level metadata types are available for use in Spring JMX:

Table 6. Source-level metadata types

Purpose	Annotation	Annotation Type
Mark all instances of a `Class` as JMX managed resources	`@ManagedResource`	Class
Mark a method as a JMX operation	`@ManagedOperation`	Method
Mark a getter or setter as one half of a JMX attribute	`@ManagedAttribute`	Method (only getters and setters)

Purpose	Annotation	Annotation Type
Define descriptions for operation parameters	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

The following configuration parameters are available for use on these source-level metadata types:

Table 7. Source-level metadata parameters

Parameter	Description	Applies to
`ObjectName`	Used by `MetadataNamingStrategy` to determine the `ObjectName` of a managed resource	`ManagedResource`
`description`	Sets the friendly description of the resource, attribute or operation	`ManagedResource`, `ManagedAttribute`, `ManagedOperation`, `ManagedOperationParameter`
`currencyTimeLimit`	Sets the value of the `currencyTimeLimit` descriptor field	`ManagedResource`, `ManagedAttribute`
`defaultValue`	Sets the value of the `defaultValue` descriptor field	`ManagedAttribute`
`log`	Sets the value of the `log` descriptor field	`ManagedResource`
`logFile`	Sets the value of the `logFile` descriptor field	`ManagedResource`
`persistPolicy`	Sets the value of the `persistPolicy` descriptor field	`ManagedResource`
`persistPeriod`	Sets the value of the `persistPeriod` descriptor field	`ManagedResource`
`persistLocation`	Sets the value of the `persistLocation` descriptor field	`ManagedResource`
`persistName`	Sets the value of the `persistName` descriptor field	`ManagedResource`
`name`	Sets the display name of an operation parameter	`ManagedOperationParameter`
`index`	Sets the index of an operation parameter	`ManagedOperationParameter`

4.3.4. AutodetectCapableMBeanInfoAssembler interface

To simplify configuration even further, Spring introduces the `AutodetectCapableMBeanInfoAssembler` interface which extends the `MBeanInfoAssembler` interface to add support for autodetection of MBean resources. If you configure the `MBeanExporter` with an instance of `AutodetectCapableMBeanInfoAssembler` then it is allowed to "vote" on the inclusion of beans for exposure to JMX.

Out of the box, the only implementation of the `AutodetectCapableMBeanInfo` interface is the `MetadataMBeanInfoAssembler` which will vote to include any bean which is marked with the `ManagedResource` attribute. The default approach in this case is to use the bean name as the `ObjectName` which results in a configuration like this:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <!-- notice how no 'beans' are explicitly configured here -->
        <property name="autodetect" value="true"/>
        <property name="assembler" ref="assembler"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="assembler" class=
"org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource">
            <bean class=
"org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
        </property>
    </bean>

</beans>
```

Notice that in this configuration no beans are passed to the `MBeanExporter`; however, the `JmxTestBean` will still be registered since it is marked with the `ManagedResource` attribute and the `MetadataMBeanInfoAssembler` detects this and votes to include it. The only problem with this approach is that the name of the `JmxTestBean` now has business meaning. You can address this issue by changing the default behavior for `ObjectName` creation as defined in Controlling the ObjectNames for your beans.

4.3.5. Defining management interfaces using Java interfaces

In addition to the `MetadataMBeanInfoAssembler`, Spring also includes the `InterfaceBasedMBeanInfoAssembler` which allows you to constrain the methods and properties that are exposed based on the set of methods defined in a collection of interfaces.

Although the standard mechanism for exposing MBeans is to use interfaces and a simple naming scheme, the `InterfaceBasedMBeanInfoAssembler` extends this functionality by removing the need for naming conventions, allowing you to use more than one interface and removing the need for your beans to implement the MBean interfaces.

Consider this interface that is used to define a management interface for the `JmxTestBean` class that you saw earlier:

```
public interface IJmxTestBean {

    public int add(int x, int y);

    public long myOperation();

    public int getAge();

    public void setAge(int age);

    public void setName(String name);

    public String getName();

}
```

This interface defines the methods and properties that will be exposed as operations and attributes on the JMX MBean. The code below shows how to configure Spring JMX to use this interface as the definition for the management interface:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean5" value-ref="testBean"/>
            </map>
        </property>
        <property name="assembler">
            <bean class=
"org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
                <property name="managedInterfaces">
                    <value>org.springframework.jmx.IJmxTestBean</value>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

Here you can see that the InterfaceBasedMBeanInfoAssembler is configured to use the IJmxTestBean interface when constructing the management interface for any bean. It is important to understand that beans processed by the InterfaceBasedMBeanInfoAssembler are *not* required to implement the

interface used to generate the JMX management interface.

In the case above, the `IJmxTestBean` interface is used to construct all management interfaces for all beans. In many cases this is not the desired behavior and you may want to use different interfaces for different beans. In this case, you can pass `InterfaceBasedMBeanInfoAssembler` a `Properties` instance via the `interfaceMappings` property, where the key of each entry is the bean name and the value of each entry is a comma-separated list of interface names to use for that bean.

If no management interface is specified through either the `managedInterfaces` or `interfaceMappings` properties, then the `InterfaceBasedMBeanInfoAssembler` will reflect on the bean and use all of the interfaces implemented by that bean to create the management interface.

4.3.6. Using MethodNameBasedMBeanInfoAssembler

The `MethodNameBasedMBeanInfoAssembler` allows you to specify a list of method names that will be exposed to JMX as attributes and operations. The code below shows a sample configuration for this:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=testBean5" value-ref="testBean"/>
        </map>
    </property>
    <property name="assembler">
        <bean class=
"org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
            <property name="managedMethods">
                <value>add,myOperation,getName,setName,getAge</value>
            </property>
        </bean>
    </property>
</bean>
```

Here you can see that the methods `add` and `myOperation` will be exposed as JMX operations and `getName()`, `setName(String)` and `getAge()` will be exposed as the appropriate half of a JMX attribute. In the code above, the method mappings apply to beans that are exposed to JMX. To control method exposure on a bean-by-bean basis, use the `methodMappings` property of `MethodNameMBeanInfoAssembler` to map bean names to lists of method names.

4.4. Controlling the ObjectNames for your beans

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `ObjectNamingStrategy` to obtain `ObjectName`s for each of the beans it is registering. The default implementation, `KeyNamingStrategy`, will, by default, use the key of the `beans` `Map` as the `ObjectName`. In addition, the `KeyNamingStrategy` can map the key of the `beans` `Map` to an entry in a `Properties` file (or files) to resolve the `ObjectName`. In addition to the `KeyNamingStrategy`, Spring provides two additional `ObjectNamingStrategy` implementations: the `IdentityNamingStrategy` that builds an `ObjectName` based on the JVM identity of the bean and the `MetadataNamingStrategy` that uses source level metadata to

obtain the `ObjectName`.

4.4.1. Reading ObjectNames from Properties

You can configure your own `KeyNamingStrategy` instance and configure it to read `ObjectName`s from a `Properties` instance rather than use bean key. The `KeyNamingStrategy` will attempt to locate an entry in the `Properties` with a key corresponding to the bean key. If no entry is found or if the `Properties` instance is `null` then the bean key itself is used.

The code below shows a sample configuration for the `KeyNamingStrategy`:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy" class=
"org.springframework.jmx.export.naming.KeyNamingStrategy">
        <property name="mappings">
            <props>
                <prop key="testBean">bean:name=testBean1</prop>
            </props>
        </property>
        <property name="mappingLocations">
            <value>names1.properties,names2.properties</value>
        </property>
    </bean>

</beans>
```

Here an instance of `KeyNamingStrategy` is configured with a `Properties` instance that is merged from the `Properties` instance defined by the mapping property and the properties files located in the paths defined by the mappings property. In this configuration, the `testBean` bean will be given the `ObjectName bean:name=testBean1` since this is the entry in the `Properties` instance that has a key corresponding to the bean key.

If no entry in the `Properties` instance can be found then the bean key name is used as the `ObjectName`.

4.4.2. Using the MetadataNamingStrategy

The `MetadataNamingStrategy` uses the `objectName` property of the `ManagedResource` attribute on each bean to create the `ObjectName`. The code below shows the configuration for the `MetadataNamingStrategy`:

```xml
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy" class=
"org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="attributeSource"/>
    </bean>

    <bean id="attributeSource"
            class=
"org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</beans>
```

If no `objectName` has been provided for the `ManagedResource` attribute, then an `ObjectName` will be created with the following format:*[fully-qualified-package-name]:type=[short-classname],name=[bean-name]*. For example, the generated `ObjectName` for the following bean would be: *com.foo:type=MyClass,name=myBean*.

```xml
<bean id="myBean" class="com.foo.MyClass"/>
```

4.4.3. Configuring annotation based MBean export

If you prefer using the annotation based approach to define your management interfaces, then a convenience subclass of `MBeanExporter` is available: `AnnotationMBeanExporter`. When defining an instance of this subclass, the `namingStrategy`, `assembler`, and `attributeSource` configuration is no longer needed, since it will always use standard Java annotation-based metadata (autodetection is always enabled as well). In fact, rather than defining an `MBeanExporter` bean, an even simpler syntax is supported by the `@EnableMBeanExport` `@Configuration` annotation.

```
@Configuration
@EnableMBeanExport
public class AppConfig {

}
```

If you prefer XML based configuration the `'context:mbean-export'` element serves the same purpose.

```
<context:mbean-export/>
```

You can provide a reference to a particular MBean `server` if necessary, and the `defaultDomain` attribute (a property of `AnnotationMBeanExporter`) accepts an alternate value for the generated MBean `ObjectNames` domains. This would be used in place of the fully qualified package name as described in the previous section on MetadataNamingStrategy.

```
@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDomain")
@Configuration
ContextConfiguration {

}
```

```
<context:mbean-export server="myMBeanServer" default-domain="myDomain"/>
```

> ℹ️ Do not use interface-based AOP proxies in combination with autodetection of JMX annotations in your bean classes. Interface-based proxies 'hide' the target class, which also hides the JMX managed resource annotations. Hence, use target-class proxies in that case: through setting the 'proxy-target-class' flag on `<aop:config/>`, `<tx:annotation-driven/>`, etc. Otherwise, your JMX beans might be silently ignored at startup…

4.5. JSR-160 Connectors

For remote access, Spring JMX module offers two `FactoryBean` implementations inside the `org.springframework.jmx.support` package for creating both server- and client-side connectors.

4.5.1. Server-side connectors

To have Spring JMX create, start and expose a JSR-160 `JMXConnectorServer` use the following configuration:

```
<bean id="serverConnector" class=
"org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

By default `ConnectorServerFactoryBean` creates a `JMXConnectorServer` bound to `"service:jmx:jmxmp://localhost:9875"`. The `serverConnector` bean thus exposes the local `MBeanServer` to clients through the JMXMP protocol on localhost, port 9875. Note that the JMXMP protocol is marked as optional by the JSR 160 specification: currently, the main open-source JMX implementation, MX4J, and the one provided with the JDK do *not* support JMXMP.

To specify another URL and register the `JMXConnectorServer` itself with the `MBeanServer` use the `serviceUrl` and `ObjectName` properties respectively:

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl"
            value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector
"/>
</bean>
```

If the `ObjectName` property is set Spring will automatically register your connector with the `MBeanServer` under that `ObjectName`. The example below shows the full set of parameters which you can pass to the `ConnectorServerFactoryBean` when creating a JMXConnector:

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=iiop"/>
    <property name="serviceUrl"
        value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
    <property name="threaded" value="true"/>
    <property name="daemon" value="true"/>
    <property name="environment">
        <map>
            <entry key="someKey" value="someValue"/>
        </map>
    </property>
</bean>
```

Note that when using a RMI-based connector you need the lookup service (tnameserv or rmiregistry) to be started in order for the name registration to complete. If you are using Spring to export remote services for you via RMI, then Spring will already have constructed an RMI registry. If not, you can easily start a registry using the following snippet of configuration:

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
    <property name="port" value="1099"/>
</bean>
```

4.5.2. Client-side connectors

To create an `MBeanServerConnection` to a remote JSR-160 enabled `MBeanServer` use the `MBeanServerConnectionFactoryBean` as shown below:

```
<bean id="clientConnector" class=
"org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value=
"service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi"/>
</bean>
```

4.5.3. JMX over Hessian or SOAP

JSR-160 permits extensions to the way in which communication is done between the client and the server. The examples above are using the mandatory RMI-based implementation required by the JSR-160 specification (IIOP and JRMP) and the (optional) JMXMP. By using other providers or JMX implementations (such as MX4J) you can take advantage of protocols like SOAP or Hessian over simple HTTP or SSL and others:

```
<bean id="serverConnector" class=
"org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=burlap"/>
    <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

In the case of the above example, MX4J 3.0.0 was used; see the official MX4J documentation for more information.

4.6. Accessing MBeans via proxies

Spring JMX allows you to create proxies that re-route calls to MBeans registered in a local or remote `MBeanServer`. These proxies provide you with a standard Java interface through which you can interact with your MBeans. The code below shows how to configure a proxy for an MBean running in a local `MBeanServer`:

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

Here you can see that a proxy is created for the MBean registered under the `ObjectName`: `bean:name=testBean`. The set of interfaces that the proxy will implement is controlled by the `proxyInterfaces` property and the rules for mapping methods and properties on these interfaces to operations and attributes on the MBean are the same rules used by the `InterfaceBasedMBeanInfoAssembler`.

The `MBeanProxyFactoryBean` can create a proxy to any MBean that is accessible via an `MBeanServerConnection`. By default, the local `MBeanServer` is located and used, but you can override this and provide an `MBeanServerConnection` pointing to a remote `MBeanServer` to cater for proxies pointing to remote MBeans:

```xml
<bean id="clientConnector"
        class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
    <property name="server" ref="clientConnector"/>
</bean>
```

Here you can see that we create an `MBeanServerConnection` pointing to a remote machine using the `MBeanServerConnectionFactoryBean`. This `MBeanServerConnection` is then passed to the `MBeanProxyFactoryBean` via the `server` property. The proxy that is created will forward all invocations to the `MBeanServer` via this `MBeanServerConnection`.

4.7. Notifications

Spring's JMX offering includes comprehensive support for JMX notifications.

4.7.1. Registering listeners for notifications

Spring's JMX support makes it very easy to register any number of `NotificationListeners` with any number of MBeans (this includes MBeans exported by Spring's `MBeanExporter` and MBeans registered via some other mechanism). By way of an example, consider the scenario where one would like to be informed (via a `Notification`) each and every time an attribute of a target MBean changes.

```java
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
        implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification
.getClass());
    }

}
```

```xml
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="bean:name=testBean1">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

With the above configuration in place, every time a JMX `Notification` is broadcast from the target MBean (`bean:name=testBean1`), the `ConsoleLoggingNotificationListener` bean that was registered as a

listener via the `notificationListenerMappings` property will be notified. The `ConsoleLoggingNotificationListener` bean can then take whatever action it deems appropriate in response to the `Notification`.

You can also use straight bean names as the link between exported beans and listeners:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="<em>testBean</em>">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="<em>testBean</em>" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

If one wants to register a single `NotificationListener` instance for all of the beans that the enclosing `MBeanExporter` is exporting, one can use the special wildcard `'*'` (sans quotes) as the key for an entry in the `notificationListenerMappings` property map; for example:

```
<property name="notificationListenerMappings">
    <map>
        <entry key="*">
            <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
    </map>
</property>
```

If one needs to do the inverse (that is, register a number of distinct listeners against an MBean), then one has to use the `notificationListeners` list property instead (and in preference to the `notificationListenerMappings` property). This time, instead of configuring simply a `NotificationListener` for a single MBean, one configures `NotificationListenerBean` instances… a `NotificationListenerBean` encapsulates a `NotificationListener` and the `ObjectName` (or `ObjectNames`) that it is to be registered against in an `MBeanServer`. The `NotificationListenerBean` also encapsulates

a number of other properties such as a `NotificationFilter` and an arbitrary handback object that can be used in advanced JMX notification scenarios.

The configuration when using `NotificationListenerBean` instances is not wildly different to what was presented previously:

```xml
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg>
                        <bean class="com.example.ConsoleLoggingNotificationListener"/>
                    </constructor-arg>
                    <property name="mappedObjectNames">
                        <list>
                            <value>bean:name=testBean1</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

The above example is equivalent to the first notification example. Lets assume then that we want to be given a handback object every time a `Notification` is raised, and that additionally we want to filter out extraneous `Notifications` by supplying a `NotificationFilter`. (For a full discussion of just what a handback object is, and indeed what a `NotificationFilter` is, please do consult that section of the JMX specification (1.2) entitled 'The JMX Notification Model'.)

```xml
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean1"/>
                <entry key="bean:name=testBean2" value-ref="testBean2"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg ref="customerNotificationListener"/>
                    <property name="mappedObjectNames">
                        <list>
                            <!-- handles notifications from two distinct MBeans -->
                            <value>bean:name=testBean1</value>
                            <value>bean:name=testBean2</value>
                        </list>
                    </property>
                    <property name="handback">
                        <bean class="java.lang.String">
                            <constructor-arg value="This could be anything..."/>
                        </bean>
                    </property>
                    <property name="notificationFilter" ref=
"customerNotificationListener"/>
                </bean>
            </list>
        </property>
    </bean>

    <!-- implements both the NotificationListener and NotificationFilter interfaces
-->
    <bean id="customerNotificationListener" class=
"com.example.ConsoleLoggingNotificationListener"/>

    <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="ANOTHER TEST"/>
        <property name="age" value="200"/>
    </bean>

</beans>
```

4.7.2. Publishing Notifications

Spring provides support not just for registering to receive `Notifications`, but also for publishing `Notifications`.

> ℹ️ Please note that this section is really only relevant to Spring managed beans that have been exposed as MBeans via an `MBeanExporter`; any existing, user-defined MBeans should use the standard JMX APIs for notification publication.

The key interface in Spring's JMX notification publication support is the `NotificationPublisher` interface (defined in the `org.springframework.jmx.export.notification` package). Any bean that is going to be exported as an MBean via an `MBeanExporter` instance can implement the related `NotificationPublisherAware` interface to gain access to a `NotificationPublisher` instance. The `NotificationPublisherAware` interface simply supplies an instance of a `NotificationPublisher` to the implementing bean via a simple setter method, which the bean can then use to publish `Notifications`.

As stated in the javadocs of the `NotificationPublisher` class, managed beans that are publishing events via the `NotificationPublisher` mechanism are *not* responsible for the state management of any notification listeners and the like … Spring's JMX support will take care of handling all the JMX infrastructure issues. All one need do as an application developer is implement the `NotificationPublisherAware` interface and start publishing events using the supplied `NotificationPublisher` instance. Note that the `NotificationPublisher` will be set *after* the managed bean has been registered with an `MBeanServer`.

Using a `NotificationPublisher` instance is quite straightforward… one simply creates a JMX `Notification` instance (or an instance of an appropriate `Notification` subclass), populates the notification with the data pertinent to the event that is to be published, and one then invokes the `sendNotification(Notification)` on the `NotificationPublisher` instance, passing in the `Notification`.

Find below a simple example… in this scenario, exported instances of the `JmxTestBean` are going to publish a `NotificationEvent` every time the `add(int, int)` operation is invoked.

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher)
{

        this.publisher = notificationPublisher;
    }

}
```

The `NotificationPublisher` interface and the machinery to get it all working is one of the nicer features of Spring's JMX support. It does however come with the price tag of coupling your classes to both Spring and JMX; as always, the advice here is to be pragmatic… if you need the functionality offered by the `NotificationPublisher` and you can accept the coupling to both Spring and JMX, then do so.

4.8. Further resources

This section contains links to further resources about JMX.

- The JMX homepage at Oracle

- The JMX specification (JSR-000003)

- The JMX Remote API specification (JSR-000160)

- The MX4J homepage (an Open Source implementation of various JMX specs)

Chapter 5. JCA CCI

5.1. Introduction

Java EE provides a specification to standardize access to enterprise information systems (EIS): the JCA (Java EE Connector Architecture). This specification is divided into several different parts:

- SPI (Service provider interfaces) that the connector provider must implement. These interfaces constitute a resource adapter which can be deployed on a Java EE application server. In such a scenario, the server manages connection pooling, transaction and security (managed mode). The application server is also responsible for managing the configuration, which is held outside the client application. A connector can be used without an application server as well; in this case, the application must configure it directly (non-managed mode).

- CCI (Common Client Interface) that an application can use to interact with the connector and thus communicate with an EIS. An API for local transaction demarcation is provided as well.

The aim of the Spring CCI support is to provide classes to access a CCI connector in typical Spring style, leveraging the Spring Framework's general resource and transaction management facilities.

> The client side of connectors doesn't alway use CCI. Some connectors expose their own APIs, only providing JCA resource adapter to use the system contracts of a Java EE container (connection pooling, global transactions, security). Spring does not offer special support for such connector-specific APIs.

5.2. Configuring CCI

5.2.1. Connector configuration

The base resource to use JCA CCI is the `ConnectionFactory` interface. The connector used must provide an implementation of this interface.

To use your connector, you can deploy it on your application server and fetch the `ConnectionFactory` from the server's JNDI environment (managed mode). The connector must be packaged as a RAR file (resource adapter archive) and contain a `ra.xml` file to describe its deployment characteristics. The actual name of the resource is specified when you deploy it. To access it within Spring, simply use Spring's `JndiObjectFactoryBean` / `<jee:jndi-lookup>` fetch the factory by its JNDI name.

Another way to use a connector is to embed it in your application (non-managed mode), not using an application server to deploy and configure it. Spring offers the possibility to configure a connector as a bean, through a provided `FactoryBean` (`LocalConnectionFactoryBean`). In this manner, you only need the connector library in the classpath (no RAR file and no `ra.xml` descriptor needed). The library must be extracted from the connector's RAR file, if necessary.

Once you have got access to your `ConnectionFactory` instance, you can inject it into your components. These components can either be coded against the plain CCI API or leverage Spring's support classes for CCI access (e.g. `CciTemplate`).

> When you use a connector in non-managed mode, you can't use global transactions because the resource is never enlisted / delisted in the current global transaction of the current thread. The resource is simply not aware of any global Java EE transactions that might be running.

5.2.2. ConnectionFactory configuration in Spring

In order to make connections to the EIS, you need to obtain a `ConnectionFactory` from the application server if you are in a managed mode, or directly from Spring if you are in a non-managed mode.

In a managed mode, you access a `ConnectionFactory` from JNDI; its properties will be configured in the application server.

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>
```

In non-managed mode, you must configure the `ConnectionFactory` you want to use in the configuration of Spring as a JavaBean. The `LocalConnectionFactoryBean` class offers this setup style, passing in the `ManagedConnectionFactory` implementation of your connector, exposing the application-level CCI `ConnectionFactory`.

```
<bean id="eciManagedConnectionFactory" class=
"com.ibm.connector2.cics.ECIManagedConnectionFactory">
    <property name="serverName" value="TXSERIES"/>
    <property name="connectionURL" value="tcp://localhost/"/>
    <property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class=
"org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

> You can't directly instantiate a specific `ConnectionFactory`. You need to go through the corresponding implementation of the `ManagedConnectionFactory` interface for your connector. This interface is part of the JCA SPI specification.

5.2.3. Configuring CCI connections

JCA CCI allow the developer to configure the connections to the EIS using the `ConnectionSpec` implementation of your connector. In order to configure its properties, you need to wrap the target connection factory with a dedicated adapter, `ConnectionSpecConnectionFactoryAdapter`. So, the dedicated `ConnectionSpec` can be configured with the property `connectionSpec` (as an inner bean).

This property is not mandatory because the CCI `ConnectionFactory` interface defines two different methods to obtain a CCI connection. Some of the `ConnectionSpec` properties can often be configured

in the application server (in managed mode) or on the corresponding local `ManagedConnectionFactory` implementation.

```
public interface ConnectionFactory implements Serializable, Referenceable {
    ...
    Connection getConnection() throws ResourceException;
    Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
    ...
}
```

Spring provides a `ConnectionSpecConnectionFactoryAdapter` that allows for specifying a `ConnectionSpec` instance to use for all operations on a given factory. If the adapter's `connectionSpec` property is specified, the adapter uses the `getConnection` variant with the `ConnectionSpec` argument, otherwise the variant without argument.

```
<bean id="managedConnectionFactory"
        class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
    <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
        class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
        class=
"org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
    <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
    <property name="connectionSpec">
        <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
            <property name="user" value="sa"/>
            <property name="password" value=""/>
        </bean>
    </property>
</bean>
```

5.2.4. Using a single CCI connection

If you want to use a single CCI connection, Spring provides a further `ConnectionFactory` adapter to manage this. The `SingleConnectionFactory` adapter class will open a single connection lazily and close it when this bean is destroyed at application shutdown. This class will expose special `Connection` proxies that behave accordingly, all sharing the same underlying physical connection.

```xml
<bean id="eciManagedConnectionFactory"
        class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
    <property name="serverName" value="TEST"/>
    <property name="connectionURL" value="tcp://localhost/"/>
    <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
        class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
        class="org.springframework.jca.cci.connection.SingleConnectionFactory">
    <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>
```

> This `ConnectionFactory` adapter cannot directly be configured with a
> `ConnectionSpec`. Use an intermediary `ConnectionSpecConnectionFactoryAdapter` that
> the `SingleConnectionFactory` talks to if you require a single connection for a
> specific `ConnectionSpec`.

5.3. Using Spring's CCI access support

5.3.1. Record conversion

One of the aims of the JCA CCI support is to provide convenient facilities for manipulating CCI
records. The developer can specify the strategy to create records and extract datas from records, for
use with Spring's `CciTemplate`. The following interfaces will configure the strategy to use input and
output records if you don't want to work with records directly in your application.

In order to create an input `Record`, the developer can use a dedicated implementation of the
`RecordCreator` interface.

```java
public interface RecordCreator {

    Record createRecord(RecordFactory recordFactory) throws ResourceException,
DataAccessException;

}
```

As you can see, the `createRecord(..)` method receives a `RecordFactory` instance as parameter, which
corresponds to the `RecordFactory` of the `ConnectionFactory` used. This reference can be used to create
`IndexedRecord` or `MappedRecord` instances. The following sample shows how to use the `RecordCreator`
interface and indexed/mapped records.

```
public class MyRecordCreator implements RecordCreator {

    public Record createRecord(RecordFactory recordFactory) throws ResourceException {
        IndexedRecord input = recordFactory.createIndexedRecord("input");
        input.add(new Integer(id));
        return input;
    }


}
```

An output `Record` can be used to receive data back from the EIS. Hence, a specific implementation of the `RecordExtractor` interface can be passed to Spring's `CciTemplate` for extracting data from the output `Record`.

```
public interface RecordExtractor {

    Object extractData(Record record) throws ResourceException, SQLException,
DataAccessException;


}
```

The following sample shows how to use the `RecordExtractor` interface.

```
public class MyRecordExtractor implements RecordExtractor {

    public Object extractData(Record record) throws ResourceException {
        CommAreaRecord commAreaRecord = (CommAreaRecord) record;
        String str = new String(commAreaRecord.toByteArray());
        String field1 = string.substring(0,6);
        String field2 = string.substring(6,1);
        return new OutputObject(Long.parseLong(field1), field2);
    }


}
```

5.3.2. CciTemplate

The `CciTemplate` is the central class of the core CCI support package (`org.springframework.jca.cci.core`). It simplifies the use of CCI since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It cares for the lifecycle of connection and interaction objects, letting application code focus on generating input records from application data and extracting application data from output records.

The JCA CCI specification defines two distinct methods to call operations on an EIS. The CCI `Interaction` interface provides two execute method signatures:

```
public interface javax.resource.cci.Interaction {

    ...

    boolean execute(InteractionSpec spec, Record input, Record output) throws
ResourceException;

    Record execute(InteractionSpec spec, Record input) throws ResourceException;

    ...

}
```

Depending on the template method called, `CciTemplate` will know which `execute` method to call on the interaction. In any case, a correctly initialized `InteractionSpec` instance is mandatory.

`CciTemplate.execute(..)` can be used in two ways:

- With direct `Record` arguments. In this case, you simply need to pass the CCI input record in, and the returned object be the corresponding CCI output record.
- With application objects, using record mapping. In this case, you need to provide corresponding `RecordCreator` and `RecordExtractor` instances.

With the first approach, the following methods of the template will be used. These methods directly correspond to those on the `Interaction` interface.

```
public class CciTemplate implements CciOperations {

    public Record execute(InteractionSpec spec, Record inputRecord)
            throws DataAccessException { ... }

    public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
            throws DataAccessException { ... }

}
```

With the second approach, we need to specify the record creation and record extraction strategies as arguments. The interfaces used are those describe in the previous section on record conversion. The corresponding `CciTemplate` methods are the following:

```
public class CciTemplate implements CciOperations {

    public Record execute(InteractionSpec spec,
            RecordCreator inputCreator) throws DataAccessException {
        // ...
    }

    public Object execute(InteractionSpec spec, Record inputRecord,
            RecordExtractor outputExtractor) throws DataAccessException {
        // ...
    }

    public Object execute(InteractionSpec spec, RecordCreator creator,
            RecordExtractor extractor) throws DataAccessException {
        // ...
    }

}
```

Unless the `outputRecordCreator` property is set on the template (see the following section), every method will call the corresponding `execute` method of the CCI `Interaction` with two parameters: `InteractionSpec` and input `Record`, receiving an output `Record` as return value.

`CciTemplate` also provides methods to create `IndexRecord` and `MappedRecord` outside a `RecordCreator` implementation, through its `createIndexRecord(..)` and `createMappedRecord(..)` methods. This can be used within DAO implementations to create `Record` instances to pass into corresponding `CciTemplate.execute(..)` methods.

```
public class CciTemplate implements CciOperations {

    public IndexedRecord createIndexedRecord(String name) throws DataAccessException {
... }

    public MappedRecord createMappedRecord(String name) throws DataAccessException {
... }

}
```

5.3.3. DAO support

Spring's CCI support provides a abstract class for DAOs, supporting injection of a `ConnectionFactory` or a `CciTemplate` instances. The name of the class is `CciDaoSupport`: It provides simple `setConnectionFactory` and `setCciTemplate` methods. Internally, this class will create a `CciTemplate` instance for a passed-in `ConnectionFactory`, exposing it to concrete data access implementations in subclasses.

```
public abstract class CciDaoSupport {

    public void setConnectionFactory(ConnectionFactory connectionFactory) {
        // ...
    }

    public ConnectionFactory getConnectionFactory() {
        // ...
    }

    public void setCciTemplate(CciTemplate cciTemplate) {
        // ...
    }

    public CciTemplate getCciTemplate() {
        // ...
    }

}
```

5.3.4. Automatic output record generation

If the connector used only supports the `Interaction.execute(..)` method with input and output records as parameters (that is, it requires the desired output record to be passed in instead of returning an appropriate output record), you can set the `outputRecordCreator` property of the `CciTemplate` to automatically generate an output record to be filled by the JCA connector when the response is received. This record will be then returned to the caller of the template.

This property simply holds an implementation of the `RecordCreator` interface, used for that purpose. The `RecordCreator` interface has already been discussed in Record conversion. The `outputRecordCreator` property must be directly specified on the `CciTemplate`. This could be done in the application code like so:

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

Or (recommended) in the Spring configuration, if the `CciTemplate` is configured as a dedicated bean instance:

```xml
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
    <property name="connectionFactory" ref="eciConnectionFactory"/>
    <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

> As the `CciTemplate` class is thread-safe, it will usually be configured as a shared instance.

5.3.5. Summary

The following table summarizes the mechanisms of the `CciTemplate` class and the corresponding methods called on the CCI `Interaction` interface:

Table 8. Usage of Interaction execute methods

CciTemplate method signature	CciTemplate outputRecordCreator property	execute method called on the CCI Interaction
Record execute(InteractionSpec, Record)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, Record)	set	boolean execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	not set	void execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, Record, RecordExtractor)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, Record, RecordExtractor)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	set	void execute(InteractionSpec, Record, Record)

5.3.6. Using a CCI Connection and Interaction directly

`CciTemplate` also offers the possibility to work directly with CCI connections and interactions, in the same manner as `JdbcTemplate` and `JmsTemplate`. This is useful when you want to perform multiple operations on a CCI connection or interaction, for example.

The interface `ConnectionCallback` provides a CCI `Connection` as argument, in order to perform custom operations on it, plus the CCI `ConnectionFactory` which the `Connection` was created with. The latter can be useful for example to get an associated `RecordFactory` instance and create indexed/mapped records, for example.

```
public interface ConnectionCallback {

    Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
            throws ResourceException, SQLException, DataAccessException;


}
```

The interface `InteractionCallback` provides the CCI `Interaction`, in order to perform custom operations on it, plus the corresponding CCI `ConnectionFactory`.

```
public interface InteractionCallback {

    Object doInInteraction(Interaction interaction, ConnectionFactory
connectionFactory)
        throws ResourceException, SQLException, DataAccessException;


}
```

> `InteractionSpec` objects can either be shared across multiple template calls or newly created inside every callback method. This is completely up to the DAO implementation.

5.3.7. Example for CciTemplate usage

In this section, the usage of the `CciTemplate` will be shown to acces to a CICS with ECI mode, with the IBM CICS ECI connector.

Firstly, some initializations on the CCI `InteractionSpec` must be done to specify which CICS program to access and how to interact with it.

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

Then the program can use CCI via Spring's template and specify mappings between custom objects and CCI `Records`.

```java
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public OutputObject getData(InputObject input) {
        ECIInteractionSpec interactionSpec = ...;

    OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,
        new RecordCreator() {
            public Record createRecord(RecordFactory recordFactory) throws
ResourceException {
                return new CommAreaRecord(input.toString().getBytes());
            }
        },
        new RecordExtractor() {
            public Object extractData(Record record) throws ResourceException {
                CommAreaRecord commAreaRecord = (CommAreaRecord)record;
                String str = new String(commAreaRecord.toByteArray());
                String field1 = string.substring(0,6);
                String field2 = string.substring(6,1);
                return new OutputObject(Long.parseLong(field1), field2);
            }
        });

        return output;
    }
}
```

As discussed previously, callbacks can be used to work directly on CCI connections or interactions.

```java
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public OutputObject getData(InputObject input) {
        ObjectOutput output = (ObjectOutput) getCciTemplate().execute(
            new ConnectionCallback() {
                public Object doInConnection(Connection connection,
                        ConnectionFactory factory) throws ResourceException {

                    // do something...

                }
            });
    }
    return output;
    }

}
```

> ℹ️ With a `ConnectionCallback`, the `Connection` used will be managed and closed by the `CciTemplate`, but any interactions created on the connection must be managed by the callback implementation.

For a more specific callback, you can implement an `InteractionCallback`. The passed-in `Interaction` will be managed and closed by the `CciTemplate` in this case.

```java
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public String getData(String input) {
        ECIInteractionSpec interactionSpec = ...;
        String output = (String) getCciTemplate().execute(interactionSpec,
            new InteractionCallback() {
                public Object doInInteraction(Interaction interaction,
                        ConnectionFactory factory) throws ResourceException {
                    Record input = new CommAreaRecord(inputString.getBytes());
                    Record output = new CommAreaRecord();
                    interaction.execute(holder.getInteractionSpec(), input, output);
                    return new String(output.toByteArray());
                }
            });
        return output;
    }

}
```

For the examples above, the corresponding configuration of the involved Spring beans could look like this in non-managed mode:

```xml
<bean id="managedConnectionFactory" class=
"com.ibm.connector2.cics.ECIManagedConnectionFactory">
    <property name="serverName" value="TXSERIES"/>
    <property name="connectionURL" value="local:"/>
    <property name="userName" value="CICSUSER"/>
    <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class=
"org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

```
<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

5.4. Modeling CCI access as operation objects

The `org.springframework.jca.cci.object` package contains support classes that allow you to access the EIS in a different style: through reusable operation objects, analogous to Spring's JDBC operation objects (see JDBC chapter). This will usually encapsulate the CCI API: an application-level input object will be passed to the operation object, so it can construct the input record and then convert the received record data to an application-level output object and return it.

> ℹ️ This approach is internally based on the `CciTemplate` class and the `RecordCreator` / `RecordExtractor` interfaces, reusing the machinery of Spring's core CCI support.

5.4.1. MappingRecordOperation

`MappingRecordOperation` essentially performs the same work as `CciTemplate`, but represents a specific, pre-configured operation as an object. It provides two template methods to specify how to convert an input object to a input record, and how to convert an output record to an output object (record mapping):

- `createInputRecord(..)` to specify how to convert an input object to an input `Record`

- `extractOutputData(..)` to specify how to extract an output object from an output `Record`

Here are the signatures of these methods:

```
public abstract class MappingRecordOperation extends EisOperation {

    ...

    protected abstract Record createInputRecord(RecordFactory recordFactory,
            Object inputObject) throws ResourceException, DataAccessException {
        // ...
    }

    protected abstract Object extractOutputData(Record outputRecord)
            throws ResourceException, SQLException, DataAccessException {
        // ...
    }

    ...

}
```

Thereafter, in order to execute an EIS operation, you need to use a single execute method, passing in an application-level input object and receiving an application-level output object as result:

```
public abstract class MappingRecordOperation extends EisOperation {

    ...

    public Object execute(Object inputObject) throws DataAccessException {
    }

    ...

}
```

As you can see, contrary to the `CciTemplate` class, this `execute(..)` method does not have an `InteractionSpec` as argument. Instead, the `InteractionSpec` is global to the operation. The following constructor must be used to instantiate an operation object with a specific `InteractionSpec`:

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation
(getConnectionFactory(), spec);
...
```

5.4.2. MappingCommAreaOperation

Some connectors use records based on a COMMAREA which represents an array of bytes containing parameters to send to the EIS and data returned by it. Spring provides a special operation class for working directly on COMMAREA rather than on records. The `MappingCommAreaOperation` class extends the `MappingRecordOperation` class to provide such special COMMAREA support. It implicitly uses the `CommAreaRecord` class as input and output record type, and provides two new methods to convert an input object into an input COMMAREA and the output COMMAREA into an output object.

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {

    ...

    protected abstract byte[] objectToBytes(Object inObject)
            throws IOException, DataAccessException;

    protected abstract Object bytesToObject(byte[] bytes)
        throws IOException, DataAccessException;

    ...

}
```

5.4.3. Automatic output record generation

As every `MappingRecordOperation` subclass is based on CciTemplate internally, the same way to automatically generate output records as with `CciTemplate` is available. Every operation object provides a corresponding `setOutputRecordCreator(..)` method. For further information, see Automatic output record generation.

5.4.4. Summary

The operation object approach uses records in the same manner as the `CciTemplate` class.

Table 9. Usage of Interaction execute methods

MappingRecordOperation method signature	MappingRecordOperation outputRecordCreator property	execute method called on the CCI Interaction
Object execute(Object)	not set	Record execute(InteractionSpec, Record)
Object execute(Object)	set	boolean execute(InteractionSpec, Record, Record)

5.4.5. Example for MappingRecordOperation usage

In this section, the usage of the `MappingRecordOperation` will be shown to access a database with the Blackbox CCI connector.

> ℹ️ The original version of this connector is provided by the Java EE SDK (version 1.3), available from Oracle.

Firstly, some initializations on the CCI `InteractionSpec` must be done to specify which SQL request to execute. In this sample, we directly define the way to convert the parameters of the request to a CCI record and the way to convert the CCI result record to an instance of the `Person` class.

```java
public class PersonMappingOperation extends MappingRecordOperation {

    public PersonMappingOperation(ConnectionFactory connectionFactory) {
        setConnectionFactory(connectionFactory);
        CciInteractionSpec interactionSpec = new CciConnectionSpec();
        interactionSpec.setSql("select * from person where person_id=?");
        setInteractionSpec(interactionSpec);
    }

    protected Record createInputRecord(RecordFactory recordFactory,
            Object inputObject) throws ResourceException {
        Integer id = (Integer) inputObject;
        IndexedRecord input = recordFactory.createIndexedRecord("input");
        input.add(new Integer(id));
        return input;
    }

    protected Object extractOutputData(Record outputRecord)
            throws ResourceException, SQLException {
        ResultSet rs = (ResultSet) outputRecord;
        Person person = null;
        if (rs.next()) {
            Person person = new Person();
            person.setId(rs.getInt("person_id"));
            person.setLastName(rs.getString("person_last_name"));
            person.setFirstName(rs.getString("person_first_name"));
        }
        return person;
    }
}
```

Then the application can execute the operation object, with the person identifier as argument. Note that operation object could be set up as shared instance, as it is thread-safe.

```java
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public Person getPerson(int id) {
        PersonMappingOperation query = new PersonMappingOperation(
getConnectionFactory());
        Person person = (Person) query.execute(new Integer(id));
        return person;
    }
}
```

The corresponding configuration of Spring beans could look as follows in non-managed mode:

```xml
<bean id="managedConnectionFactory"
        class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
    <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
        class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
        class=
"org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
    <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
    <property name="connectionSpec">
        <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
            <property name="user" value="sa"/>
            <property name="password" value=""/>
        </bean>
    </property>
</bean>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

```xml
<jee:jndi-lookup id="targetConnectionFactory" jndi-name="eis/blackbox"/>

<bean id="connectionFactory"
        class=
"org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
    <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
    <property name="connectionSpec">
        <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
            <property name="user" value="sa"/>
            <property name="password" value=""/>
        </bean>
    </property>
</bean>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

5.4.6. Example for MappingCommAreaOperation usage

In this section, the usage of the `MappingCommAreaOperation` will be shown: accessing a CICS with ECI mode with the IBM CICS ECI connector.

Firstly, the CCI `InteractionSpec` needs to be initialized to specify which CICS program to access and how to interact with it.

```java
public abstract class EciMappingOperation extends MappingCommAreaOperation {

    public EciMappingOperation(ConnectionFactory connectionFactory, String
programName) {
        setConnectionFactory(connectionFactory);
        ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),
        interactionSpec.setFunctionName(programName);
        interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
        interactionSpec.setCommareaLength(30);
        setInteractionSpec(interactionSpec);
        setOutputRecordCreator(new EciOutputRecordCreator());
    }

    private static class EciOutputRecordCreator implements RecordCreator {
        public Record createRecord(RecordFactory recordFactory) throws
ResourceException {
            return new CommAreaRecord();
        }
    }

}
```

The abstract `EciMappingOperation` class can then be subclassed to specify mappings between custom objects and `Records`.

```java
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public OutputObject getData(Integer id) {
        EciMappingOperation query = new EciMappingOperation(getConnectionFactory(),
"MYPROG") {

            protected abstract byte[] objectToBytes(Object inObject) throws
IOException {
                Integer id = (Integer) inObject;
                return String.valueOf(id);
            }

            protected abstract Object bytesToObject(byte[] bytes) throws IOException;
                String str = new String(bytes);
                String field1 = str.substring(0,6);
                String field2 = str.substring(6,1);
                String field3 = str.substring(7,1);
                return new OutputObject(field1, field2, field3);
            }
        });

        return (OutputObject) query.execute(new Integer(id));
    }

}
```

The corresponding configuration of Spring beans could look as follows in non-managed mode:

```xml
<bean id="managedConnectionFactory" class=
"com.ibm.connector2.cics.ECIManagedConnectionFactory">
    <property name="serverName" value="TXSERIES"/>
    <property name="connectionURL" value="local:"/>
    <property name="userName" value="CICSUSER"/>
    <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class=
"org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

```
<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

5.5. Transactions

JCA specifies several levels of transaction support for resource adapters. The kind of transactions that your resource adapter supports is specified in its `ra.xml` file. There are essentially three options: none (for example with CICS EPI connector), local transactions (for example with a CICS ECI connector), global transactions (for example with an IMS connector).

```
<connector>
    <resourceadapter>
        <!-- <transaction-support>NoTransaction</transaction-support> -->
        <!-- <transaction-support>LocalTransaction</transaction-support> -->
        <transaction-support>XATransaction</transaction-support>
    </resourceadapter>
<connector>
```

For global transactions, you can use Spring's generic transaction infrastructure to demarcate transactions, with `JtaTransactionManager` as backend (delegating to the Java EE server's distributed transaction coordinator underneath).

For local transactions on a single CCI `ConnectionFactory`, Spring provides a specific transaction management strategy for CCI, analogous to the `DataSourceTransactionManager` for JDBC. The CCI API defines a local transaction object and corresponding local transaction demarcation methods. Spring's `CciLocalTransactionManager` executes such local CCI transactions, fully compliant with Spring's generic `PlatformTransactionManager` abstraction.

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>

<bean id="eciTransactionManager"
        class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
    <property name="connectionFactory" ref="eciConnectionFactory"/>
</bean>
```

Both transaction strategies can be used with any of Spring's transaction demarcation facilities, be it declarative or programmatic. This is a consequence of Spring's generic `PlatformTransactionManager` abstraction, which decouples transaction demarcation from the actual execution strategy. Simply switch between `JtaTransactionManager` and `CciLocalTransactionManager` as needed, keeping your transaction demarcation as-is.

For more information on Spring's transaction facilities, see the chapter entitled Transaction Management.

Chapter 6. Email

6.1. Introduction

> **Library dependencies**
>
> The following JAR needs to be on the classpath of your application in order to use the Spring Framework's email library.
>
> - The JavaMail library
>
> This library is freely available on the web — for example, in Maven Central as `com.sun.mail:javax.mail`.

The Spring Framework provides a helpful utility library for sending email that shields the user from the specifics of the underlying mailing system and is responsible for low level resource handling on behalf of the client.

The `org.springframework.mail` package is the root level package for the Spring Framework's email support. The central interface for sending emails is the `MailSender` interface; a simple value object encapsulating the properties of a simple mail such as *from* and *to* (plus many others) is the `SimpleMailMessage` class. This package also contains a hierarchy of checked exceptions which provide a higher level of abstraction over the lower level mail system exceptions with the root exception being `MailException`. Please refer to the javadocs for more information on the rich mail exception hierarchy.

The `org.springframework.mail.javamail.JavaMailSender` interface adds specialized *JavaMail* features such as MIME message support to the `MailSender` interface (from which it inherits). `JavaMailSender` also provides a callback interface for preparation of JavaMail MIME messages, called `org.springframework.mail.javamail.MimeMessagePreparator`

6.2. Usage

Let's assume there is a business interface called `OrderManager`:

```
public interface OrderManager {

    void placeOrder(Order order);

}
```

Let us also assume that there is a requirement stating that an email message with an order number needs to be generated and sent to a customer placing the relevant order.

6.2.1. Basic MailSender and SimpleMailMessage usage

```java
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class SimpleOrderManager implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage templateMessage;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setTemplateMessage(SimpleMailMessage templateMessage) {
        this.templateMessage = templateMessage;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        // Create a thread safe "copy" of the template message and customize it
        SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear " + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try{
            this.mailSender.send(msg);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

Find below the bean definitions for the above code:

```xml
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.mycompany.com"/>
</bean>

<!-- this is a template message that we can pre-load with default state -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="customerservice@mycompany.com"/>
    <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">
    <property name="mailSender" ref="mailSender"/>
    <property name="templateMessage" ref="templateMessage"/>
</bean>
```

6.2.2. Using the JavaMailSender and the MimeMessagePreparator

Here is another implementation of `OrderManager` using the `MimeMessagePreparator` callback interface. Please note in this case that the `mailSender` property is of type `JavaMailSender` so that we are able to use the JavaMail `MimeMessage` class:

```java
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws Exception {

                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));
                mimeMessage.setText(
                        "Dear " + order.getCustomer().getFirstName() + " "
                            + order.getCustomer().getLastName()
                            + ", thank you for placing order. Your order number is "
                            + order.getOrderNumber());
            }
        };

        try {
            this.mailSender.send(preparator);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

The mail code is a crosscutting concern and could well be a candidate for refactoring into a custom Spring AOP aspect, which then could be executed at appropriate joinpoints on the `OrderManager` target.

The Spring Framework's mail support ships with the standard JavaMail implementation. Please refer to the relevant javadocs for more information.

6.3. Using the JavaMail MimeMessageHelper

A class that comes in pretty handy when dealing with JavaMail messages is the `org.springframework.mail.javamail.MimeMessageHelper` class, which shields you from having to use the verbose JavaMail API. Using the `MimeMessageHelper` it is pretty easy to create a `MimeMessage`:

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

6.3.1. Sending attachments and inline resources

Multipart email messages allow for both attachments and inline resources. Examples of inline resources would be images or a stylesheet you want to use in your message, but that you don't want displayed as an attachment.

Attachments

The following example shows you how to use the `MimeMessageHelper` to send an email along with a single JPEG image attachment.

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)
FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

sender.send(message);
```

Inline resources

The following example shows you how to use the `MimeMessageHelper` to send an email along with an inline image.

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

sender.send(message);
```

> ⚠️ Inline resources are added to the mime message using the specified `Content-ID` (`identifier1234` in the above example). The order in which you are adding the text and the resource are *very* important. Be sure to *first add the text* and after that the resources. If you are doing it the other way around, it won't work!

6.3.2. Creating email content using a templating library

The code in the previous examples explicitly created the content of the email message, using methods calls such as `message.setText(..)`. This is fine for simple cases, and it is okay in the context of the aforementioned examples, where the intent was to show you the very basics of the API.

In your typical enterprise application though, you are not going to create the content of your emails using the above approach for a number of reasons.

- Creating HTML-based email content in Java code is tedious and error prone

- There is no clear separation between display logic and business logic

- Changing the display structure of the email content requires writing Java code, recompiling, redeploying…

Typically the approach taken to address these issues is to use a template library such as FreeMarker to define the display structure of email content. This leaves your code tasked only with creating the data that is to be rendered in the email template and sending the email. It is definitely a best practice for when the content of your emails becomes even moderately complex, and with the Spring Framework's support classes for FreeMarker becomes quite easy to do.

Chapter 7. Task Execution and Scheduling

7.1. Introduction

The Spring Framework provides abstractions for asynchronous execution and scheduling of tasks with the `TaskExecutor` and `TaskScheduler` interfaces, respectively. Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment. Ultimately the use of these implementations behind the common interfaces abstracts away the differences between Java SE 5, Java SE 6 and Java EE environments.

Spring also features integration classes for supporting scheduling with the `Timer`, part of the JDK since 1.3, and the Quartz Scheduler (http://quartz-scheduler.org). Both of those schedulers are set up using a `FactoryBean` with optional references to `Timer` or `Trigger` instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the `Timer` is available that allows you to invoke a method of an existing target object (analogous to the normal `MethodInvokingFactoryBean` operation).

7.2. The Spring TaskExecutor abstraction

Executors are the JDK name for the concept of thread pools. The "executor" naming is due to the fact that there is no guarantee that the underlying implementation is actually a pool; an executor may be single-threaded or even synchronous. Spring's abstraction hides implementation details between Java SE and Java EE environments.

Spring's `TaskExecutor` interface is identical to the `java.util.concurrent.Executor` interface. In fact, originally, its primary reason for existence was to abstract away the need for Java 5 when using thread pools. The interface has a single method `execute(Runnable task)` that accepts a task for execution based on the semantics and configuration of the thread pool.

The `TaskExecutor` was originally created to give other Spring components an abstraction for thread pooling where needed. Components such as the `ApplicationEventMulticaster`, JMS's `AbstractMessageListenerContainer`, and Quartz integration all use the `TaskExecutor` abstraction to pool threads. However, if your beans need thread pooling behavior, it is possible to use this abstraction for your own needs.

7.2.1. TaskExecutor types

There are a number of pre-built implementations of `TaskExecutor` included with the Spring distribution. In all likelihood, you shouldn't ever need to implement your own.

- `SimpleAsyncTaskExecutor` This implementation does not reuse any threads, rather it starts up a new thread for each invocation. However, it does support a concurrency limit which will block any invocations that are over the limit until a slot has been freed up. If you are looking for true pooling, see the discussions of `SimpleThreadPoolTaskExecutor` and `ThreadPoolTaskExecutor` below.

- `SyncTaskExecutor` This implementation doesn't execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multi-threading isn't necessary such as simple test cases.

- `ConcurrentTaskExecutor` This implementation is an adapter for a `java.util.concurrent.Executor` object. There is an alternative, `ThreadPoolTaskExecutor`, that exposes the `Executor` configuration parameters as bean properties. It is rare to need to use the `ConcurrentTaskExecutor`, but if the `ThreadPoolTaskExecutor` isn't flexible enough for your needs, the `ConcurrentTaskExecutor` is an alternative.

- `SimpleThreadPoolTaskExecutor` This implementation is actually a subclass of Quartz's `SimpleThreadPool` which listens to Spring's lifecycle callbacks. This is typically used when you have a thread pool that may need to be shared by both Quartz and non-Quartz components.

- `ThreadPoolTaskExecutor` This implementation is the most commonly used one. It exposes bean properties for configuring a `java.util.concurrent.ThreadPoolExecutor` and wraps it in a `TaskExecutor`. If you need to adapt to a different kind of `java.util.concurrent.Executor`, it is recommended that you use a `ConcurrentTaskExecutor` instead.

- `WorkManagerTaskExecutor`

> CommonJ is a set of specifications jointly developed between BEA and IBM. These specifications are not Java EE standards, but are standard across BEA's and IBM's Application Server implementations.

This implementation uses the CommonJ `WorkManager` as its backing implementation and is the central convenience class for setting up a CommonJ `WorkManager` reference in a Spring context. Similar to the `SimpleThreadPoolTaskExecutor`, this class implements the `WorkManager` interface and therefore can be used directly as a `WorkManager` as well.

7.2.2. Using a TaskExecutor

Spring's `TaskExecutor` implementations are used as simple JavaBeans. In the example below, we define a bean that uses the `ThreadPoolTaskExecutor` to asynchronously print out a set of messages.

```java
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

    private class MessagePrinterTask implements Runnable {

        private String message;

        public MessagePrinterTask(String message) {
            this.message = message;
        }

        public void run() {
            System.out.println(message);
        }

    }

    private TaskExecutor taskExecutor;

    public TaskExecutorExample(TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public void printMessages() {
        for(int i = 0; i < 25; i++) {
            taskExecutor.execute(new MessagePrinterTask("Message" + i));
        }
    }

}
```

As you can see, rather than retrieving a thread from the pool and executing yourself, you add your Runnable to the queue and the TaskExecutor uses its internal rules to decide when the task gets executed.

To configure the rules that the TaskExecutor will use, simple bean properties have been exposed.

```xml
<bean id="taskExecutor" class=
"org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="5" />
    <property name="maxPoolSize" value="10" />
    <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
    <constructor-arg ref="taskExecutor" />
</bean>
```

7.3. The Spring TaskScheduler abstraction

In addition to the `TaskExecutor` abstraction, Spring 3.0 introduces a `TaskScheduler` with a variety of methods for scheduling tasks to run at some point in the future.

```java
public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task, Trigger trigger);

    ScheduledFuture schedule(Runnable task, Date startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

}
```

The simplest method is the one named 'schedule' that takes a `Runnable` and `Date` only. That will cause the task to run once after the specified time. All of the other methods are capable of scheduling tasks to run repeatedly. The fixed-rate and fixed-delay methods are for simple, periodic execution, but the method that accepts a Trigger is much more flexible.

7.3.1. Trigger interface

The `Trigger` interface is essentially inspired by JSR-236, which, as of Spring 3.0, has not yet been officially implemented. The basic idea of the `Trigger` is that execution times may be determined based on past execution outcomes or even arbitrary conditions. If these determinations do take into account the outcome of the preceding execution, that information is available within a `TriggerContext`. The `Trigger` interface itself is quite simple:

```java
public interface Trigger {

    Date nextExecutionTime(TriggerContext triggerContext);

}
```

As you can see, the `TriggerContext` is the most important part. It encapsulates all of the relevant data, and is open for extension in the future if necessary. The `TriggerContext` is an interface (a `SimpleTriggerContext` implementation is used by default). Here you can see what methods are available for `Trigger` implementations.

```
public interface TriggerContext {

    Date lastScheduledExecutionTime();

    Date lastActualExecutionTime();

    Date lastCompletionTime();

}
```

7.3.2. Trigger implementations

Spring provides two implementations of the `Trigger` interface. The most interesting one is the `CronTrigger`. It enables the scheduling of tasks based on cron expressions. For example, the following task is being scheduled to run 15 minutes past each hour but only during the 9-to-5 "business hours" on weekdays.

```
scheduler.schedule(task, new CronTrigger("0 15 9-17 * * MON-FRI"));
```

The other out-of-the-box implementation is a `PeriodicTrigger` that accepts a fixed period, an optional initial delay value, and a boolean to indicate whether the period should be interpreted as a fixed-rate or a fixed-delay. Since the `TaskScheduler` interface already defines methods for scheduling tasks at a fixed-rate or with a fixed-delay, those methods should be used directly whenever possible. The value of the `PeriodicTrigger` implementation is that it can be used within components that rely on the `Trigger` abstraction. For example, it may be convenient to allow periodic triggers, cron-based triggers, and even custom trigger implementations to be used interchangeably. Such a component could take advantage of dependency injection so that such `Triggers` could be configured externally and therefore easily modified or extended.

7.3.3. TaskScheduler implementations

As with Spring's `TaskExecutor` abstraction, the primary benefit of the `TaskScheduler` is that code relying on scheduling behavior need not be coupled to a particular scheduler implementation. The flexibility this provides is particularly relevant when running within Application Server environments where threads should not be created directly by the application itself. For such cases, Spring provides a `TimerManagerTaskScheduler` that delegates to a CommonJ TimerManager instance, typically configured with a JNDI-lookup.

A simpler alternative, the `ThreadPoolTaskScheduler`, can be used whenever external thread management is not a requirement. Internally, it delegates to a `ScheduledExecutorService` instance. `ThreadPoolTaskScheduler` actually implements Spring's `TaskExecutor` interface as well, so that a single instance can be used for asynchronous execution *as soon as possible* as well as scheduled, and potentially recurring, executions.

7.4. Annotation Support for Scheduling and Asynchronous Execution

Spring provides annotation support for both task scheduling and asynchronous method execution.

7.4.1. Enable scheduling annotations

To enable support for `@Scheduled` and `@Async` annotations add `@EnableScheduling` and `@EnableAsync` to one of your `@Configuration` classes:

```
@Configuration
@EnableAsync
@EnableScheduling
public class AppConfig {
}
```

You are free to pick and choose the relevant annotations for your application. For example, if you only need support for `@Scheduled`, simply omit `@EnableAsync`. For more fine-grained control you can additionally implement the `SchedulingConfigurer` and/or `AsyncConfigurer` interfaces. See the javadocs for full details.

If you prefer XML configuration use the `<task:annotation-driven>` element.

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>
<task:executor id="myExecutor" pool-size="5"/>
<task:scheduler id="myScheduler" pool-size="10"/>
```

Notice with the above XML that an executor reference is provided for handling those tasks that correspond to methods with the `@Async` annotation, and the scheduler reference is provided for managing those methods annotated with `@Scheduled`.

> 🛈 The default advice mode for processing `@Async` annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile-time or load-time weaving.

7.4.2. The @Scheduled annotation

The `@Scheduled` annotation can be added to a method along with trigger metadata. For example, the following method would be invoked every 5 seconds with a fixed delay, meaning that the period will be measured from the completion time of each preceding invocation.

```
@Scheduled(fixedDelay=5000)
public void doSomething() {
    // something that should execute periodically
}
```

If a fixed rate execution is desired, simply change the property name specified within the annotation. The following would be executed every 5 seconds measured between the successive start times of each invocation.

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

For fixed-delay and fixed-rate tasks, an initial delay may be specified indicating the number of milliseconds to wait before the first execution of the method.

```
@Scheduled(initialDelay=1000, fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

If simple periodic scheduling is not expressive enough, then a cron expression may be provided. For example, the following will only execute on weekdays.

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

> 💡 You can additionally use the `zone` attribute to specify the time zone in which the cron expression will be resolved.

Notice that the methods to be scheduled must have void returns and must not expect any arguments. If the method needs to interact with other objects from the Application Context, then those would typically have been provided through dependency injection.

As of Spring Framework 4.3, @Scheduled methods are supported on beans of any scope.

Make sure that you are not initializing multiple instances of the same @Scheduled annotation class at runtime, unless you do want to schedule callbacks to each such instance. Related to this, make sure that you do not use @Configurable on bean classes which are annotated with @Scheduled and registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the @Configurable aspect, with the consequence of each @Scheduled method being invoked twice.

7.4.3. The @Async annotation

The @Async annotation can be provided on a method so that invocation of that method will occur asynchronously. In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring TaskExecutor. In the simplest case, the annotation may be applied to a void-returning method.

```java
@Async
void doSomething() {
    // this will be executed asynchronously
}
```

Unlike the methods annotated with the @Scheduled annotation, these methods can expect arguments, because they will be invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container. For example, the following is a legitimate application of the @Async annotation.

```java
@Async
void doSomething(String s) {
    // this will be executed asynchronously
}
```

Even methods that return a value can be invoked asynchronously. However, such methods are required to have a Future typed return value. This still provides the benefit of asynchronous execution so that the caller can perform other tasks prior to calling get() on that Future.

```java
@Async
Future<String> returnSomething(int i) {
    // this will be executed asynchronously
}
```

> 💡 `@Async` methods may not only declare a regular `java.util.concurrent.Future` return type but also Spring's `org.springframework.util.concurrent.ListenableFuture` or, as of Spring 4.2, JDK 8's `java.util.concurrent.CompletableFuture`: for richer interaction with the asynchronous task and for immediate composition with further processing steps.

`@Async` can not be used in conjunction with lifecycle callbacks such as `@PostConstruct`. To asynchronously initialize Spring beans you currently have to use a separate initializing Spring bean that invokes the `@Async` annotated method on the target then.

```java
public class SampleBeanImpl implements SampleBean {

    @Async
    void doSomething() {
        // ...
    }

}

public class SampleBeanInitializer {

    private final SampleBean bean;

    public SampleBeanInitializer(SampleBean bean) {
        this.bean = bean;
    }

    @PostConstruct
    public void initialize() {
        bean.doSomething();
    }

}
```

> ℹ️ There is no direct XML equivalent for `@Async` since such methods should be designed for asynchronous execution in the first place, not externally re-declared to be async. However, you may manually set up Spring's `AsyncExecutionInterceptor` with Spring AOP, in combination with a custom pointcut.

7.4.4. Executor qualification with @Async

By default when specifying `@Async` on a method, the executor that will be used is the one supplied to the 'annotation-driven' element as described above. However, the `value` attribute of the `@Async` annotation can be used when needing to indicate that an executor other than the default should be used when executing a given method.

```
@Async("otherExecutor")
void doSomething(String s) {
    // this will be executed asynchronously by "otherExecutor"
}
```

In this case, "otherExecutor" may be the name of any `Executor` bean in the Spring container, or may be the name of a *qualifier* associated with any `Executor`, e.g. as specified with the `<qualifier>` element or Spring's `@Qualifier` annotation.

7.4.5. Exception management with @Async

When an `@Async` method has a `Future` typed return value, it is easy to manage an exception that was thrown during the method execution as this exception will be thrown when calling `get` on the `Future` result. With a void return type however, the exception is uncaught and cannot be transmitted. For those cases, an `AsyncUncaughtExceptionHandler` can be provided to handle such exceptions.

```
public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler
{

    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object... params)
    {
        // handle exception
    }
}
```

By default, the exception is simply logged. A custom `AsyncUncaughtExceptionHandler` can be defined *via* `AsyncConfigurer` or the `task:annotation-driven` XML element.

7.5. The task namespace

Beginning with Spring 3.0, there is an XML namespace for configuring `TaskExecutor` and `TaskScheduler` instances. It also provides a convenient way to configure tasks to be scheduled with a trigger.

7.5.1. The 'scheduler' element

The following element will create a `ThreadPoolTaskScheduler` instance with the specified thread pool size.

```
<task:scheduler id="scheduler" pool-size="10"/>
```

The value provided for the 'id' attribute will be used as the prefix for thread names within the pool. The 'scheduler' element is relatively straightforward. If you do not provide a 'pool-size' attribute,

the default thread pool will only have a single thread. There are no other configuration options for the scheduler.

7.5.2. The 'executor' element

The following will create a `ThreadPoolTaskExecutor` instance:

```
<task:executor id="executor" pool-size="10"/>
```

As with the scheduler above, the value provided for the 'id' attribute will be used as the prefix for thread names within the pool. As far as the pool size is concerned, the 'executor' element supports more configuration options than the 'scheduler' element. For one thing, the thread pool for a `ThreadPoolTaskExecutor` is itself more configurable. Rather than just a single size, an executor's thread pool may have different values for the *core* and the *max* size. If a single value is provided then the executor will have a fixed-size thread pool (the core and max sizes are the same). However, the 'executor' element's 'pool-size' attribute also accepts a range in the form of "min-max".

```
<task:executor
        id="executorWithPoolSizeRange"
        pool-size="5-25"
        queue-capacity="100"/>
```

As you can see from that configuration, a 'queue-capacity' value has also been provided. The configuration of the thread pool should also be considered in light of the executor's queue capacity. For the full description of the relationship between pool size and queue capacity, consult the documentation for ThreadPoolExecutor. The main idea is that when a task is submitted, the executor will first try to use a free thread if the number of active threads is currently less than the core size. If the core size has been reached, then the task will be added to the queue as long as its capacity has not yet been reached. Only then, if the queue's capacity *has* been reached, will the executor create a new thread beyond the core size. If the max size has also been reached, then the executor will reject the task.

By default, the queue is *unbounded*, but this is rarely the desired configuration, because it can lead to `OutOfMemoryErrors` if enough tasks are added to that queue while all pool threads are busy. Furthermore, if the queue is unbounded, then the max size has no effect at all. Since the executor will always try the queue before creating a new thread beyond the core size, a queue must have a finite capacity for the thread pool to grow beyond the core size (this is why a *fixed size* pool is the only sensible case when using an unbounded queue).

In a moment, we will review the effects of the keep-alive setting which adds yet another factor to consider when providing a pool size configuration. First, let's consider the case, as mentioned above, when a task is rejected. By default, when a task is rejected, a thread pool executor will throw a `TaskRejectedException`. However, the rejection policy is actually configurable. The exception is thrown when using the default rejection policy which is the `AbortPolicy` implementation. For applications where some tasks can be skipped under heavy load, either the `DiscardPolicy` or `DiscardOldestPolicy` may be configured instead. Another option that works well for applications

that need to throttle the submitted tasks under heavy load is the `CallerRunsPolicy`. Instead of throwing an exception or discarding tasks, that policy will simply force the thread that is calling the submit method to run the task itself. The idea is that such a caller will be busy while running that task and not able to submit other tasks immediately. Therefore it provides a simple way to throttle the incoming load while maintaining the limits of the thread pool and queue. Typically this allows the executor to "catch up" on the tasks it is handling and thereby frees up some capacity on the queue, in the pool, or both. Any of these options can be chosen from an enumeration of values available for the 'rejection-policy' attribute on the 'executor' element.

```xml
<task:executor
        id="executorWithCallerRunsPolicy"
        pool-size="5-25"
        queue-capacity="100"
        rejection-policy="CALLER_RUNS"/>
```

Finally, the `keep-alive` setting determines the time limit (in seconds) for which threads may remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads will get terminated. A time value of zero will cause excess threads to terminate immediately after executing a task without remaining follow-up work in the task queue.

```xml
<task:executor
        id="executorWithKeepAlive"
        pool-size="5-25"
        keep-alive="120"/>
```

7.5.3. The 'scheduled-tasks' element

The most powerful feature of Spring's task namespace is the support for configuring tasks to be scheduled within a Spring Application Context. This follows an approach similar to other "method-invokers" in Spring, such as that provided by the JMS namespace for configuring Message-driven POJOs. Basically a "ref" attribute can point to any Spring-managed object, and the "method" attribute provides the name of a method to be invoked on that object. Here is a simple example.

```xml
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

As you can see, the scheduler is referenced by the outer element, and each individual task includes the configuration of its trigger metadata. In the preceding example, that metadata defines a periodic trigger with a fixed delay indicating the number of milliseconds to wait after each task execution has completed. Another option is 'fixed-rate', indicating how often the method should be executed regardless of how long any previous execution takes. Additionally, for both fixed-delay and fixed-rate tasks an 'initial-delay' parameter may be specified indicating the number of

milliseconds to wait before the first execution of the method. For more control, a "cron" attribute may be provided instead. Here is an example demonstrating these other options.

```xml
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-delay=
"1000"/>
    <task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
    <task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

7.6. Using the Quartz Scheduler

Quartz uses `Trigger`, `Job` and `JobDetail` objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://quartz-scheduler.org. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

7.6.1. Using the JobDetailFactoryBean

Quartz `JobDetail` objects contain all information needed to run a job. Spring provides a `JobDetailFactoryBean` which provides bean-style properties for XML configuration purposes. Let's have a look at an example:

```xml
<bean name="exampleJob" class=
"org.springframework.scheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass" value="example.ExampleJob"/>
    <property name="jobDataAsMap">
        <map>
            <entry key="timeout" value="5"/>
        </map>
    </property>
</bean>
```

The job detail configuration has all information it needs to run the job (`ExampleJob`). The timeout is specified in the job data map. The job data map is available through the `JobExecutionContext` (passed to you at execution time), but the `JobDetail` also gets its properties from the job data mapped to properties of the job instance. So in this case, if the `ExampleJob` contains a bean property named `timeout`, the `JobDetail` will have it applied automatically:

```
package example;

public class ExampleJob extends QuartzJobBean {

    private int timeout;

    /**
     * Setter called after the ExampleJob is instantiated
     * with the value from the JobDetailFactoryBean (5)
     */
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext ctx) throws
JobExecutionException {
        // do the actual work
    }

}
```

All additional properties from the job data map are of course available to you as well.

> 🛈 Using the `name` and `group` properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the bean name of the `JobDetailFactoryBean` (in the example above, this is `exampleJob`).

7.6.2. Using the MethodInvokingJobDetailFactoryBean

Often you just need to invoke a method on a specific object. Using the `MethodInvokingJobDetailFactoryBean` you can do exactly this:

```
<bean id="jobDetail" class=
"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
</bean>
```

The above example will result in the `doIt` method being called on the `exampleBusinessObject` method (see below):

```java
public class ExampleBusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

```xml
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

Using the `MethodInvokingJobDetailFactoryBean`, you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same `JobDetail`, it might be possible that before the first job has finished, the second one will start. If `JobDetail` classes implement the `Stateful` interface, this won't happen. The second job will not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryBean` non-concurrent, set the `concurrent` flag to `false`.

```xml
<bean id="jobDetail" class=
"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
    <property name="concurrent" value="false"/>
</bean>
```

> By default, jobs will run in a concurrent fashion.

7.6.3. Wiring up jobs using triggers and the SchedulerFactoryBean

We've created job details and jobs. We've also reviewed the convenience bean that allows you to invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a `SchedulerFactoryBean`. Several triggers are available within Quartz and Spring offers two Quartz `FactoryBean` implementations with convenient defaults: `CronTriggerFactoryBean` and `SimpleTriggerFactoryBean`.

Triggers need to be scheduled. Spring offers a `SchedulerFactoryBean` that exposes triggers to be set as properties. `SchedulerFactoryBean` schedules the actual jobs with those triggers.

Find below a couple of examples:

```xml
<bean id="simpleTrigger" class=
"org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail"/>
    <!-- 10 seconds -->
    <property name="startDelay" value="10000"/>
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000"/>
</bean>

<bean id="cronTrigger" class=
"org.springframework.scheduling.quartz.CronTriggerFactoryBean">
    <property name="jobDetail" ref="exampleJob"/>
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?"/>
</bean>
```

Now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryBean`:

```xml
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger"/>
            <ref bean="simpleTrigger"/>
        </list>
    </property>
</bean>
```

More properties are available for the `SchedulerFactoryBean` for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the `SchedulerFactoryBean` javadocs for more information.

Chapter 8. Cache Abstraction

8.1. Introduction

Since version 3.1, Spring Framework provides support for transparently adding caching into an existing Spring application. Similar to the transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

As from Spring 4.1, the cache abstraction has been significantly improved with the support of JSR-107 annotations and more customization options.

8.2. Understanding the cache abstraction

Cache vs Buffer

The terms "buffer" and "cache" tend to be used interchangeably; note however they represent different things. A buffer is used traditionally as an intermediate temporary store for data between a fast and a slow entity. As one party would have to *wait* for the other affecting performance, the buffer alleviates this by allowing entire blocks of data to move at once rather then in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are *visible* to at least one party which is aware of it.

A cache on the other hand by definition is hidden and neither party is aware that caching occurs.It as well improves performance but does that by allowing the same data to be read multiple times in a fast fashion.

A further explanation of the differences between two can be found here.

At its core, the abstraction applies caching to Java methods, reducing thus the number of executions based on the information available in the cache. That is, each time a *targeted* method is invoked, the abstraction will apply a caching behavior checking whether the method has been already executed for the given arguments. If it has, then the cached result is returned without having to execute the actual method; if it has not, then method is executed, the result cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU or IO bound) can be executed only once for a given set of parameters and the result reused without having to actually execute the method again. The caching logic is applied transparently without any interference to the invoker.

> ❗ Obviously this approach works only for methods that are guaranteed to return the same output (result) for a given input (or arguments) no matter how many times it is being executed.

Other cache-related operations are provided by the abstraction such as the ability to update the content of the cache or remove one of all entries. These are useful if the cache deals with data that can change during the course of the application.

Just like other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of an actual storage to store the cache data - that is, the abstraction frees the developer from having to write the caching logic but does not provide the actual stores. This abstraction is materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

There are a few implementations of that abstraction available out of the box: JDK `java.util.concurrent.ConcurrentMap` based caches, Ehcache 2.x, Gemfire cache, Caffeine and JSR-107 compliant caches (e.g. Ehcache 3.x). See Plugging-in different back-end caches for more information on plugging in other cache stores/providers.

> ⊗ The caching abstraction has no special handling of multi-threaded and multi-process environments as such features are handled by the cache implementation. .

If you have a multi-process environment (i.e. an application deployed on several nodes), you will need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes may be enough but if you change the data during the course of the application, you may need to enable other propagation mechanisms.

Caching a particular item is a direct equivalent of the typical get-if-not-found-then- proceed-and-put-eventually code blocks found with programmatic cache interaction: no locks are applied and several threads may try to load the same item concurrently. The same applies to eviction: if several threads are trying to update or evict data concurrently, you may use stale data. Certain cache providers offer advanced features in that area, refer to the documentation of the cache provider that you are using for more details.

To use the cache abstraction, the developer needs to take care of two aspects:

- caching declaration - identify the methods that need to be cached and their policy
- cache configuration - the backing cache where the data is stored and read from

8.3. Declarative annotation-based caching

For caching declaration, the abstraction provides a set of Java annotations:

- `@Cacheable` triggers cache population
- `@CacheEvict` triggers cache eviction
- `@CachePut` updates the cache without interfering with the method execution
- `@Caching` regroups multiple cache operations to be applied on a method
- `@CacheConfig` shares some common cache-related settings at class-level

Let us take a closer look at each annotation:

8.3.1. @Cacheable annotation

As the name implies, `@Cacheable` is used to demarcate methods that are cacheable - that is, methods for whom the result is stored into the cache so on subsequent invocations (with the same

arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

In the snippet above, the method `findBook` is associated with the cache named `books`. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation allows multiple names to be specified so that more than one cache are being used. In this case, each of the caches will be checked before executing the method - if at least one cache is hit, then the associated value will be returned:

> All the other caches that do not contain the value will be updated as well even though the cached method was not actually executed.

```
@Cacheable({"books", "isbns"})
public Book findBook(ISBN isbn) {...}
```

Default Key Generation

Since caches are essentially key-value stores, each invocation of a cached method needs to be translated into a suitable key for cache access. Out of the box, the caching abstraction uses a simple `KeyGenerator` based on the following algorithm:

- If no params are given, return `SimpleKey.EMPTY`.
- If only one param is given, return that instance.
- If more the one param is given, return a `SimpleKey` containing all parameters.

This approach works well for most use-cases; As long as parameters have *natural keys* and implement valid `hashCode()` and `equals()` methods. If that is not the case then the strategy needs to be changed.

To provide a different *default* key generator, one needs to implement the `org.springframework.cache.interceptor.KeyGenerator` interface.

> The default key generation strategy changed with the release of Spring 4.0. Earlier versions of Spring used a key generation strategy that, for multiple key parameters, only considered the `hashCode()` of parameters and not `equals()`; this could cause unexpected key collisions (see SPR-10237 for background). The new 'SimpleKeyGenerator' uses a compound key for such scenarios.
>
> If you want to keep using the previous key strategy, you can configure the deprecated `org.springframework.cache.interceptor.DefaultKeyGenerator` class or create a custom hash-based 'KeyGenerator' implementation.

Custom Key Generation Declaration

Since caching is generic, it is quite likely the target methods have various signatures that cannot be simply mapped on top of the cache structure. This tends to become obvious when the target method has multiple arguments out of which only some are suitable for caching (while the rest are used only by the method logic). For example:

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

At first glance, while the two boolean arguments influence the way the book is found, they are no use for the cache. Further more what if only one of the two is important while the other is not?

For such cases, the @Cacheable annotation allows the user to specify how the key is generated through its key attribute. The developer can use SpEL to pick the arguments of interest (or their nested properties), perform operations or even invoke arbitrary methods without having to write any code or implement any interface. This is the recommended approach over the default generator since methods tend to be quite different in signatures as the code base grows; while the default strategy might work for some methods, it rarely does for all methods.

Below are some examples of various SpEL declarations - if you are not familiar with it, do yourself a favor and read Spring Expression Language:

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The snippets above show how easy it is to select a certain argument, one of its properties or even an arbitrary (static) method.

If the algorithm responsible to generate the key is too specific or if it needs to be shared, you may define a custom keyGenerator on the operation. To do this, specify the name of the KeyGenerator bean implementation to use:

```
@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

> ℹ️ The key and keyGenerator parameters are mutually exclusive and an operation specifying both will result in an exception.

Default Cache Resolution

Out of the box, the caching abstraction uses a simple `CacheResolver` that retrieves the cache(s) defined at the operation level using the configured `CacheManager`.

To provide a different *default* cache resolver, one needs to implement the `org.springframework.cache.interceptor.CacheResolver` interface.

Custom cache resolution

The default cache resolution fits well for applications working with a single `CacheManager` and with no complex cache resolution requirements.

For applications working with several cache managers, it is possible to set the `cacheManager` to use per operation:

```
@Cacheable(cacheNames="books", <strong>cacheManager="anotherCacheManager"</strong>)
public Book findBook(ISBN isbn) {...}
```

It is also possible to replace the `CacheResolver` entirely in a similar fashion as for key generation. The resolution is requested for every cache operation, giving a chance to the implementation to actually resolve the cache(s) to use based on runtime arguments:

```
@Cacheable(<strong>cacheResolver="runtimeCacheResolver"</strong>)
public Book findBook(ISBN isbn) {...}
```

> Since Spring 4.1, the `value` attribute of the cache annotations are no longer mandatory since this particular information can be provided by the `CacheResolver` regardless of the content of the annotation.
>
> Similarly to `key` and `keyGenerator`, the `cacheManager` and `cacheResolver` parameters are mutually exclusive and an operation specifying both will result in an exception as a custom `CacheManager` will be ignored by the `CacheResolver` implementation. This is probably not what you expect.

Synchronized caching

In a multi-threaded environment, certain operations might be concurrently invoked for the same argument (typically on startup). By default, the cache abstraction does not lock anything and the same value may be computed several times, defeating the purpose of caching.

For those particular cases, the `sync` attribute can be used to instruct the underlying cache provider to *lock* the cache entry while the value is being computed. As a result, only one thread will be busy computing the value while the others are blocked until the entry is updated in the cache.

```
@Cacheable(cacheNames="foos", sync=true)
public Foo executeExpensiveOperation(String id) {...}
```

> ℹ️ This is an optional feature and your favorite cache library may not support it. All
> `CacheManager` implementations provided by the core framework support it. Check
> the documentation of your cache provider for more details.

Conditional caching

Sometimes, a method might not be suitable for caching all the time (for example, it might depend
on the given arguments). The cache annotations support such functionality through the `condition`
parameter which takes a `SpEL` expression that is evaluated to either `true` or `false`. If `true`, the
method is cached - if not, it behaves as if the method is not cached, that is executed every time no
matter what values are in the cache or what arguments are used. A quick example - the following
method will be cached only if the argument `name` has a length shorter than 32:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32")
public Book findBook(String name)
```

In addition the `condition` parameter, the `unless` parameter can be used to veto the adding of a value
to the cache. Unlike `condition`, `unless` expressions are evaluated *after* the method has been called.
Expanding on the previous example - perhaps we only want to cache paperback books:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32", unless=
"#result.hardback")
public Book findBook(String name)
```

The cache abstraction supports `java.util.Optional`, using its content as cached value only if it
present. `#result` always refers to the business entity and never on a supported wrapper so the
previous example can be rewritten as follows:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32", unless=
"#result?.hardback")
public Optional<Book> findBook(String name)
```

Note that `result` still refers to `Book` and not `Optional`. As it might be `null`, we should use the safe
navigation operator.

Available caching SpEL evaluation context

Each `SpEL` expression evaluates again a dedicated `context`. In addition to the build in parameters,
the framework provides dedicated caching related metadata such as the argument names. The next
table lists the items made available to the context so one can use them for key and conditional
computations:

Table 10. Cache SpEL available metadata

Name	Location	Description	Example
methodName	root object	The name of the method being invoked	`#root.methodName`
method	root object	The method being invoked	`#root.method.name`
target	root object	The target object being invoked	`#root.target`
targetClass	root object	The class of the target being invoked	`#root.targetClass`
args	root object	The arguments (as array) used for invoking the target	`#root.args[0]`
caches	root object	Collection of caches against which the current method is executed	`#root.caches[0].name`
argument name	evaluation context	Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the `#a<#arg>` where *#arg* stands for the argument index (starting from 0).	`#iban` or `#a0` (one can also use `#p0` or `#p<#arg>` notation as an alias).
result	evaluation context	The result of the method call (the value to be cached). Only available in `unless` expressions, `cache put` expressions (to compute the `key`), or `cache evict` expressions (when `beforeInvocation` is `false`). For supported wrappers such as `Optional`, `#result` refers to the actual object, not the wrapper.	`#result`

8.3.2. @CachePut annotation

For cases where the cache needs to be updated without interfering with the method execution, one can use the `@CachePut` annotation. That is, the method will always be executed and its result placed

into the cache (according to the `@CachePut` options). It supports the same options as `@Cacheable` and should be used for cache population rather than method flow optimization:

```
@CachePut(cacheNames="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

> ❗ Note that using `@CachePut` and `@Cacheable` annotations on the same method is generally strongly discouraged because they have different behaviors. While the latter causes the method execution to be skipped by using the cache, the former forces the execution in order to execute a cache update. This leads to unexpected behavior and with the exception of specific corner-cases (such as annotations having conditions that exclude them from each other), such declaration should be avoided. Note also that such condition should not rely on the result object (i.e. the `#result` variable) as these are validated upfront to confirm the exclusion.

8.3.3. @CacheEvict annotation

The cache abstraction allows not just population of a cache store but also eviction. This process is useful for removing stale or unused data from the cache. Opposed to `@Cacheable`, annotation `@CacheEvict` demarcates methods that perform cache *eviction*, that is methods that act as triggers for removing data from the cache. Just like its sibling, `@CacheEvict` requires specifying one (or multiple) caches that are affected by the action, allows a custom cache and key resolution or a condition to be specified but in addition, features an extra parameter `allEntries` which indicates whether a cache-wide eviction needs to be performed rather then just an entry one (based on the key):

```
@CacheEvict(cacheNames="books", <strong>allEntries=true</strong>)
public void loadBooks(InputStream batch)
```

This option comes in handy when an entire cache region needs to be cleared out - rather then evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation as shown above. Note that the framework will ignore any key specified in this scenario as it does not apply (the entire cache is evicted not just one entry).

One can also indicate whether the eviction should occur after (the default) or before the method executes through the `beforeInvocation` attribute. The former provides the same semantics as the rest of the annotations - once the method completes successfully, an action (in this case eviction) on the cache is executed. If the method does not execute (as it might be cached) or an exception is thrown, the eviction does not occur. The latter (`beforeInvocation=true`) causes the eviction to occur always, before the method is invoked - this is useful in cases where the eviction does not need to be tied to the method outcome.

It is important to note that void methods can be used with `@CacheEvict` - as the methods act as triggers, the return values are ignored (as they don't interact with the cache) - this is not the case with `@Cacheable` which adds/updates data into the cache and thus requires a result.

8.3.4. @Caching annotation

There are cases when multiple annotations of the same type, such as @CacheEvict or @CachePut need to be specified, for example because the condition or the key expression is different between different caches. @Caching allows multiple nested @Cacheable, @CachePut and @CacheEvict to be used on the same method:

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key=
"#p0") })
public Book importBooks(String deposit, Date date)
```

8.3.5. @CacheConfig annotation

So far we have seen that caching operations offered many customization options and these can be set on an operation basis. However, some of the customization options can be tedious to configure if they apply to all operations of the class. For instance, specifying the name of the cache to use for every cache operation of the class could be replaced by a single class-level definition. This is where @CacheConfig comes into play.

```
<strong>@CacheConfig("books")</strong>
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) {...}
}
```

@CacheConfig is a class-level annotation that allows to share the cache names, the custom KeyGenerator, the custom CacheManager and finally the custom CacheResolver. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization will always override a customization set on @CacheConfig. This gives therefore three levels of customizations per cache operation:

- Globally configured, available for CacheManager, KeyGenerator
- At class level, using @CacheConfig
- At the operation level

8.3.6. Enable caching annotations

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation @EnableCaching to one of your @Configuration classes:

```
@Configuration
@EnableCaching
public class AppConfig {
}
```

Alternatively for XML configuration use the `cache:annotation-driven` element:

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

        <cache:annotation-driven />

</beans>
```

Both the `cache:annotation-driven` element and `@EnableCaching` annotation allow various options to be specified that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of `@Transactional`:

> ℹ️ The default advice mode for processing caching annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile-time or load-time weaving.

> ℹ️ Advanced customizations using Java config require to implement `CachingConfigurer`: Please refer to the javadoc for more details.

Table 11. Cache annotation settings

XML Attribute	Annotation Attribute	Default	Description
`cache-manager`	N/A (See `CachingConfigurer` javadocs)	cacheManager	Name of cache manager to use. A default `CacheResolver` will be initialized behind the scenes with this cache manager (or `cacheManager`if not set). For more fine-grained management of the cache resolution, consider setting the 'cache-resolver' attribute.

XML Attribute	Annotation Attribute	Default	Description
`cache-resolver`	N/A (See `CachingConfigurer` javadocs)	A `SimpleCacheResolver` using the configured `cacheManager`.	The bean name of the CacheResolver that is to be used to resolve the backing caches. This attribute is not required, and only needs to be specified as an alternative to the 'cache-manager' attribute.
`key-generator`	N/A (See `CachingConfigurer` javadocs)	`SimpleKeyGenerator`	Name of the custom key generator to use.
`error-handler`	N/A (See `CachingConfigurer` javadocs)	`SimpleCacheErrorHandler`	Name of the custom cache error handler to use. By default, any exception throw during a cache related operations are thrown back at the client.
`mode`	`mode`	proxy	The default mode "proxy" processes annotated beans to be proxied using Spring's AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring's AspectJ caching aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See Spring configuration for details on how to set up load-time weaving.)
`proxy-target-class`	`proxyTargetClass`	false	Applies to proxy mode only. Controls what type of caching proxies are created for classes annotated with the `@Cacheable` or `@CacheEvict` annotations. If the `proxy-target-class` attribute is set to `true`, then class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, then standard JDK interface-based proxies are created. (See Proxying mechanisms for a detailed examination of the different proxy types.)
`order`	`order`	Ordered.LOWEST_PRECEDENCE	Defines the order of the cache advice that is applied to beans annotated with `@Cacheable` or `@CacheEvict`. (For more information about the rules related to ordering of AOP advice, see Advice ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

<cache:annotation-driven/> only looks for @Cacheable/@CachePut/@CacheEvict/@Caching on beans in the same application context it is defined in. This means that, if you put <cache:annotation-driven/> in a WebApplicationContext for a DispatcherServlet, it only checks for beans in your controllers, and not your services. See the MVC section for more information.

Method visibility and cache annotations

When using proxies, you should apply the cache annotations only to methods with *public* visibility. If you do annotate protected, private or package-visible methods with these annotations, no error is raised, but the annotated method does not exhibit the configured caching settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods as it changes the bytecode itself.

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Cache* annotation, as opposed to annotating interfaces. You certainly can place the @Cache* annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are *not inherited from interfaces* means that if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the caching settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a caching proxy, which would be decidedly *bad*.

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual caching at runtime even if the invoked method is marked with @Cacheable - considering using the aspectj mode in this case. Also, the proxy must be fully initialized to provide the expected behaviour so you should not rely on this feature in your initialization code, i.e. @PostConstruct.

8.3.7. Using custom annotations

Custom annotation and AspectJ

This feature only works out-of-the-box with the proxy-based approach but can be enabled with a bit of extra effort using AspectJ.

The spring-aspects module defines an aspect for the standard annotations only. If you have defined your own annotations, you also need to define an aspect for those. Check AnnotationCacheAspect for an example.

The caching abstraction allows you to use your own annotations to identify what method triggers cache population or eviction. This is quite handy as a template mechanism as it eliminates the need

to duplicate cache annotation declarations (especially useful if the key or condition are specified) or if the foreign imports (`org.springframework`) are not allowed in your code base. Similar to the rest of the stereotype annotations, `@Cacheable`, `@CachePut`, `@CacheEvict` and `@CacheConfig` can be used as meta-annotations, that is annotations that can annotate other annotations. To wit, let us replace a common `@Cacheable` declaration with our own, custom annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(cacheNames="books", key="#isbn")
public @interface SlowService {
}
```

Above, we have defined our own `SlowService` annotation which itself is annotated with `@Cacheable` - now we can replace the following code:

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

with:

```
@SlowService
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

Even though `@SlowService` is not a Spring annotation, the container automatically picks up its declaration at runtime and understands its meaning. Note that as mentioned above, the annotation-driven behavior needs to be enabled.

8.4. JCache (JSR-107) annotations

Since the Spring Framework 4.1, the caching abstraction fully supports the JCache standard annotations: these are `@CacheResult`, `@CachePut`, `@CacheRemove` and `@CacheRemoveAll` as well as the `@CacheDefaults`, `@CacheKey` and `@CacheValue` companions. These annotations can be used right the way without migrating your cache store to JSR-107: the internal implementation uses Spring's caching abstraction and provides default `CacheResolver` and `KeyGenerator` implementations that are compliant with the specification. In other words, if you are already using Spring's caching abstraction, you can switch to these standard annotations without changing your cache storage (or configuration, for that matter).

8.4.1. Feature summary

For those who are familiar with Spring's caching annotations, the following table describes the main differences between the Spring annotations and the JSR-107 counterpart:

Table 12. Spring vs. JSR-107 caching annotations

Spring	JSR-107	Remark
@Cacheable	@CacheResult	Fairly similar. @CacheResult can cache specific exceptions and force the execution of the method regardless of the content of the cache.
@CachePut	@CachePut	While Spring updates the cache with the result of the method invocation, JCache requires to pass it as an argument that is annotated with @CacheValue. Due to this difference, JCache allows to update the cache before or after the actual method invocation.
@CacheEvict	@CacheRemove	Fairly similar. @CacheRemove supports a conditional evict in case the method invocation results in an exception.
@CacheEvict(allEntries=true)	@CacheRemoveAll	See @CacheRemove.
@CacheConfig	@CacheDefaults	Allows to configure the same concepts, in a similar fashion.

JCache has the notion of `javax.cache.annotation.CacheResolver` that is identical to the Spring's `CacheResolver` interface, except that JCache only supports a single cache. By default, a simple implementation retrieves the cache to use based on the name declared on the annotation. It should be noted that if no cache name is specified on the annotation, a default is automatically generated, check the javadoc of `@CacheResult#cacheName()` for more information.

`CacheResolver` instances are retrieved by a `CacheResolverFactory`. It is possible to customize the factory per cache operation:

```
@CacheResult(cacheNames="books", <strong>cacheResolverFactory=MyCacheResolverFactory
.class</strong>)
public Book findBook(ISBN isbn)
```

> ℹ️ For all referenced *classes*, Spring tries to locate a bean with the given type. If more than one match exists, a new instance is created and can use the regular bean lifecycle callbacks such as dependency injection.

Keys are generated by a `javax.cache.annotation.CacheKeyGenerator` that serves the same purpose as Spring's `KeyGenerator`. By default, all method arguments are taken into account unless at least one parameter is annotated with `@CacheKey`. This is similar to Spring's custom key generation declaration. For instance these are identical operations, one using Spring's abstraction and the other with JCache:

```
@Cacheable(cacheNames="books", <strong>key="#isbn"</strong>)
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")
public Book findBook(<strong>@CacheKey</strong> ISBN isbn, boolean checkWarehouse,
boolean includeUsed)
```

The `CacheKeyResolver` to use can also be specified on the operation, in a similar fashion as the `CacheResolverFactory`.

JCache can manage exceptions thrown by annotated methods: this can prevent an update of the cache but it can also cache the exception as an indicator of the failure instead of calling the method again. Let's assume that `InvalidIsbnNotFoundException` is thrown if the structure of the ISBN is invalid. This is a permanent failure, no book could ever be retrieved with such parameter. The following caches the exception so that further calls with the same, invalid ISBN, throws the cached exception directly instead of invoking the method again.

```
@CacheResult(cacheName="books", <strong>exceptionCacheName="failures"</strong>
            <strong>cachedExceptions = InvalidIsbnNotFoundException.class</strong>)
public Book findBook(ISBN isbn)
```

8.4.2. Enabling JSR-107 support

Nothing specific needs to be done to enable the JSR-107 support alongside Spring's declarative annotation support. Both `@EnableCaching` and the `cache:annotation-driven` element will enable automatically the JCache support if both the JSR-107 API and the `spring-context-support` module are present in the classpath.

> Depending of your use case, the choice is basically yours. You can even mix and match services using the JSR-107 API and others using Spring's own annotations. Be aware however that if these services are impacting the same caches, a consistent and identical key generation implementation should be used.

8.5. Declarative XML-based caching

If annotations are not an option (no access to the sources or no external code), one can use XML for declarative caching. So instead of annotating the methods for caching, one specifies the target method and the caching directives externally (similar to the declarative transaction management advice). The previous example can be translated into:

```xml
<!-- the service we want to make cacheable -->
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#isbn"/>
        <cache:cache-evict method="loadBooks" all-entries="true"/>
    </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->
<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(*
x.y.BookService.*(..))"/>
</aop:config>

<!-- cache manager definition omitted -->
```

In the configuration above, the `bookService` is made cacheable. The caching semantics to apply are encapsulated in the `cache:advice` definition which instructs method `findBooks` to be used for putting data into the cache while method `loadBooks` for evicting data. Both definitions are working against the `books` cache.

The `aop:config` definition applies the cache advice to the appropriate points in the program by using the AspectJ pointcut expression (more information is available in Aspect Oriented Programming with Spring). In the example above, all methods from the `BookService` are considered and the cache advice applied to them.

The declarative XML caching supports all of the annotation-based model so moving between the two should be fairly easy - further more both can be used inside the same application. The XML based approach does not touch the target code however it is inherently more verbose; when dealing with classes with overloaded methods that are targeted for caching, identifying the proper methods does take an extra effort since the `method` argument is not a good discriminator - in these cases, the AspectJ pointcut can be used to cherry pick the target methods and apply the appropriate caching functionality. However through XML, it is easier to apply a package/group/interface-wide caching (again due to the AspectJ pointcut) and to create template-like definitions (as we did in the example above by defining the target cache through the `cache:definitions cache` attribute).

8.6. Configuring the cache storage

Out of the box, the cache abstraction provides several storage integration. To use them, one needs to simply declare an appropriate `CacheManager` - an entity that controls and manages `Cache`s and can be used to retrieve these for storage.

8.6.1. JDK ConcurrentMap-based Cache

The JDK-based `Cache` implementation resides under `org.springframework.cache.concurrent` package.

It allows one to use `ConcurrentHashMap` as a backing `Cache` store.

```xml
<!-- simple cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean class=
"org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="default
"/>
            <bean class=
"org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="books"/>
        </set>
    </property>
</bean>
```

The snippet above uses the `SimpleCacheManager` to create a `CacheManager` for the two nested `ConcurrentMapCache` instances named *default* and *books*. Note that the names are configured directly for each cache.

As the cache is created by the application, it is bound to its lifecycle, making it suitable for basic use cases, tests or simple applications. The cache scales well and is very fast but it does not provide any management or persistence capabilities nor eviction contracts.

8.6.2. Ehcache-based Cache

Ehcache 3.x is fully JSR-107 compliant and no dedicated support is required for it.

The Ehcache 2.x implementation is located under `org.springframework.cache.ehcache` package. Again, to use it, one simply needs to declare the appropriate `CacheManager`:

```xml
<bean id="cacheManager"
        class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-
ref="ehcache"/>

<!-- EhCache library setup -->
<bean id="ehcache"
        class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-
location="ehcache.xml"/>
```

This setup bootstraps the ehcache library inside Spring IoC (through the `ehcache` bean) which is then wired into the dedicated `CacheManager` implementation. Note the entire ehcache-specific configuration is read from `ehcache.xml`.

8.6.3. Caffeine Cache

Caffeine is a Java 8 rewrite of Guava's cache and its implementation is located under `org.springframework.cache.caffeine` package and provides access to several features of Caffeine.

Configuring a `CacheManager` that creates the cache on demand is straightforward:

```
<bean id="cacheManager"
        class="org.springframework.cache.caffeine.CaffeineCacheManager"/>
```

It is also possible to provide the caches to use explicitly. In that case, only those will be made available by the manager:

```
<bean id="cacheManager" class="
org.springframework.cache.caffeine.CaffeineCacheManager">
    <property name="caches">
        <set>
            <value>default</value>
            <value>books</value>
        </set>
    </property>
</bean>
```

The Caffeine `CacheManager` also supports customs `Caffeine` and `CacheLoader`. See the Caffeine documentation for more information about those.

8.6.4. GemFire-based Cache

GemFire is a memory-oriented/disk-backed, elastically scalable, continuously available, active (with built-in pattern-based subscription notifications), globally replicated database and provides fully-featured edge caching. For further information on how to use GemFire as a CacheManager (and more), please refer to the Spring Data GemFire reference documentation.

8.6.5. JSR-107 Cache

JSR-107 compliant caches can also be used by Spring's caching abstraction. The JCache implementation is located under `org.springframework.cache.jcache` package.

Again, to use it, one simply needs to declare the appropriate `CacheManager`:

```
<bean id="cacheManager"
        class="org.springframework.cache.jcache.JCacheCacheManager"
        p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup  -->
<bean id="jCacheManager" .../>
```

8.6.6. Dealing with caches without a backing store

Sometimes when switching environments or doing testing, one might have cache declarations without an actual backing cache configured. As this is an invalid configuration, at runtime an

exception will be thrown since the caching infrastructure is unable to find a suitable store. In situations like this, rather then removing the cache declarations (which can prove tedious), one can wire in a simple, dummy cache that performs no caching - that is, forces the cached methods to be executed every time:

```xml
<bean id="cacheManager" class="
org.springframework.cache.support.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache"/>
            <ref bean="gemfireCache"/>
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

The `CompositeCacheManager` above chains multiple `CacheManager`s and additionally, through the `fallbackToNoOpCache` flag, adds a *no op* cache that for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either `jdkCache` or `gemfireCache` (configured above) will be handled by the no op cache, which will not store any information causing the target method to be executed every time.

8.7. Plugging-in different back-end caches

Clearly there are plenty of caching products out there that can be used as a backing store. To plug them in, one needs to provide a `CacheManager` and `Cache` implementation since unfortunately there is no available standard that we can use instead. This may sound harder than it is since in practice, the classes tend to be simple [adapter](#)s that map the caching abstraction framework on top of the storage API as the `ehcache` classes can show. Most `CacheManager` classes can use the classes in `org.springframework.cache.support` package, such as `AbstractCacheManager` which takes care of the boiler-plate code leaving only the actual *mapping* to be completed. We hope that in time, the libraries that provide integration with Spring can fill in this small configuration gap.

8.8. How can I set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is... well, an abstraction not a cache implementation. The solution you are using might support various data policies and different topologies which other solutions do not (take for example the JDK `ConcurrentHashMap`) - exposing that in the cache abstraction would be useless simply because there would no backing support. Such functionality should be controlled directly through the backing cache, when configuring it or through its native API.

Chapter 9. Appendix

9.1. XML Schemas

This part of the appendix lists XML schemas related to integration technologies.

9.1.1. The jee schema

The `jee` tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the `jee` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jee` namespace are available to you.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <em>xmlns:jee="http://www.springframework.org/schema/jee"</em>
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        <em>http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee.xsd"</em>> <!-- bean definitions
here -->

</beans>
```

<jee:jndi-lookup/> (simple)

Before…

```xml
<bean id="<strong>dataSource</strong>"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="<strong>dataSource</strong>"/>
</bean>
```

After…

```
<jee:jndi-lookup id="<strong>dataSource</strong>" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="<strong>dataSource</strong>"/>
</bean>
```

<jee:jndi-lookup/> (with single JNDI environment setting)

Before…

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
        </props>
    </property>
</bean>
```

After…

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

<jee:jndi-lookup/> (with multiple JNDI environment settings)

Before…

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

After…

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties
format) -->
    <jee:environment>
        foo=bar
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

<jee:jndi-lookup/> (complex)

Before…

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultFoo"/>
    <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

After…

```
<jee:jndi-lookup id="simple"
        jndi-name="jdbc/MyDataSource"
        cache="true"
        resource-ref="true"
        lookup-on-startup="false"
        expected-type="com.myapp.DefaultFoo"
        proxy-interface="com.myapp.Foo"/>
```

<jee:local-slsb/> (simple)

The `<jee:local-slsb/>` tag configures a reference to an EJB Stateless SessionBean.

Before…

```
<bean id="simple"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

After…

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"/>
```

\<jee:local-slsb/\> (complex)

```
<bean id="complexLocalEjb"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
</bean>
```

After…

```
<jee:local-slsb id="complexLocalEjb"
        jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true">
```

\<jee:remote-slsb/\>

The `<jee:remote-slsb/>` tag configures a reference to a `remote` EJB Stateless SessionBean.

Before…

```
<bean id="complexRemoteEjb"
        class=
"org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/MyRemoteBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="homeInterface" value="com.foo.service.RentalService"/>
    <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

After…

```
<jee:remote-slsb id="complexRemoteEjb"
        jndi-name="ejb/MyRemoteBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true"
        home-interface="com.foo.service.RentalService"
        refresh-home-on-connect-failure="true">
```

9.1.2. The jms schema

The `jms` tags deal with configuring JMS-related beans such as Spring's [MessageListenerContainers](#). These tags are detailed in the section of the [JMS chapter](#) entitled [JMS namespace support](#). Please do consult that chapter for full details on this support and the `jms` tags themselves.

In the interest of completeness, to use the tags in the `jms` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `jms` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms.xsd"> <!-- bean definitions
here -->

</beans>
```

9.1.3. <context:mbean-export/>

This element is detailed in [Configuring annotation based MBean export](#).

9.1.4. The cache schema

The `cache` tags can be used to enable support for Spring's `@CacheEvict`, `@CachePut` and `@Caching` annotations. It it also supports declarative XML-based caching. See [Enable caching annotations](#) and [Declarative XML-based caching](#) for details.

To use the tags in the `cache` schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the `cache` namespace are available to you.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <em>xmlns:cache="http://www.springframework.org/schema/cache"</em>
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        <em>http://www.springframework.org/schema/cache
http://www.springframework.org/schema/cache/spring-cache.xsd"</em>> <!-- bean
definitions here -->

</beans>
```