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.

Remoting and Web Services

Spring provides support for remoting with various technologies. The remoting support eases the development of remote-enabled services, implemented via Java interfaces and objects as input and output. Currently, Spring supports the following remoting technologies:

- Remote Method Invocation (RMI): Through the use of `RmiProxyFactoryBean` and `RmiServiceExporter`, Spring supports both traditional RMI (with `java.rmi.Remote` interfaces and `java.rmi.RemoteException`) and transparent remoting through RMI invokers (with any Java interface).
- Spring HTTP Invoker: Spring provides a special remoting strategy that allows for Java serialization though HTTP, supporting any Java interface (as the RMI invoker does). The corresponding support classes are `HttpInvokerProxyFactoryBean` and `HttpInvokerServiceExporter`.
- Hessian: By using Spring's `HessianProxyFactoryBean` and the `HessianServiceExporter`, you can transparently expose your services through the lightweight binary HTTP-based protocol provided by Caucho.
- Java Web Services: Spring provides remoting support for web services through JAX-WS.
- JMS: Remoting via JMS as the underlying protocol is supported through the `JmsInvokerServiceExporter` and `JmsInvokerProxyFactoryBean` classes in the `spring-jms` module.
- AMQP: Remoting via AMQP as the underlying protocol is supported by the separate Spring AMQP project.

While discussing the remoting capabilities of Spring, we 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;
    }
}
```

```
public interface AccountService {

    public void insertAccount(Account account);

    public List<Account> getAccounts(String name);
}
```

```
// 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...
    }
}
```

This section starts by exposing the service to a remote client by using RMI and talk a bit about the drawbacks of using RMI. It then continues with an example that uses Hessian as the protocol.

RMI

By 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 you use the RMI invoker, so you can, for example, plug in security frameworks or custom security credentials.

Exporting the Service by Using `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.

We first have to set up our service in the Spring container. The following example shows how to do so:

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

Next, we have to expose our service by using `RmiServiceExporter`. The following example shows how to do so:

```
<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>
```

In the preceding example, we override 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. So, in the preceding example, the service is bound at `'rmi://HOST:1199/AccountService'`. We use this 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 is used to communicate with the service.

Linking in the Service at the Client

Our client is a simple object that uses the `AccountService` to manage accounts, as the following example shows:

```
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 create a separate Spring container, to contain the following 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 is all we need to do to support the remote account service on the client. Spring transparently creates an invoker and remotely enables the account service through the `RmiServiceExporter`. At the client, we link it in by using the `RmiProxyFactoryBean`.

Using Hessian to Remotely Call Services through HTTP

Hessian offers a binary HTTP-based remoting protocol. It is developed by Caucho, and you can find more information about Hessian itself at https://www.caucho.com/.

Hessian

Hessian communicates through HTTP and does so by using a custom servlet. By using Spring's `DispatcherServlet` principles (see webmvc.pdf), we can wire up such a servlet to expose your services. First, we have to create a new servlet in our application, as shown in the following 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>
```

If you are familiar with Spring's `DispatcherServlet` principles, you probably know that now you 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 is used in the next section.

Alternatively, consider the use of Spring's simpler `HttpRequestHandlerServlet`. Doing so lets you 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. In this case, each servlet name needs to match the bean name of its target exporter.

Exposing Your Beans by Using `HessianServiceExporter`

In the newly created application context called `remoting-servlet.xml`, we create a `HessianServiceExporter` to export our services, as the following example shows:

```
<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 are ready to link in the service at the client. No explicit handler mapping is specified (to map request URLs onto services), so we use `BeanNameUrlHandlerMapping` used. Hence, the service is exported at the URL indicated through its bean name within the containing `DispatcherServlet` instance's mapping (as defined earlier): `https://HOST:8080/remoting/AccountService`.

Alternatively, you can create a `HessianServiceExporter` in your root application context (for example, in `WEB-INF/applicationContext.xml`), as the following example shows:

```
<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, you should define a corresponding servlet for this exporter in `web.xml`, with the same end result: The exporter gets mapped to the request path at `/remoting/AccountService`. Note that the servlet name needs to match the bean name of the target exporter. The following example shows how to do so:

```
<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>
```

Linking in the Service on the Client

By using the `HessianProxyFactoryBean`, we can link in the service at the client. The same principles apply as with the RMI example. We create a separate bean factory or application context and mention the following beans where the `SimpleObject` is by using the `AccountService` to manage accounts, as the following example shows:

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

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

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 be applied through using the `web.xml` security features, for example. Usually, you need not use per-user security credentials here. Rather, you can use shared credentials that you define at the `HessianProxyFactoryBean` level (similar to a JDBC `DataSource`), as the following example shows:

```
<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>
```

In the preceding example, we explicitly mention the `BeanNameUrlHandlerMapping` and set an interceptor, to let only administrators and operators call the beans mentioned in this application context.

The preceding example does not 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 https://projects.spring.io/spring-security/.

Spring HTTP Invoker

As opposed to Hessian, Spring HTTP invokers are both lightweight protocols that use their own slim serialization mechanisms and 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 by using the serialization mechanisms Hessian uses (see the next section for more considerations when you choose a remoting technology).

Under the hood, Spring uses either the standard facilities provided by the JDK or Apache

HttpComponents to perform HTTP calls. If you need more advanced and easier-to-use functionality, use the latter. See hc.apache.org/httpcomponents-client-ga/ for more information.

Be aware of vulnerabilities due to unsafe Java deserialization: Manipulated input streams can lead to unwanted code execution on the server during the deserialization step. As a consequence, do not expose HTTP invoker endpoints to untrusted clients. Rather, expose them only between your own services. In general, we strongly recommend using any other message format (such as 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. See https://blogs.oracle.com/java-platform-group/entry/incoming_filter_serialization_data_a and https://openjdk.java.net/jeps/290.

Exposing the Service Object

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

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

```
<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 is exposed through the DispatcherServlet instance's standard mapping facilities, as explained in the section on Hessian.

Alternatively, you can create an HttpInvokerServiceExporter in your root application context (for example, in 'WEB-INF/applicationContext.xml'), as the following example shows:

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

In addition, you can define a corresponding servlet for this exporter in web.xml, with the servlet name matching the bean name of the target exporter, as the following example shows:

```
<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>
```

Linking in the Service at the Client

Again, linking in the service from the client much resembles the way you would do it when you use Hessian. By using a proxy, Spring can translate your calls to HTTP POST requests to the URL that points to the exported service. The following example shows how to configure this arrangement:

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

As mentioned earlier, 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. The following example shows how to do so:

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

Java Web Services

Spring provides full support for the 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, which is a solution for contract-first, document-driven web services — highly recommended for building modern, future-proof web services.

Exposing Servlet-based Web Services by 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 use Spring's `@Autowired` annotation to express such dependencies on Spring-managed beans. The following example shows our class that extends `SpringBeanAutowiringSupport`:

```
/**
 * 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 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 environments, using the standard contract for JAX-WS servlet endpoint deployment. See the various Java EE web service tutorials for details.

Exporting Standalone Web Services by Using JAX-WS

The built-in JAX-WS provider that comes with Oracle's JDK supports exposure of web services by using the built-in HTTP server that is also included in the JDK. Spring's `SimpleJaxWsServiceExporter` detects all `@WebService`-annotated beans in the Spring application context and exports 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 are registered with the JAX-WS engine, but their lifecycle is up to the Spring application context. This means that you can apply Spring functionality (such as explicit dependency injection) to the endpoint instances. Annotation-driven injection through `@Autowired` works as well. The following example shows how to define these beans:

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

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

...
```

The `AccountServiceEndpoint` can but does not have to derive from Spring's `SpringBeanAutowiringSupport`, since the endpoint in this example is a fully Spring-managed bean. This means that the endpoint implementation can be as follows (without any superclass declared — and Spring's `@Autowired` configuration annotation is still 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);
    }
}
```

Exporting Web Services by Using 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.

This is not portable in a Java EE environment. It is mainly intended for non-EE environments, such as Tomcat, that embed the JAX-WS RI as part of the web application.

The differences from the standard style of exporting servlet-based endpoints are that the lifecycle of the endpoint instances themselves are managed by Spring and that there is only one JAX-WS servlet defined in `web.xml`. With the standard Java EE style (as shown earlier), you have one servlet definition per service endpoint, with each endpoint typically delegating to Spring beans (through the use of `@Autowired`, as shown earlier).

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

Accessing Web Services by Using JAX-WS

Spring provides two factory beans to create JAX-WS web service proxies, namely `LocalJaxWsServiceFactoryBean` and `JaxWsPortProxyFactoryBean`. The former can return only 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 the following example, we use `JaxWsPortProxyFactoryBean` to create a proxy for the `AccountService` endpoint (again):

```
<bean id="accountWebService" class=
"org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="example.AccountService"/> ①
    <property name="wsdlDocumentUrl" value=
"http://localhost:8888/AccountServiceEndpoint?WSDL"/>
    <property name="namespaceUri" value="https://example/"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountServiceEndpointPort"/>
</bean>
```

① Where `serviceInterface` is our business interface that the clients use.

`wsdlDocumentUrl` is the URL for the WSDL file. Spring needs this at 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 easy, as we have a bean factory for it that exposes it as an interface called `AccountService`. The following example shows how 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 as if it were a normal class, as the following example shows:

```
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.

JMS

You can also expose services transparently by 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`. As a result, throughput is 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 sides:

```
package com.foo;

public interface CheckingAccountService {

    public void cancelAccount(Long accountId);
}
```

The following simple implementation of the preceding interface is used on the server-side:

```
package com.foo;

public class SimpleCheckingAccountService implements CheckingAccountService {

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

The following configuration file contains the JMS-infrastructure beans that are shared on both the client and the server:

```
<?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
        https://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>
```

Server-side Configuration

On the server, you need to expose the service object that uses the `JmsInvokerServiceExporter`, as the following example shows:

```
<?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
        https://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>
```

```
package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

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

Client-side Configuration

The client merely needs to create a client-side proxy that implements the agreed-upon interface (`CheckingAccountService`).

The following example defines beans that you can inject into other client-side objects (and the proxy takes care of forwarding the call to the server-side object via JMS):

```
<?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
        https://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>
```

```
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(
"com/foo/client.xml", "com/foo/jms.xml");
        CheckingAccountService service = (CheckingAccountService) ctx.getBean(
"checkingAccountService");
        service.cancelAccount(new Long(10));
    }
}
```

AMQP

Remoting via AMQP as the underlying protocol is supported in the Spring AMQP project. For further details please visit the Spring Remoting section of the Spring AMQP reference.

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, such as `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. However, when you export 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 only 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.

Considerations when Choosing a Technology

Each and every technology presented here has its drawbacks. When choosing a technology, you should carefully consider your needs, the services you expose, and the objects you send over the wire.

When using RMI, you cannot access the objects through the HTTP protocol, unless you tunnel the RMI traffic. RMI is a fairly heavy-weight protocol, in that it supports full-object serialization, which is important when you use 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 but uses HTTP as transport. Note that HTTP invokers are not limited only to Java-to-Java remoting but also to Spring on both the client and the 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 letting the JMS broker take care of load balancing, discovery, and auto-failover. By default, Java serialization is used for JMS remoting, but the JMS provider could use a different mechanism for the wire formatting, such as XStream to let servers 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. Spring offers only appropriate hooks for plugging in third-party or custom

solutions.

REST Endpoints

The Spring Framework provides two choices for making calls to REST endpoints:

- `RestTemplate`: The original Spring REST client with a synchronous, template method API.
- WebClient: a non-blocking, reactive alternative that supports both synchronous and asynchronous as well as streaming scenarios.

As of 5.0, the non-blocking, reactive `WebClient` offers a modern alternative to the `RestTemplate` with efficient support for both synchronous and asynchronous as well as streaming scenarios. The `RestTemplate` will be deprecated in a future version and will not have major new features added going forward.

`RestTemplate`

The `RestTemplate` provides a higher level API over HTTP client libraries. It makes it easy to invoke REST endpoints in a single line. It exposes the following groups of overloaded methods:

Table 1. RestTemplate methods

Method group	Description
`getForObject`	Retrieves a representation via GET.
`getForEntity`	Retrieves a `ResponseEntity` (that is, status, headers, and body) by using GET.
`headForHeaders`	Retrieves all headers for a resource by using HEAD.
`postForLocation`	Creates a new resource by using POST and returns the `Location` header from the response.
`postForObject`	Creates a new resource by using POST and returns the representation from the response.
`postForEntity`	Creates a new resource by using POST and returns the representation from the response.
`put`	Creates or updates a resource by using PUT.
`patchForObject`	Updates a resource by using PATCH and returns the representation from the response. Note that the JDK `HttpURLConnection` does not support the `PATCH`, but Apache HttpComponents and others do.
`delete`	Deletes the resources at the specified URI by using DELETE.
`optionsForAllow`	Retrieves allowed HTTP methods for a resource by using ALLOW.

Method group	Description
`exchange`	More generalized (and less opinionated) version of the preceding methods that provides extra flexibility when needed. It accepts a `RequestEntity` (including HTTP method, URL, headers, and body as input) and returns a `ResponseEntity`. These methods allow the use of `ParameterizedTypeReference` instead of `Class` to specify a response type with generics.
`execute`	The most generalized way to perform a request, with full control over request preparation and response extraction through callback interfaces.

Initialization

The default constructor uses `java.net.HttpURLConnection` to perform requests. You can switch to a different HTTP library with an implementation of `ClientHttpRequestFactory`. There is built-in support for the following:

- Apache HttpComponents
- Netty
- OkHttp

For example, to switch to Apache HttpComponents, you can use the following:

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

Each `ClientHttpRequestFactory` exposes configuration options specific to the underlying HTTP client library — for example, for credentials, connection pooling, and other details.

Note that the `java.net` implementation for HTTP requests can raise an exception when accessing the status of a response that represents an error (such as 401). If this is an issue, switch to another HTTP client library.

URIs

Many of the `RestTemplate` methods accept a URI template and URI template variables, either as a `String` variable argument, or as `Map<String,String>`.

The following example uses a `String` variable argument:

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

The following example uses a `Map<String, String>`:

```
Map<String, String> vars = Collections.singletonMap("hotel", "42");

String result = restTemplate.getForObject(
        "https://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);
```

Keep in mind URI templates are automatically encoded, as the following example shows:

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

// Results in request to "https://example.com/hotel%20list"
```

You can use the `uriTemplateHandler` property of `RestTemplate` to customize how URIs are encoded. Alternatively, you can prepare a `java.net.URI` and pass it into one of the `RestTemplate` methods that accepts a `URI`.

For more details on working with and encoding URIs, see URI Links.

Headers

You can use the `exchange()` methods to specify request headers, as the following example shows:

```
String uriTemplate = "https://example.com/hotels/{hotel}";
URI uri = UriComponentsBuilder.fromUriString(uriTemplate).build(42);

RequestEntity<Void> requestEntity = RequestEntity.get(uri)
        .header(("MyRequestHeader", "MyValue")
        .build();

ResponseEntity<String> response = template.exchange(requestEntity, String.class);

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

You can obtain response headers through many `RestTemplate` method variants that return `ResponseEntity`.

Body

Objects passed into and returned from `RestTemplate` methods are converted to and from raw content with the help of an `HttpMessageConverter`.

On a POST, an input object is serialized to the request body, as the following example shows:

```
URI location = template.postForLocation("https://example.com/people", person);
```

You need not explicitly set the Content-Type header of the request. In most cases, you can find a

compatible message converter based on the source `Object` type, and the chosen message converter sets the content type accordingly. If necessary, you can use the `exchange` methods to explicitly provide the `Content-Type` request header, and that, in turn, influences what message converter is selected.

On a GET, the body of the response is deserialized to an output `Object`, as the following example shows:

```
Person person = restTemplate.getForObject("https://example.com/people/{id}",
Person.class, 42);
```

The `Accept` header of the request does not need to be explicitly set. In most cases, a compatible message converter can be found based on the expected response type, which then helps to populate the `Accept` header. If necessary, you can use the `exchange` methods to provide the `Accept` header explicitly.

By default, `RestTemplate` registers all built-in message converters, depending on classpath checks that help to determine what optional conversion libraries are present. You can also set the message converters to use explicitly.

Message Conversion

Same as in Spring WebFlux

The `spring-web` module contains the `HttpMessageConverter` contract for reading and writing the body of HTTP requests and responses through `InputStream` and `OutputStream`. `HttpMessageConverter` instances are used on the client side (for example, in the `RestTemplate`) and on the server side (for example, in Spring MVC REST controllers).

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

The implementations of `HttpMessageConverter` are described in the following sections. For all converters, a default media type is used, but you can override it by setting the `supportedMediaTypes` bean property. The following table describes each implementation:

Table 2. HttpMessageConverter Implementations

MessageConverter	Description
`StringHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write `String` instances 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`.

MessageConverter	Description
`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 `application/x-www-form-urlencoded` media type. Form data is read from and written into a `MultiValueMap<String, String>`. The converter can also write (but not read) multipart data read from a `MultiValueMap<String, Object>`. By default, `multipart/form-data` is supported. As of Spring Framework 5.2, additional multipart subtypes can be supported for writing form data. Consult the javadoc for `FormHttpMessageConverter` for further details.
`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`. You can override this by setting the `supportedMediaTypes` property and overriding `getContentType(byte[])`.
`MarshallingHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write XML by 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. You can inject these through constructor or bean properties. By default, this converter supports `text/xml` and `application/xml`.
`MappingJackson2HttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write JSON by using Jackson's `ObjectMapper`. You can customize JSON mapping as needed through the use of Jackson's provided annotations. When you need further control (for cases where custom JSON serializers/deserializers need to be provided for specific types), you can inject a custom `ObjectMapper` through the `ObjectMapper` property. By default, this converter supports `application/json`.
`MappingJackson2XmlHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write XML by using Jackson XML extension's `XmlMapper`. You can customize XML mapping as needed through the use of JAXB or Jackson's provided annotations. When you need further control (for cases where custom XML serializers/deserializers need to be provided for specific types), you can inject a custom `XmlMapper` through the `ObjectMapper` property. 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.

Jackson JSON Views

You can specify a Jackson JSON View to serialize only a subset of the object properties, as the following example shows:

```
MappingJacksonValue value = new MappingJacksonValue(new User("eric", "7!jd#h23"));
value.setSerializationView(User.WithoutPasswordView.class);

RequestEntity<MappingJacksonValue> requestEntity =
    RequestEntity.post(new URI("https://example.com/user")).body(value);

ResponseEntity<String> response = template.exchange(requestEntity, String.class);
```

Multipart

To send multipart data, you need to provide a `MultiValueMap<String, Object>` whose values may be an `Object` for part content, a `Resource` for a file part, or an `HttpEntity` for part content with headers. For example:

```
    MultiValueMap<String, Object> parts = new LinkedMultiValueMap<>();

    parts.add("fieldPart", "fieldValue");
    parts.add("filePart", new FileSystemResource("...logo.png"));
    parts.add("jsonPart", new Person("Jason"));

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_XML);
    parts.add("xmlPart", new HttpEntity<>(myBean, headers));
```

In most cases, you do not have to specify the `Content-Type` for each part. The content type is determined automatically based on the `HttpMessageConverter` chosen to serialize it or, in the case of a `Resource` based on the file extension. If necessary, you can explicitly provide the `MediaType` with an `HttpEntity` wrapper.

Once the `MultiValueMap` is ready, you can pass it to the `RestTemplate`, as show below:

```
    MultiValueMap<String, Object> parts = ...;
    template.postForObject("https://example.com/upload", parts, Void.class);
```

If the `MultiValueMap` contains at least one non-`String` value, the `Content-Type` is set to `multipart/form-data` by the `FormHttpMessageConverter`. If the `MultiValueMap` has `String` values the `Content-Type` is defaulted to `application/x-www-form-urlencoded`. If necessary the `Content-Type` may also be set explicitly.

Using `AsyncRestTemplate` (Deprecated)

The `AsyncRestTemplate` is deprecated. For all use cases where you might consider using

`AsyncRestTemplate`, use the WebClient instead.

Enterprise JavaBeans (EJB) Integration

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 through 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 begin by discussing this topic.

Accessing EJBs

This section covers how to access EJBs.

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 and 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:

- Typically, code that uses 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 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 need not write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you actually add real value in such code.

Accessing Local SLSBs

Assume that we have a web controller that needs to use a local EJB. We 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. We call this business methods interface `MyComponent`. The following example shows such an interface:

```
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. We 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 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 saves the reference as an instance variable in the controller. The following example shows how to do so:

```
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 obtain our controller object out of a Spring container, we can (in the same context) configure a `LocalStatelessSessionProxyFactoryBean` instance, which is the EJB proxy object. We configure the proxy and set the `myComponent` property of the controller with the following configuration entry:

```
<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>
```

A lot of work happens behind the scenes, courtesy of the Spring AOP framework, although you are not 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 is 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. The following example shows how to do so:

```
<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. To replace this EJB reference with a POJO or a mock object or other test stub, we could change the `myComponent` bean definition without changing a line of Java code. Additionally, we have not 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 undetectable in typical use. Remember that we do not want to make fine-grained calls to EJBs anyway, as there is 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 is no reason to make it a prototype). However, if that bean container pre-instantiates singletons (as do the various XML `ApplicationContext` variants), you can have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup is performed in the `init()` method of this class and then cached, but the EJB has not been bound at the target location yet. The solution is to not pre-instantiate this factory object but to let it be created on first use. In the XML containers, you can control this by using the `lazy-init` attribute.

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

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 need not. Client code written for local EJBs that 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 that needs to be moved to local EJBs can either stay the same but do a lot of unnecessary handling of remote exceptions or 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 that 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 is re-thrown as the non-checked `RemoteAccessException` class, which is a subclass of `RuntimeException`. You can then switch the target service 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: Nothing stops you from declaring `RemoteException` in your business interface.

Accessing EJB 2.x SLSBs Versus EJB 3 SLSBs

Accessing EJB 2.x Session Beans and EJB 3 Session Beans through 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 can 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>` or `<jee:remote-slsb>` lookups provides consistent and more explicit EJB access configuration.

JMS (Java Message Service)

Spring provides a JMS integration framework that simplifies the use of the JMS API in much the same way as 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 you can use to create Message-Driven POJOs (MDPs). Spring also provides a declarative way to create message listeners.

The `org.springframework.jms.core` package 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 sending messages, consuming messages synchronously, and exposing the JMS session and message producer to the user.

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

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

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

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

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

Finally, the `org.springframework.jms.connection` package 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.

As of Spring Framework 5, Spring's JMS package fully supports JMS 2.0 and requires the JMS 2.0 API to be present at runtime. We recommend the use of a JMS 2.0 compatible provider.

If you happen to use an older message broker in your system, you may try upgrading to a JMS 2.0 compatible driver for your existing broker generation. Alternatively, you may also try to run against a JMS 1.1 based driver, simply putting the JMS 2.0 API jar on the classpath but only using JMS 1.1 compatible API against your driver. Spring's JMS support adheres to JMS 1.1 conventions by default, so with corresponding configuration it does support such a scenario. However, please consider this for transition scenarios only.

Using Spring JMS

This section describes how to use Spring's JMS components.

Using `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` needs only to implement callback interfaces that give them a clearly defined high-level contract. The `MessageCreator` callback interface creates a message when given a `Session` provided by the calling code in `JmsTemplate`. To allow for more complex usage of the JMS API, `SessionCallback` provides the JMS session, and `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 and uses default values. Since `JmsTemplate` has many send methods, setting 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 by using the `setReceiveTimeout` property.

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 a `MessageProducer` instance's `send` method (`send(Destination destination, Message message)`) uses 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 for sending a message and waiting 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 — that is, `org.springframework.messaging.Message`. This lets you create the message to send in a generic manner.

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, three intermediate objects are created and destroyed. To optimize the resource usage and increase performance, Spring provides two implementations of `ConnectionFactory`.

Using `SingleConnectionFactory`

Spring provides an implementation of the `ConnectionFactory` interface, `SingleConnectionFactory`, that returns the same `Connection` on all `createConnection()` calls and ignores 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.

Using `CachingConnectionFactory`

The `CachingConnectionFactory` extends the functionality of `SingleConnectionFactory` and adds the caching of `Session`, `MessageProducer`, and `MessageConsumer` instances. The initial cache size is set to `1`. You can use the `sessionCacheSize` property to increase the number of cached sessions. Note that the

number of actual cached sessions is more than that number, as sessions are cached based on their acknowledgment mode, so there can be up to four cached session instances (one for each acknowledgment mode) when `sessionCacheSize` is set to one . `MessageProducer` and `MessageConsumer` instances 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).

Destination Management

Destinations, as `ConnectionFactory` instances, are JMS administered objects that you can store and retrieved in JNDI. When configuring a Spring application context, you can use the JNDI `JndiObjectFactoryBean` factory class or `<jee:jndi-lookup>` to perform dependency injection on your object's references to JMS destinations. However, this strategy is often 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 include 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 that implements the `DestinationResolver` interface. `DynamicDestinationResolver` is the default implementation used by `JmsTemplate` and accommodates resolving dynamic destinations. A `JndiDestinationResolver` is also provided to act 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 user-defined name, 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 method `TopicSession createTopic(String topicName)` or the `QueueSession createQueue(String queueName)` method to create a new destination with default destination properties. Depending on the vendor implementation, `DynamicDestinationResolver` can 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`, is to be used. This property (used by `JmsTemplate`) determines the behavior of dynamic destination resolution through implementations of the `DestinationResolver` interface.

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

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.) Since Spring Framework 4.1, endpoint methods can be annotated with `@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 so on. This lets you write the (possibly complex) business logic associated with receiving a message (and possibly respond 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 specialized feature set.

- `SimpleMessageListenerContainer`
- `DefaultMessageListenerContainer`

Using `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 by 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 participation in externally managed transactions, it does support native JMS transactions. To enable this feature, you can switch the `sessionTransacted` flag to `true` or, in the XML namespace, set the `acknowledge` attribute to `transacted`. Exceptions thrown from your listener then lead to a rollback, 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 `Session` instances 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 can get lost when listener execution fails (since the provider automatically acknowledges each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (you can configure this by setting the `acceptMessagesWhileStopping` flag). Make sure to use transacted sessions in case of reliability needs (for example, for reliable queue handling and durable topic subscriptions).

Using `DefaultMessageListenerContainer`

This message listener container is 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`. As a result, 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 participation in externally managed transactions), and compatibility with Java EE environments.

You can customize the cache level of the container. 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 loss. Make sure to use a proper cache level in such a case.

This container also has recoverable capabilities when the broker goes down. By default, a simple `BackOff` implementation retries every five seconds. You can specify a custom `BackOff` implementation for more fine-grained recovery options. See api-spring-framework/util/backoff/ExponentialBackOff.html[`ExponentialBackOff`] for an example.

Like its sibling (`SimpleMessageListenerContainer`), `DefaultMessageListenerContainer` supports native JMS transactions and allows for customizing the acknowledgment mode. If feasible for your scenario, This is strongly recommended over externally managed transactions — 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 can cover such situations — for example, in the form of a business entity existence check or a protocol table check. Any such arrangements are significantly more efficient than the alternative: wrapping your entire processing with an XA transaction (through configuring your `DefaultMessageListenerContainer` with an `JtaTransactionManager`) to cover 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 can get lost when listener execution fails (since the provider automatically acknowledges each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (you can configure this by setting the `acceptMessagesWhileStopping` flag). Make sure to use transacted sessions in case of reliability needs (for example, for reliable queue handling and durable topic subscriptions).

Transaction Management

Spring provides a `JmsTransactionManager` that manages transactions for a single JMS `ConnectionFactory`. This lets JMS applications leverage the managed-transaction features of Spring, as described in Transaction Management section of the Data Access chapter. 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` pools Connection and Session instances, so those resources are efficiently reused across transactions. In a standalone environment, using Spring's `SingleConnectionFactory` 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.

You can also use `JmsTemplate` with the `JtaTransactionManager` and an XA-capable JMS `ConnectionFactory` to perform 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 or 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 you use the `JmsTemplate` in an unmanaged environment, you can specify these values through the use of the properties `sessionTransacted` and `sessionAcknowledgeMode`. When you use a `PlatformTransactionManager` with `JmsTemplate`, the template is always given a transactional JMS `Session`.

Sending a Message

The `JmsTemplate` contains many convenience methods to send a message. Send methods specify the destination by using a `javax.jms.Destination` object, and others specify the destination by using a `String` in a JNDI lookup. The `send` method that takes no destination argument uses the default destination.

The following example uses the `MessageCreator` callback to create a text message from the supplied `Session` object:

```
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");
            }
        });
    }
}
```

In the preceding example, 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 `send(String destinationName, MessageCreator creator)` method lets you send a message by 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.

Using Message Converters

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()` methods 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 through 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 or XStream) to create a `TextMessage` that represents 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 following example shows how to modify a message header and a property after a `java.util.Map` is converted to a message:

```
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 following form:

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

Using `SessionCallback` and `ProducerCallback`

While the send operations cover many common usage scenarios, you might sometimes 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.

Receiving a Message

This describes how to receive messages with JMS in Spring.

Synchronous Reception

While JMS is typically associated with asynchronous processing, you can 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 `receiveTimeout` property specifies how long the receiver should wait before giving up waiting for a message.

Asynchronous reception: Message-Driven POJOs

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 Using `MessageListenerAdapter`) on an MDP is that it must implement the `javax.jms.MessageListener` interface. Note that, if your POJO receives messages on multiple threads, it is important to ensure that your implementation is thread-safe.

The following example shows a simple implementation of an MDP:

```
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 have implemented your `MessageListener`, it is time to create a message listener container.

The following example shows how to define and configure one of the message listener containers that ships with Spring (in this case, `DefaultMessageListenerContainer`):

```
<!-- 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"/>
    <property name="messageListener" ref="messageListener"/>
</bean>
```

See the Spring javadoc of the various message listener containers (all of which implement MessageListenerContainer) for a full description of the features supported by each implementation.

Using the `SessionAwareMessageListener` Interface

The `SessionAwareMessageListener` interface is a Spring-specific interface that provides a similar contract to the JMS `MessageListener` interface but also gives the message-handling method access to the JMS `Session` from which the `Message` was received. The following listing shows the definition of the `SessionAwareMessageListener` interface:

```
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 (by 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.

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 thrown exceptions.

Using `MessageListenerAdapter`

The `MessageListenerAdapter` class is the final component in Spring's asynchronous messaging support. In a nutshell, it lets you expose almost any class as an MDP (though there are some constraints).

Consider the following interface definition:

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

Notice that, although the interface extends neither the `MessageListener` nor the `SessionAwareMessageListener` interface, you can still use it as an MDP by using 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.

Now consider the following implementation of the `MessageDelegate` interface:

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

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

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

<!-- 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"/>
    <property name="messageListener" ref="messageListener"/>
</bean>
```

The next example shows another MDP that can handle only receiving 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 can see later in this section). Notice also how the `receive(..)` method is strongly typed to receive and respond only to JMS `TextMessage` messages. The following listing shows the definition of the `TextMessageDelegate` interface:

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

The following listing shows a class that implements the `TextMessageDelegate` interface:

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

The configuration of the attendant `MessageListenerAdapter` would then be as follows:

```
<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>
```

Note that, if the `messageListener` receives a JMS `Message` of a type other than `TextMessage`, an `IllegalStateException` is 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 following interface and class:

```
public interface ResponsiveTextMessageDelegate {

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

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

If you use the `DefaultResponsiveTextMessageDelegate` in conjunction with a `MessageListenerAdapter`, any non-null value that is returned from the execution of the `'receive(..)'` method is (in the default configuration) converted into a `TextMessage`. The resulting `TextMessage` is then 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, an `InvalidDestinationException` is thrown (note that this exception is not swallowed and propagates up the call stack).

Processing Messages Within Transactions

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

You can activate local resource transactions through the `sessionTransacted` flag on the listener container definition. Each message listener invocation then operates within an active JMS transaction, with message reception rolled back in case of listener execution failure. Sending a response message (through `SessionAwareMessageListener`) is part of the same local transaction, but any other resource operations (such as database access) operate independently. This usually

requires duplicate message detection in the listener implementation, to cover the case where database processing has committed but message processing failed to commit.

Consider the following bean definition:

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

To participate in an externally managed transaction, you need to configure a transaction manager and use a listener container that supports externally managed transactions (typically, `DefaultMessageListenerContainer`).

To configure a message listener container for XA transaction participation, you 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 lets message reception as well as (for example) database access be part of the same transaction (with unified commit semantics, at the expense of XA transaction log overhead).

The following bean definition creates a transaction manager:

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

Then we need to add it to our earlier container configuration. The container takes care of the rest. The following example shows how to do so:

```
<bean id="jmsContainer" class=
"org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="transactionManager" ref="transactionManager"/> ①
</bean>
```

① Our transaction manager.

Support for JCA Message Endpoints

Beginning with version 2.5, Spring also provides support for a JCA-based `MessageListener` container. The `JmsMessageEndpointManager` tries to automatically determine the `ActivationSpec` class name from the provider's `ResourceAdapter` class name. Therefore, it is typically possible to provide Spring's

generic `JmsActivationSpecConfig`, as the following example shows:

```
<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 can set up a `JmsMessageEndpointManager` with a given `ActivationSpec` object. The `ActivationSpec` object may also come from a JNDI lookup (using `<jee:jndi-lookup>`). The following example shows how to do so:

```
<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`, you can configure the target `ResourceAdapter` locally, as the following example shows:

```
<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` can also point to an environment-specific thread pool — typically through a `SimpleTaskWorkManager` instance's `asyncTaskExecutor` property. Consider defining a shared thread pool for all your `ResourceAdapter` instances if you happen to use multiple adapters.

In some environments (such as WebLogic 9 or above), you can instead obtain the entire `ResourceAdapter` object from JNDI (by using `<jee:jndi-lookup>`). The Spring-based message listeners can then interact with the server-hosted `ResourceAdapter`, which also use the server's built-in `WorkManager`.

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

Spring also provides a generic JCA message endpoint manager that is not tied to JMS: `org.springframework.jca.endpoint.GenericMessageEndpointManager`. This component allows for using any message listener type (such as a CCI `MessageListener`) and any provider-specific `ActivationSpec` object. See your JCA provider's documentation to find out about the actual capabilities of your connector, and see the `GenericMessageEndpointManager` 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. As with EJB 2.1 MDBs, you can use any message listener interface supported by your JCA provider in the Spring context as well. Spring nevertheless provides explicit "convenience" support for JMS, because JMS is the most common endpoint API used with the JCA endpoint management contract.

Annotation-driven Listener Endpoints

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

```
@Component
public class MyService {

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

The idea of the preceding example 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, similar to what the `MessageListenerAdapter` provides).

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

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

Enable Listener Endpoint Annotations

To enable support for `@JmsListener` annotations, you can add `@EnableJms` to one of your `@Configuration` classes, as the following example shows:

```
@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), you can invoke the `processOrder` method with a core poll size of three threads and a maximum pool size of ten threads.

You can customize the listener container factory to use for each annotation or you can configure an explicit default by implementing the `JmsListenerConfigurer` interface. The default is required only if at least one endpoint is registered without a specific container factory. See the javadoc of classes that implement `JmsListenerConfigurer` for details and examples.

If you prefer XML configuration, you can use the `<jms:annotation-driven>` element, as the following example shows:

```
<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>
```

Programmatic Endpoint Registration

`JmsListenerEndpoint` provides a model of a JMS endpoint and is responsible for configuring the container for that model. The infrastructure lets you programmatically configure endpoints in

addition to the ones that are detected by the `JmsListener` annotation. The following example shows how to do so:

```
@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 preceding example, we used `SimpleJmsListenerEndpoint`, which provides the actual `MessageListener` to invoke. However, you could also build your own endpoint variant to describe a custom invocation mechanism.

Note that you could skip the use of `@JmsListener` altogether and programmatically register only your endpoints through `JmsListenerConfigurer`.

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. In the following example, we rewrite it to inject the `Order` with a custom header:

```
@Component
public class MyService {

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

The main elements you can inject in JMS listener endpoints are as follows:

- The raw `javax.jms.Message` or any of its subclasses (provided that it matches the incoming message type).
- The `javax.jms.Session` for optional access to the native JMS API (for example, for sending a custom reply).

- The `org.springframework.messaging.Message` that represents 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.
- A `@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 (`Message` or `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. The following example shows how to do so:

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

Handling of method arguments is provided by `DefaultMessageHandlerMethodFactory`, which you can further customize to support additional method arguments. You can customize the conversion and validation support 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 the following example shows:

```
@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;
    }
}
```

Response Management

The existing support in `MessageListenerAdapter` already lets your method have a non-`void` return type. When that is 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. You can now set that default destination by using the `@SendTo` annotation of the messaging abstraction.

Assuming that our `processOrder` method should now return an `OrderStatus`, we can write it to automatically send a response, as the following example shows:

```
@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 can return a `Message` instead, with a method similar to the following:

```
@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. We can rewrite the previous example as follows:

```
@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, as the following example

shows:

```
@Configuration
@EnableJms
public class AppConfig {

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

JMS Namespace Support

Spring provides an XML namespace for simplifying JMS configuration. To use the JMS namespace elements, you need to reference the JMS schema, as the following example shows:

```
<?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" ①
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/jms
https://www.springframework.org/schema/jms/spring-jms.xsd">

    <!-- bean definitions here -->

</beans>
```

① Referencing the JMS schema.

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/>` define shared listener container configuration and can contain `<listener/>` child elements. The following example shows 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 preceding example is equivalent to creating two distinct listener container bean definitions and two distinct `MessageListenerAdapter` bean definitions, as shown in Using `MessageListenerAdapter`. In addition to the attributes shown in the preceding example, the `listener` element can contain several optional ones. The following table describes all of the available attributes:

Table 3. Attributes of the JMS <listener> element

Attribute	Description
`id`	A bean name for the hosting listener container. If not specified, a bean name is 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` attribute points to a `MessageListener` or Spring `SessionAwareMessageListener`, you can omit this attribute.
`response-destination`	The name of the default response destination to which to send response messages. This is applied in case of a request message that does not carry a `JMSReplyTo` field. The type of this destination is determined by the listener-container's `response-destination-type` attribute. Note that this applies only to a listener method with a return value, for which each result object is 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 or consumers to start for this listener. This value can either be a simple number indicating the maximum number (for example, `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is only a hint and might be ignored at runtime. The 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. By using these attributes, you can define highly-customized listener containers while still benefiting from the convenience of the namespace.

You can automatically expose such settings as a `JmsListenerContainerFactory` by specifying the `id` of

the bean to expose through the `factory-id` attribute, as the following example shows:

```
<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. See the class-level javadoc of the `AbstractMessageListenerContainer` and its concrete subclasses for more details on the individual properties. The javadoc also provides a discussion of transaction choices and message redelivery scenarios.

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

Attribute	Description
`container-type`	The type of this listener container. The available options are `default`, `simple`, `default102`, or `simple102` (the default option is `default`).
`container-class`	A custom listener container implementation class as a fully qualified class name. The 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 `Destination` instances.
`message-converter`	A reference to the `MessageConverter` strategy for converting JMS Messages to listener method arguments. The 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 potentially enables the `pubSubDomain`, `subscriptionDurable` and `subscriptionShared` properties of the container. The default is `queue` (which disables those three properties).

Attribute	Description
`response-destination-type`	The JMS destination type for responses: `queue` or `topic`. The default is the value of the `destination-type` attribute.
`client-id`	The JMS client ID for this listener container. You must specify it when you use durable subscriptions.
`cache`	The cache level for JMS resources: `none`, `connection`, `session`, `consumer`, or `auto`. By default (`auto`), the cache level is effectively `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, you can specify the `transaction-manager` attribute, described later in table. The default is `auto`.
`transaction-manager`	A reference to an external `PlatformTransactionManager` (typically an XA-based transaction coordinator, such as Spring's `JtaTransactionManager`). If not specified, native acknowledging is used (see the `acknowledge` attribute).
`concurrency`	The number of concurrent sessions or consumers to start for each listener. It can either be a simple number indicating the maximum number (for example, `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is just a hint and might be ignored at runtime. The default is `1`. You should 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 (in milliseconds) to use for receive calls. The default is `1000` (one second). `-1` indicates no timeout.
`back-off`	Specifies the `BackOff` instance to use to compute the interval between recovery attempts. If the `BackOffExecution` implementation returns `BackOffExecution#STOP`, the listener container does not further try to recover. The `recovery-interval` value is ignored when this property is set. The default is a `FixedBackOff` with an interval of 5000 milliseconds (that is, five seconds).
`recovery-interval`	Specifies the interval between recovery attempts, in milliseconds. It offers a convenient way to create a `FixedBackOff` with the specified interval. For more recovery options, consider specifying a `BackOff` instance instead. The default is 5000 milliseconds (that is, five seconds).
`phase`	The lifecycle phase within which this container should start and stop. The lower the value, the earlier this container starts and the later it stops. The default is `Integer.MAX_VALUE`, meaning that the container starts as late as possible and stops as soon as possible.

Configuring a JCA-based listener container with the `jms` schema support is very similar, as the following example shows:

```
<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 following table describes the available configuration options for the JCA variant:

Table 5. 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. The default is `SimpleMessageConverter`.
`destination-type`	The JMS destination type for this listener: `queue`, `topic`, `durableTopic`, `sharedTopic`. or `sharedDurableTopic`. This potentially enables the `pubSubDomain`, `subscriptionDurable`, and `subscriptionShared` properties of the container. The default is `queue` (which disables those three properties).
`response-destination-type`	The JMS destination type for responses: `queue` or `topic`. The default is the value of the `destination-type` attribute.
`client-id`	The JMS client ID for this listener container. It 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, you can specify the `transaction-manager` attribute described later. The 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 is used (see the `acknowledge` attribute).

Attribute	Description
`concurrency`	The number of concurrent sessions or consumers to start for each listener. It can either be a simple number indicating the maximum number (for example `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is only a hint and is typically ignored at runtime when you use a JCA listener container. The default is 1.
`prefetch`	The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers.

JMX

The JMX (Java Management Extensions) support in Spring provides features that let you easily and transparently integrate your Spring application into a JMX infrastructure.

> **JMX?**
>
> This chapter is not an introduction to JMX. It does not try to explain why you might want to use JMX. If you are new to JMX, see 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.

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:

```
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 can configure an instance of the `MBeanExporter` class in your configuration file and pass in the bean, as the following example shows:

```
<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 preceding 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. You can change this behavior, as described in Controlling `ObjectName` Instances 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 (except those inherited from the `Object` class) are exposed as operations.

`MBeanExporter` is a `Lifecycle` bean (see Startup and Shutdown Callbacks). By default, MBeans are exported as late as possible during the application lifecycle. You can configure the `phase` at which the export happens or disable automatic registration by setting the `autoStartup` flag.

Creating an MBeanServer

The configuration shown in the preceding section assumes that the application is running in an environment that has one (and only one) `MBeanServer` already running. In this case, Spring tries to locate the running `MBeanServer` and register your beans with that server (if any). This behavior is useful when your application runs 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` instance's `server` property to the `MBeanServer` value returned by an `MBeanServerFactoryBean`, as the following example shows:

```
<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>
```

In the preceding example, an instance of `MBeanServer` is created by the `MBeanServerFactoryBean` and is supplied to the `MBeanExporter` through the `server` property. When you supply your own `MBeanServer` instance, the `MBeanExporter` does not try to locate a running `MBeanServer` and uses the supplied `MBeanServer` instance. For this to work correctly, you must have a JMX implementation on your classpath.

Reusing an Existing `MBeanServer`

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

```
<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 or cases where the existing `MBeanServer` has a dynamic (or unknown) `agentId` that is retrieved through lookup methods, you should use factory-method, as the following example shows:

```
<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>
```

Lazily Initialized MBeans

If you configure a bean with an `MBeanExporter` that is also configured for lazy initialization, the `MBeanExporter` does not break this contract and avoids instantiating the bean. Instead, it registers a proxy with the `MBeanServer` and defers obtaining the bean from the container until the first invocation on the proxy occurs.

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. You can cause MBeans to be automatically detected by the `MBeanExporter` by setting the `autodetect` property to `true`, as the following example shows:

```
<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
"/>
```

In the preceding example, the bean called `spring:mbean=true` is already a valid JMX MBean and is automatically registered by Spring. By default, a bean that is autodetected for JMX registration has its bean name used as the `ObjectName`. You can override this behavior, as detailed in Controlling `ObjectName` Instances for Your Beans.

Controlling the Registration Behavior

Consider the scenario where a Spring `MBeanExporter` attempts to register an `MBean` with an `MBeanServer` by 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`).

You can control 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`. The following table summarizes these registration behaviors:

Table 6. Registration Behaviors

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

The values in the preceding table are defined as enums on the `RegistrationPolicy` class. If you want to change the default registration behavior, you need to set the value of the `registrationPolicy` property on your `MBeanExporter` definition to one of those values.

The following example shows how to change from the default registration behavior to the `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="registrationPolicy" value="REPLACE_EXISTING"/>
    </bean>

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

</beans>
```

Controlling the Management Interface of Your Beans

In the example in the preceding section, you had little control over the management interface of your bean. All of the `public` properties and methods of each exported bean were 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.

Using the `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 exposed. The default implementation, `org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler`, defines a management interface that exposes all public properties and methods (as you saw in the examples in the preceding sections). Spring provides two additional implementations of the `MBeanInfoAssembler` interface that let you control the generated management interface by using either source-level metadata or any arbitrary interface.

Using Source-level Metadata: Java Annotations

By using the `MetadataMBeanInfoAssembler`, you can define the management interfaces for your beans by 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 that uses Java annotations, namely `org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource`. You must configure the `MetadataMBeanInfoAssembler` 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. You must mark each method you wish to expose as an operation with the `ManagedOperation` annotation and mark each property you wish to expose 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.

A `ManagedResource`-annotated bean must be public, as must the methods exposing an operation or an attribute.

The following example shows the annotated version of the `JmxTestBean` class that we used in Creating an MBeanServer:

```
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();
    }

}
```

In the preceding example, 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 Source-level Metadata Types.

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 causes both of these properties to be included in the management interface as attributes, but the `age` attribute is read-only.

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

The following configuration shows how you can 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>
```

In the preceding example, 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.

Source-level Metadata Types

The following table describes the source-level metadata types that are available for use in Spring JMX:

Table 7. 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

Purpose	Annotation	Annotation Type
Mark a getter or setter as one half of a JMX attribute.	`@ManagedAttribute`	Method (only getters and setters)
Define descriptions for operation parameters.	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

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

Table 8. 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`, or `ManagedOperationParameter`
`currencyTimeLimit`	Sets the value of the `currencyTimeLimit` descriptor field.	`ManagedResource` or `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`

Using the `AutodetectCapableMBeanInfoAssembler` Interface

To simplify configuration even further, Spring includes 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`, it is allowed to "vote" on the inclusion of beans for exposure to JMX.

The only implementation of the `AutodetectCapableMBeanInfo` interface is the `MetadataMBeanInfoAssembler`, which votes to include any bean that 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 similar to the following:

```
<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 the preceding configuration, no beans are passed to the `MBeanExporter`. However, the `JmxTestBean` is still 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 `ObjectName` Instances for Your Beans.

Defining Management Interfaces by Using Java Interfaces

In addition to the `MetadataMBeanInfoAssembler`, Spring also includes the `InterfaceBasedMBeanInfoAssembler`, which lets you 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, `InterfaceBasedMBeanInfoAssembler` extends this functionality by removing the need for naming conventions, letting you use more than one interface and removing the need for your beans to implement the MBean interfaces.

Consider the following interface, which is used to define a management interface for the `JmxTestBean` class that we showed 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 are exposed as operations and attributes on the JMX MBean. The following code 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>
```

In the preceding example, 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 preceding case, 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 through 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, the `InterfaceBasedMBeanInfoAssembler` reflects on the bean and uses all of the interfaces implemented by that bean to create the management interface.

Using `MethodNameBasedMBeanInfoAssembler`

`MethodNameBasedMBeanInfoAssembler` lets you specify a list of method names that are exposed to JMX as attributes and operations. The following code shows a sample configuration:

```
<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>
```

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

Controlling `ObjectName` Instances for Your Beans

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `ObjectNamingStrategy` to obtain an `ObjectName` instance for each of the beans it registers. By default, the default implementation, `KeyNamingStrategy` uses 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` (which builds an `ObjectName` based on the JVM identity of the bean) and the `MetadataNamingStrategy` (which uses source-level

metadata to obtain the `ObjectName`).

Reading `ObjectName` Instances from Properties

You can configure your own `KeyNamingStrategy` instance and configure it to read `ObjectName` instances from a `Properties` instance rather than use a bean key. The `KeyNamingStrategy` tries to locate an entry in the `Properties` with a key that corresponds to the bean key. If no entry is found or if the `Properties` instance is `null`, the bean key itself is used.

The following code 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>
```

The preceding example configures an instance of `KeyNamingStrategy` 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 is given an `ObjectName` of `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, the bean key name is used as the `ObjectName`.

Using `MetadataNamingStrategy`

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

```
<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, an `ObjectName` is 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.example:type=MyClass,name=myBean`:

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

Configuring Annotation-based MBean Export

If you prefer to use the annotation-based approach to define your management interfaces, a convenience subclass of `MBeanExporter` is available: `AnnotationMBeanExporter`. When defining an instance of this subclass, you no longer need the `namingStrategy`, `assembler`, and `attributeSource` configuration, since it always uses 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, as the following example shows:

```
@Configuration
@EnableMBeanExport
public class AppConfig {

}
```

If you prefer XML-based configuration, the `<context:mbean-export/>` element serves the same purpose and is shown in the following listing:

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

If necessary, you can provide a reference to a particular MBean `server`, and the `defaultDomain` attribute (a property of `AnnotationMBeanExporter`) accepts an alternate value for the generated MBean `ObjectName` domains. This is used in place of the fully qualified package name as described in the previous section on MetadataNamingStrategy, as the following example shows:

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

}
```

The following example shows the XML equivalent of the preceding annotation-based example:

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

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, you should use target-class proxies in that case (through setting the 'proxy-target-class' flag on `<aop:config/>`, `<tx:annotation-driven/>` and so on). Otherwise, your JMX beans might be silently ignored at startup.

Using 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.

Server-side Connectors

To have Spring JMX create, start, and expose a JSR-160 `JMXConnectorServer`, you can 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`, you can use the `serviceUrl` and `ObjectName` properties, respectively, as the following example shows:

```
<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 automatically registers your connector with the `MBeanServer` under that `ObjectName`. The following example shows the full set of parameters that 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 you use 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 use Spring to export remote services for you through RMI, Spring has already constructed an RMI registry. If not, you can easily start a registry by using the following snippet of configuration:

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

Client-side Connectors

To create an `MBeanServerConnection` to a remote JSR-160-enabled `MBeanServer`, you can use the `MBeanServerConnectionFactoryBean`, as the following example shows:

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

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 shown in the preceding sections use 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 such as SOAP or Hessian over simple HTTP or SSL and others, as the following example shows:

```
<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 preceding example, we used MX4J 3.0.0. See the official MX4J documentation for more information.

Accessing MBeans through Proxies

Spring JMX lets you create proxies that re-route calls to MBeans that are 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 following code 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>
```

In the preceding example, you can see that a proxy is created for the MBean registered under the `ObjectName` of `bean:name=testBean`. The set of interfaces that the proxy implements 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 through an `MBeanServerConnection`. By default, the local `MBeanServer` is located and used, but you can override this and provide an `MBeanServerConnection` that points to a remote `MBeanServer` to cater for proxies that point to remote MBeans:

```
<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>
```

In the preceding example, we create an `MBeanServerConnection` that points to a remote machine that uses the `MBeanServerConnectionFactoryBean`. This `MBeanServerConnection` is then passed to the `MBeanProxyFactoryBean` through the `server` property. The proxy that is created forwards all invocations to the `MBeanServer` through this `MBeanServerConnection`.

Notifications

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

Registering Listeners for Notifications

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

```
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());
    }

}
```

The following example adds `ConsoleLoggingNotificationListener` (defined in the preceding example) to `notificationListenerMappings`:

```
<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 preceding 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 through the `notificationListenerMappings` property is 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, as the following example shows:

```
<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 you want to register a single `NotificationListener` instance for all of the beans that the enclosing `MBeanExporter` exports, you can use the special wildcard (`*`) as the key for an entry in the `notificationListenerMappings` property map, as the following example shows:

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

If you need to do the inverse (that is, register a number of distinct listeners against an MBean), you must instead use the `notificationListeners` list property (in preference to the `notificationListenerMappings` property). This time, instead of configuring a `NotificationListener` for

a single MBean, we configure `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, as the following example shows:

```
<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 preceding example is equivalent to the first notification example. Assume, then, that we want to be given a handback object every time a `Notification` is raised and that we also want to filter out extraneous `Notifications` by supplying a `NotificationFilter`. The following example accomplishes these goals:

```
<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>
```

(For a full discussion of what a handback object is and, indeed, what a `NotificationFilter` is, see the

section of the JMX specification (1.2) entitled 'The JMX Notification Model'.)

Publishing Notifications

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

This section is really only relevant to Spring-managed beans that have been exposed as MBeans through 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 through an `MBeanExporter` instance can implement the related `NotificationPublisherAware` interface to gain access to a `NotificationPublisher` instance. The `NotificationPublisherAware` interface supplies an instance of a `NotificationPublisher` to the implementing bean through a simple setter method, which the bean can then use to publish `Notifications`.

As stated in the javadoc of the `NotificationPublisher` interface, managed beans that publish events through the `NotificationPublisher` mechanism are not responsible for the state management of notification listeners. Spring's JMX support takes care of handling all the JMX infrastructure issues. All you need to do, as an application developer, is implement the `NotificationPublisherAware` interface and start publishing events by using the supplied `NotificationPublisher` instance. Note that the `NotificationPublisher` is set after the managed bean has been registered with an `MBeanServer`.

Using a `NotificationPublisher` instance is quite straightforward. You create a JMX `Notification` instance (or an instance of an appropriate `Notification` subclass), populate the notification with the data pertinent to the event that is to be published, and invoke the `sendNotification(Notification)` on the `NotificationPublisher` instance, passing in the `Notification`.

In the following example, exported instances of the `JmxTestBean` 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.

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. (MX4J is an open-source implementation of various JMX specs.)

JCA CCI

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

- SPI (Service Provider Interfaces) that the connector provider must implement. These interfaces constitute a resource adapter that can be deployed on a Java EE application server. In such a scenario, the server manages connection pooling, transactions, 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, using the Spring Framework's general resource and transaction management facilities.

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

Configuring CCI

This section covers how to configure a Common Client Interface (CCI). It includes the following topics:

Connector Configuration

The base resource to use JCA CCI is the `ConnectionFactory` interface. The connector you use 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, you can use Spring's `JndiObjectFactoryBean` or `<jee:jndi-lookup>` to fetch the factory by its JNDI name.

Another way to use a connector is to embed it in your application (non-managed mode) and not use an application server to deploy and configure it. Spring offers the possibility to configure a connector as a bean, through a `FactoryBean` implementation called (`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 access to your `ConnectionFactory` instance, you can inject it into your components. These components can either be coded against the plain CCI API or use Spring's support classes for CCI access (e.g. `CciTemplate`).

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

`ConnectionFactory` Configuration in Spring

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 managed mode, you can access a `ConnectionFactory` from JNDI. Its properties are configured in the application server. The following example shows how to do so:

```
<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`. The following example shows how to do so:

```
<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 cannot 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.

Configuring CCI Connections

JCA CCI lets you configure the connections to the EIS by using the `ConnectionSpec` implementation of your connector. To configure its properties, you need to wrap the target connection factory with a

dedicated adapter, `ConnectionSpecConnectionFactoryAdapter`. You can configure the dedicated `ConnectionSpec` with the `connectionSpec` property (as an inner bean).

This property is not mandatory, because the CCI `ConnectionFactory` interface defines two different methods to obtain a CCI connection. You can often configure some of the `ConnectionSpec` properties in the application server (in managed mode) or on the corresponding local `ManagedConnectionFactory` implementation. The following listing shows the relevant parts of the `ConnectionFactory` interface definition:

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

Spring provides a `ConnectionSpecConnectionFactoryAdapter` that lets you specify 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 adapter uses the variant without that argument. The following example shows how to configure a `ConnectionSpecConnectionFactoryAdapter`:

```
<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>
```

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 lazily opens a single connection and closes it when this bean is destroyed at application shutdown. This class exposes special `Connection` proxies that behave accordingly, all sharing the same underlying physical connection. The following example shows how to use the `SingleConnectionFactory` adapter class:

```
<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`. You can use an intermediary `ConnectionSpecConnectionFactoryAdapter` that the `SingleConnectionFactory` talks to if you require a single connection for a specific `ConnectionSpec`.

Using Spring's CCI Access Support

This section describes how to use Spring's support for CCI to achieve various purposes. It includes the following topics:

- Record Conversion
- Using `CciTemplate`
- Using DAO Support
- Automatic Output Record Generation
- `CciTemplate` `Interaction` Summary
- Using a CCI Connection and an Interaction Directly
- Example of `CciTemplate` Usage

Record Conversion

One of the aims of Spring's JCA CCI support is to provide convenient facilities for manipulating CCI records. You can specify the strategy to create records and extract data from records, for use with Spring's `CciTemplate`. The interfaces described in this section configure the strategy to use input and

output records if you do not want to work with records directly in your application.

To create an input `Record`, you can use a dedicated implementation of the `RecordCreator` interface. The following listing shows the `RecordCreator` interface definition:

```
public interface RecordCreator {

    Record createRecord(RecordFactory recordFactory) throws ResourceException,
DataAccessException;

}
```

The `createRecord(..)` method receives a `RecordFactory` instance as a parameter, which corresponds to the `RecordFactory` of the `ConnectionFactory` used. You can use this reference to create `IndexedRecord` or `MappedRecord` instances. The following sample shows how to use the `RecordCreator` interface and indexed or 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;
    }

}
```

You can use an output `Record` to receive data back from the EIS. Hence, you can pass a specific implementation of the `RecordExtractor` interface to Spring's `CciTemplate` to extract data from the output `Record`. The following listing shows the `RecordExtractor` interface definition:

```
public interface RecordExtractor {

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

}
```

The following example 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);
    }

}
```

Using `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, such as 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, as the following listing shows:

```
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` knows which `execute` method to call on the interaction. In any case, a correctly initialized `InteractionSpec` instance is mandatory.

You can use `CciTemplate.execute(..)` in two ways:

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

With the first approach, the following methods (which directly correspond to those on the `Interaction` interface) of the template are used:

```
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 following listing shows the corresponding `CciTemplate` methods:

```
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 calls the corresponding `execute` method of the CCI `Interaction` with two parameters: `InteractionSpec` and an input `Record`. It receives an output `Record` as its return value.

`CciTemplate` also provides methods to create `IndexRecord` and `MappedRecord` outside of a `RecordCreator` implementation, through its `createIndexRecord(..)` and `createMappedRecord(..)` methods. You can use this within DAO implementations to create `Record` instances to pass into corresponding `CciTemplate.execute(..)` methods. The following listing shows the `CciTemplate` interface definition:

```
public class CciTemplate implements CciOperations {

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

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

}
```

Using DAO Support

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

```
public abstract class CciDaoSupport {

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

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

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

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

}
```

Automatic Output Record Generation

If the connector you use supports only 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 is then returned to the caller of the template.

This property holds an implementation of the `RecordCreator` interface, to be used for that purpose.

You must directly specify the `outputRecordCreator` property on the `CciTemplate`. The following example shows how to do so:

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

Alternatively (and we recommend this approach), in the Spring configuration, if the `CciTemplate` is configured as a dedicated bean instance, you can define beans in the following fashion:

```
<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 is usually configured as a shared instance.

`CciTemplate` `Interaction` Summary

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

Table 9. 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)`

`CciTemplate` **method signature**	`CciTemplate outputRecordCreator` **property**	`execute` **method called on the CCI Interaction**
`Record execute(InteractionSpec, RecordCreator, RecordExtractor)`	Set	`void execute(InteractionSpec, Record, Record)`

Using a CCI Connection and an Interaction Directly

`CciTemplate` also lets you 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 `ConnectionCallback` interface provides a CCI `Connection` as an argument (to perform custom operations on it) plus the CCI `ConnectionFactory` with which the `Connection` was created. The latter can be useful (for example, to get an associated `RecordFactory` instance and create indexed/mapped records). The following listing shows the `ConnectionCallback` interface definition:

```
public interface ConnectionCallback {

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

}
```

The `InteractionCallback` interface provides the CCI `Interaction` (to perform custom operations on it) plus the corresponding CCI `ConnectionFactory`. The following listing shows the `InteractionCallback` interface definition:

```
public interface InteractionCallback {

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

}
```

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

Example of `CciTemplate` Usage

In this section, we show the usage of the `CciTemplate` to access a CICS with ECI mode, with the IBM CICS ECI connector.

First, we must do some initializations on the CCI `InteractionSpec` to specify which CICS program to

access and how to interact with it, as the following example shows:

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

Then the program can use CCI through Spring's template and specify mappings between custom objects and CCI `Records`, as the following example shows:

```
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, you can use callbacks to work directly on CCI connections or interactions. The following example shows how to do so:

```
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;
    }

}
```

With a `ConnectionCallback`, the `Connection` used is managed and closed by the `CciTemplate`, but the callback implementation must manage any interactions created on the connection.

For a more specific callback, you can implement an `InteractionCallback`. If you do so, the passed-in `Interaction` is managed and closed by the `CciTemplate`. The following example shows how to do so:

```
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 preceding examples, the corresponding configuration of the involved Spring beans could resemble the following example in non-managed mode:

```
<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 resemble the following example:

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

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

Modeling CCI Access as Operation Objects

The `org.springframework.jca.cci.object` package contains support classes that let you access the EIS in a different style: through reusable operation objects, analogous to Spring's JDBC operation objects (see the JDBC section of the Data Access chapter). This usually encapsulates the CCI API. An application-level input object is 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.

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

Using `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 an 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`

The following listing shows 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, ito 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 the result. The following example shows how to do so:

```
public abstract class MappingRecordOperation extends EisOperation {

    ...

    public Object execute(Object inputObject) throws DataAccessException {
    }

    ...
}
```

Contrary to the `CciTemplate` class, this `execute(..)` method does not have an `InteractionSpec` as an argument. Instead, the `InteractionSpec` is global to the operation. You must use the following constructor to instantiate an operation object with a specific `InteractionSpec`. The following example shows how to do so:

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

Using `MappingCommAreaOperation`

Some connectors use records based on a COMMAREA, which represents an array of bytes that contain parameters to send to the EIS and the 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 this special COMMAREA support. It implicitly uses the `CommAreaRecord` class as the input and output record type and provides two new methods to convert an input object into an input COMMAREA and convert the output COMMAREA into an output object. The following listing shows the relevant method signatures:

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {

    ...

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

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

    ...

}
```

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.

Summary

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

Table 10. 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)`

Example of `MappingRecordOperation` Usage

In this section, we show how to use `MappingRecordOperation` to access a database with the Blackbox CCI connector.

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

First, you must do some initializations on the CCI `InteractionSpec` to specify which SQL request to execute. In the following example, 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:

```
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 an argument. Note that you could set up the operation object as a shared instance, as it is thread-safe. The following executes the operation object with the person identifier as an argument:

```
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 be as follows in non-managed mode:

```
<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 be as follows:

```
<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>
```

Example of `MappingCommAreaOperation` Usage

In this section, we show how to use the usage of the `MappingCommAreaOperation` to access a CICS with ECI mode with the IBM CICS ECI connector.

First, we need to initialize the CCI `InteractionSpec` to specify which CICS program to access and how to interact with it, as the following example shows:

```
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();
        }
    }

}
```

We can then subclass the abstract `EciMappingOperation` class to specify mappings between custom objects and `Records`, as the following example shows:

```
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 be as follows in non-managed mode:

```
<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 be as follows:

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

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

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 the CICS EPI connector), local transactions (for example, with a CICS ECI connector), and global transactions (for example, with an IMS connector). The following example configures the global option:

```
<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 the 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 in a fashion that is fully compliant with Spring's generic `PlatformTransactionManager` abstraction. The following example configures a `CciLocalTransactionManager`:

```
<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>
```

You can use both transaction strategies 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. You can switch between `JtaTransactionManager` and `CciLocalTransactionManager` as needed, keeping your

transaction demarcation as-is.

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

Email

This section describes how to send email with the Spring Framework.

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 you 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 that encapsulates 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 that provide a higher level of abstraction over the lower level mail system exceptions, with the root exception being `MailException`. See the javadoc 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 called `org.springframework.mail.javamail.MimeMessagePreparator` for preparing a `MimeMessage`.

Usage

Assume that we have a business interface called `OrderManager`, as the following example shows:

```
public interface OrderManager {

    void placeOrder(Order order);

}
```

Further assume that we have a requirement stating that an email message with an order number needs to be generated and sent to a customer who placed the relevant order.

Basic `MailSender` and `SimpleMailMessage` Usage

The following example shows how to use `MailSender` and `SimpleMailMessage` to send an email when someone places an order:

```
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());
        }
    }

}
```

The following example shows the bean definitions for the preceding code:

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

Using `JavaMailSender` and `MimeMessagePreparator`

This section describes another implementation of `OrderManager` that uses the `MimeMessagePreparator` callback interface. In the following example, the `mailSender` property is of type `JavaMailSender` so that we are able to use the JavaMail `MimeMessage` class:

```
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.example"));
                mimeMessage.setText("Dear " + order.getCustomer().getFirstName() + " "
+
                        order.getCustomer().getLastName() + ", thanks for your 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. See the relevant javadoc for more information.

Using the JavaMail `MimeMessageHelper`

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

```
// 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);
```

Sending Attachments and Inline Resources

Multipart email messages allow for both attachments and inline resources. Examples of inline resources include an image or a stylesheet that you want to use in your message but that you do not want displayed as an attachment.

Attachments

The following example shows you how to use the `MimeMessageHelper` to send an email 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 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 `MimeMessage` by using the specified `Content-ID` (`identifier1234` in the above example). The order in which you add the text and the resource are very important. Be sure to first add the text and then the resources. If you are doing it the other way around, it does not work.

Creating Email Content by Using a Templating Library

The code in the examples shown in the previous sections explicitly created the content of the email message, by 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, developers often do not create the content of email messages by using the previously shown 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, and so on.

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 when the content of your email messages becomes even moderately complex, and, with the Spring Framework's support classes for FreeMarker, it becomes quite easy to do.

Task Execution and Scheduling

The Spring Framework provides abstractions for the 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 to support scheduling with the `Timer` (part of the JDK since 1.3) and the Quartz Scheduler (https://www.quartz-scheduler.org/). You can set up both of those schedulers by 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 lets you invoke a method of an existing target object (analogous to the normal `MethodInvokingFactoryBean` operation).

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 the 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, you can also use this abstraction for your own needs.

`TaskExecutor` Types

Spring includes a number of pre-built implementations of `TaskExecutor`. In all likelihood, you should never need to implement your own. The variants that Spring provides are as follows:

- `SyncTaskExecutor`: This implementation does not execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multi-threading is not necessary, such as in simple test cases.
- `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 that blocks any invocations that are over the limit until a slot has been freed up. If you are looking for true pooling, see `ThreadPoolTaskExecutor`, later in this list.
- `ConcurrentTaskExecutor`: This implementation is an adapter for a `java.util.concurrent.Executor` instance. There is an alternative (`ThreadPoolTaskExecutor`) that exposes the `Executor`

configuration parameters as bean properties. There is rarely a need to use `ConcurrentTaskExecutor` directly. However, if the `ThreadPoolTaskExecutor` is not flexible enough for your needs, `ConcurrentTaskExecutor` is an alternative.

- `ThreadPoolTaskExecutor`: This implementation is most commonly used. 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`, we recommend that you use a `ConcurrentTaskExecutor` instead.
- `WorkManagerTaskExecutor`: This implementation uses a CommonJ `WorkManager` as its backing service provider and is the central convenience class for setting up CommonJ-based thread pool integration on WebLogic or WebSphere within a Spring application context.
- `DefaultManagedTaskExecutor`: This implementation uses a JNDI-obtained `ManagedExecutorService` in a JSR-236 compatible runtime environment (such as a Java EE 7+ application server), replacing a CommonJ WorkManager for that purpose.

Using a `TaskExecutor`

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

```
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 it yourself, you add your `Runnable` to the queue. Then the `TaskExecutor` uses its internal rules to decide when the task gets executed.

To configure the rules that the `TaskExecutor` uses, we expose simple bean properties:

```
<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>
```

The Spring `TaskScheduler` Abstraction

In addition to the `TaskExecutor` abstraction, Spring 3.0 introduced a `TaskScheduler` with a variety of methods for scheduling tasks to run at some point in the future. The following listing shows the `TaskScheduler` interface definition:

```
public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task, Trigger trigger);

    ScheduledFuture schedule(Runnable task, Instant startTime);

    ScheduledFuture schedule(Runnable task, Date startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Instant startTime, Duration
period);

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

    ScheduledFuture scheduleAtFixedRate(Runnable task, Duration period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Instant startTime, Duration
delay);

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

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Duration delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);
}
```

The simplest method is the one named `schedule` that takes only a `Runnable` and a `Date`. That causes 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.

`Trigger` Interface

The `Trigger` interface is essentially inspired by JSR-236 which, as of Spring 3.0, was not yet 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, as the following listing shows:

```
public interface Trigger {

    Date nextExecutionTime(TriggerContext triggerContext);
}
```

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). The following listing shows the available methods for `Trigger` implementations.

```
public interface TriggerContext {

    Date lastScheduledExecutionTime();

    Date lastActualExecutionTime();

    Date lastCompletionTime();
}
```

`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 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 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 you can use it 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 you can configure such `Triggers` externally and, therefore, easily modify or extend them.

`TaskScheduler` implementations

As with Spring's `TaskExecutor` abstraction, the primary benefit of the `TaskScheduler` arrangement is that an application's scheduling needs are decoupled from the deployment environment. This abstraction level is particularly relevant when deploying to an application server environment where threads should not be created directly by the application itself. For such scenarios, Spring provides a `TimerManagerTaskScheduler` that delegates to a CommonJ `TimerManager` on WebLogic or

WebSphere as well as a more recent `DefaultManagedTaskScheduler` that delegates to a JSR-236 `ManagedScheduledExecutorService` in a Java EE 7+ environment. Both are typically configured with a JNDI lookup.

Whenever external thread management is not a requirement, a simpler alternative is a local `ScheduledExecutorService` setup within the application, which can be adapted through Spring's `ConcurrentTaskScheduler`. As a convenience, Spring also provides a `ThreadPoolTaskScheduler`, which internally delegates to a `ScheduledExecutorService` to provide common bean-style configuration along the lines of `ThreadPoolTaskExecutor`. These variants work perfectly fine for locally embedded thread pool setups in lenient application server environments, as well — in particular on Tomcat and Jetty.

Annotation Support for Scheduling and Asynchronous Execution

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

Enable Scheduling Annotations

To enable support for `@Scheduled` and `@Async` annotations, you can add `@EnableScheduling` and `@EnableAsync` to one of your `@Configuration` classes, as the following example shows:

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

You can pick and choose the relevant annotations for your application. For example, if you need only support for `@Scheduled`, you can omit `@EnableAsync`. For more fine-grained control, you can additionally implement the `SchedulingConfigurer` interface, the `AsyncConfigurer` interface, or both. See the `SchedulingConfigurer` and `AsyncConfigurer` javadoc for full details.

If you prefer XML configuration, you can use the `<task:annotation-driven>` element, as the following example shows:

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

Note that, with the preceding XML, 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.

The `@Scheduled` annotation

You can add the `@Scheduled` annotation to a method, along with trigger metadata. For example, the following method is invoked every five seconds with a fixed delay, meaning that the period is measured from the completion time of each preceding invocation:

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

If you need a fixed-rate execution, you can change the property name specified within the annotation. The following method is invoked every five 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, you can specify an initial delay by indicating the number of milliseconds to wait before the first execution of the method, as the following `fixedRate` example shows:

```
@Scheduled(initialDelay=1000, fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

If simple periodic scheduling is not expressive enough, you can provide a cron expression. For example, the following executes only on weekdays:

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

You can also use the `zone` attribute to specify the time zone in which the cron expression is 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, 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 that are annotated with `@Scheduled` and registered as regular Spring beans with the container. Otherwise, you would get double initialization (once through the container and once through the `@Configurable` aspect), with the consequence of each `@Scheduled` method being invoked twice.

The `@Async` annotation

You can provide the `@Async` annotation on a method so that invocation of that method occurs asynchronously. In other words, the caller returns immediately upon invocation, while the actual execution of the method occurs in a task that has been submitted to a Spring `TaskExecutor`. In the simplest case, you can apply the annotation to a method that returns `void`, as the following example shows:

```
@Async
void doSomething() {
    // this will be executed asynchronously
}
```

Unlike the methods annotated with the `@Scheduled` annotation, these methods can expect arguments, because they are invoked in the “normal” way by callers at runtime rather than from a scheduled task being managed by the container. For example, the following code is a legitimate application of the `@Async` annotation:

```
@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`. The following example shows how to use `@Async` on a method that returns a value:

```
@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.

You can not use `@Async` in conjunction with lifecycle callbacks such as `@PostConstruct`. To asynchronously initialize Spring beans, you currently have to use a separate initializing Spring bean that then invokes the `@Async` annotated method on the target, as the following example shows:

```
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();
    }

}
```

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 asynchronous. However, you can manually set up Spring's `AsyncExecutionInterceptor` with Spring AOP, in combination with a custom pointcut.

Executor Qualification with `@Async`

By default, when specifying `@Async` on a method, the executor that is used is the one configured when enabling async support, i.e. the "annotation-driven" element if you are using XML or your `AsyncConfigurer` implementation, if any. However, you can use the `value` attribute of the `@Async` annotation when you need to indicate that an executor other than the default should be used when executing a given method. The following example shows how to do so:

```
@Async("otherExecutor")
void doSomething(String s) {
    // this will be executed asynchronously by "otherExecutor"
}
```

In this case, `"otherExecutor"` can be the name of any `Executor` bean in the Spring container, or it may be the name of a qualifier associated with any `Executor` (for example, as specified with the `<qualifier>` element or Spring's `@Qualifier` annotation).

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 is thrown when calling `get` on the `Future` result. With a `void` return type, however, the exception is uncaught and cannot be transmitted. You can provide an `AsyncUncaughtExceptionHandler` to handle such exceptions. The following example shows how to do so:

```
public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler
{

    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object... params)
{
        // handle exception
    }
}
```

By default, the exception is merely logged. You can define a custom `AsyncUncaughtExceptionHandler` by using `AsyncConfigurer` or the `<task:annotation-driven/>` XML element.

The `task` Namespace

As of version 3.0, Spring includes an XML namespace for configuring `TaskExecutor` and `TaskScheduler` instances. It also provides a convenient way to configure tasks to be scheduled with a trigger.

The 'scheduler' Element

The following element creates a `ThreadPoolTaskScheduler` instance with the specified thread pool

size:

```
<task:scheduler id="scheduler" pool-size="10"/>
```

The value provided for the `id` attribute is 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 has only a single thread. There are no other configuration options for the scheduler.

The `executor` Element

The following creates a `ThreadPoolTaskExecutor` instance:

```
<task:executor id="executor" pool-size="10"/>
```

As with the scheduler shown in the previous section, the value provided for the `id` attribute is 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 only a single size, an executor's thread pool can have different values for the core and the max size. If you provide a single value, the executor has 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`. The following example sets a minimum value of `5` and a maximum value of `25`:

```
<task:executor
        id="executorWithPoolSizeRange"
        pool-size="5-25"
        queue-capacity="100"/>
```

In the preceding 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, see the documentation for `ThreadPoolExecutor`. The main idea is that, when a task is submitted, the executor first tries 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, the task is added to the queue, as long as its capacity has not yet been reached. Only then, if the queue's capacity has been reached, does the executor create a new thread beyond the core size. If the max size has also been reached, then the executor rejects 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, the max size has no effect at all. Since the executor always tries 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).

Consider the case, as mentioned above, when a task is rejected. By default, when a task is rejected, a

thread pool executor throws 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, you can instead configure either `DiscardPolicy` or `DiscardOldestPolicy`. 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 forces the thread that is calling the submit method to run the task itself. The idea is that such a caller is 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. You can choose any of these options from an enumeration of values available for the `rejection-policy` attribute on the `executor` element.

The following example shows an `executor` element with a number of attributes to specify various behaviors:

```
<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 get terminated. A time value of zero causes excess threads to terminate immediately after executing a task without remaining follow-up work in the task queue. The following example sets the `keep-alive` value to two minutes:

```
<task:executor
        id="executorWithKeepAlive"
        pool-size="5-25"
        keep-alive="120"/>
```

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. The following listing shows a simple example:

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

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, you can specify an 'initial-delay' parameter, indicating the number of milliseconds to wait before the first execution of the method. For more control, you can instead provide a `cron` attribute. The following example shows these other options:

```
<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"/>
```

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, see https://www.quartz-scheduler.org/. For convenience purposes, Spring offers a couple of classes that simplify using Quartz within Spring-based applications.

Using the `JobDetailFactoryBean`

Quartz `JobDetail` objects contain all the information needed to run a job. Spring provides a `JobDetailFactoryBean`, which provides bean-style properties for XML configuration purposes. Consider the following example:

```
<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 the 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 the following example, the `ExampleJob` contains a bean property named `timeout`, and the `JobDetail` has 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 available to you as well.

By 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` (`exampleJob` in the preceding example above).

Using the `MethodInvokingJobDetailFactoryBean`

Often you merely need to invoke a method on a specific object. By using the `MethodInvokingJobDetailFactoryBean`, you can do exactly this, as the following example shows:

```
<bean id="jobDetail" class=
"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
</bean>
```

The preceding example results in the `doIt` method being called on the `exampleBusinessObject` method, as the following example shows:

```
public class ExampleBusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

By using the `MethodInvokingJobDetailFactoryBean`, you need not create one-line jobs that merely invoke a method. You need only 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 is possible that, before the first job has finished, the second one starts. If `JobDetail` classes implement the `Stateful` interface, this does not happen. The second job does not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryBean` be non-concurrent, set the `concurrent` flag to `false`, as the following example shows:

```
<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.

Wiring up Jobs by Using Triggers and `SchedulerFactoryBean`

We have created job details and jobs. We have also reviewed the convenience bean that lets you invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done by 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.

The following listing uses both a `SimpleTriggerFactoryBean` and a `CronTriggerFactoryBean`:

```
<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>
```

The preceding example sets up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one running every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryBean`, as the following example shows:

```
<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`, such as the calendars used by the job details, properties to customize Quartz with, and others. See the `SchedulerFactoryBean` javadoc for more information.

Cache Abstraction

Since version 3.1, the Spring Framework provides support for transparently adding caching to 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 extended with the support of JSR-107 annotations and more customization options.

Understanding the Cache Abstraction

> **Cache vs Buffer**
>
> The terms, "buffer" and "cache," tend to be used interchangeably. Note, however, that they represent different things. Traditionally, a buffer is used as an intermediate temporary store for data between a fast and a slow entity. As one party would have to wait for the other (which affects performance), the buffer alleviates this by allowing entire blocks of data to move at once rather than in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are visible to at least one party that is aware of it.
>
> A cache, on the other hand, is, by definition, hidden, and neither party is aware that caching occurs. It also improves performance but does so by letting the same data be read multiple times in a fast fashion.
>
> You can find a further explanation of the differences between a buffer and a cache here.

At its core, the cache abstraction applies caching to Java methods, thus reducing the number of executions based on the information available in the cache. That is, each time a targeted method is invoked, the abstraction applies a caching behavior that checks whether the method has been already executed for the given arguments. If it has been executed, the cached result is returned without having to execute the actual method. If the method has not been executed, then it is executed, and the result is 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.

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 executed.

The caching abstraction provides other cache-related operations, such as the ability to update the content of the cache or to remove one or all entries. These are useful if the cache deals with data that can change during the course of the application.

As with other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of actual storage to store the cache data — that is, the

abstraction frees you from having to write the caching logic but does not provide the actual data store. This abstraction is materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

Spring provides a few implementations of that abstraction: JDK `java.util.concurrent.ConcurrentMap` based caches, Ehcache 2.x, Gemfire cache, Caffeine, and JSR-107 compliant caches (such as Ehcache 3.x). See Plugging-in Different Back-end Caches for more information on plugging in other cache stores and providers.

The caching abstraction has no special handling for multi-threaded and multi-process environments, as such features are handled by the cache implementation.
.

If you have a multi-process environment (that is, an application deployed on several nodes), you need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes can be enough. However, 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. See the documentation of your cache provider for more details.

To use the cache abstraction, you need 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 from which it is read.

Declarative Annotation-based Caching

For caching declaration, Spring's caching 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.

The `@Cacheable` Annotation

As the name implies, you can use `@Cacheable` to demarcate methods that are cacheable — that is, methods for which the result is stored in the cache so that, 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, as the following example shows:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

In the preceding snippet, the `findBook` method is associated with the cache named `books`. Each time the method is called, the cache is checked to see whether the invocation has already been executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation lets multiple names be specified so that more than one cache is being used. In this case, each of the caches is checked before executing the method — if at least one cache is hit, the associated value is returned.

All the other caches that do not contain the value are also updated, even though the cached method was not actually executed.

The following example uses `@Cacheable` on the `findBook` method:

```
@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. 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 than one param is given, return a `SimpleKey` that contains 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, you need to change the strategy.

To provide a different default key generator, you need 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, considered only 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, the target methods are quite likely to have various signatures that cannot be readily 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). Consider the following 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. Furthermore, what if only one of the two is important while the other is not?

For such cases, the `@Cacheable` annotation lets you specify how the key is generated through its `key` attribute. You 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 works for all methods.

The following examples ise various SpEL declarations (if you are not familiar with SpEL, 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 preceding snippets show how easy it is to select a certain argument, one of its properties, or even an arbitrary (static) method.

If the algorithm responsible for generating the key is too specific or if it needs to be shared, you can define a custom `keyGenerator` on the operation. To do so, specify the name of the `KeyGenerator` bean implementation to use, as the following example shows:

```
@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The `key` and `keyGenerator` parameters are mutually exclusive and an operation that specifies both results in an exception.

Default Cache Resolution

The caching abstraction uses a simple `CacheResolver` that retrieves the caches defined at the operation level by using the configured `CacheManager`.

To provide a different default cache resolver, you need to implement the `org.springframework.cache.interceptor.CacheResolver` interface.

Custom Cache Resolution

The default cache resolution fits well for applications that work with a single `CacheManager` and have no complex cache resolution requirements.

For applications that work with several cache managers, you can set the `cacheManager` to use for each operation, as the following example shows:

```
@Cacheable(cacheNames="books", cacheManager="anotherCacheManager") ①
public Book findBook(ISBN isbn) {...}
```

① Specifying `anotherCacheManager`.

You can also replace the `CacheResolver` entirely in a fashion similar to that of replacing key generation. The resolution is requested for every cache operation, letting the implementation actually resolve the caches to use based on runtime arguments. The following example shows how to specify a `CacheResolver`:

```
@Cacheable(cacheResolver="runtimeCacheResolver") ①
public Book findBook(ISBN isbn) {...}
```

① Specifying the `CacheResolver`.

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 results in an exception. as a custom `CacheManager` is 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, you can use the `sync` attribute to instruct the underlying cache provider to lock the cache entry while the value is being computed. As a result, only one thread is busy computing the value, while the others are blocked until the entry is updated in the cache. The

following example shows how to use the `sync` attribute:

```
@Cacheable(cacheNames="foos", sync=true) ①
public Foo executeExpensiveOperation(String id) {...}
```

① Using the `sync` attribute.

This is an optional feature, and your favorite cache library may not support it. All `CacheManager` implementations provided by the core framework support it. See 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, the method is executed every time no matter what values are in the cache or what arguments are used). For example, the following method is cached only if the argument `name` has a length shorter than 32:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32") ①
public Book findBook(String name)
```

① Setting a condition on `@Cacheable`.

In addition to the `condition` parameter, you can use the `unless` parameter to veto the adding of a value to the cache. Unlike `condition`, `unless` expressions are evaluated after the method has been called. To expand on the previous example, perhaps we only want to cache paperback books, as the following example does:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32", unless=
"#result.hardback") ①
public Book findBook(String name)
```

① Using the `unless` attribute to block hardbacks.

The cache abstraction supports `java.util.Optional`, using its content as the cached value only if it is present. `#result` always refers to the business entity and never 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 against a dedicated `context`. In addition to the built-in parameters, the framework provides dedicated caching-related metadata, such as the argument names. The following table describes the items made available to the context so that you can use them for key and conditional computations:

Table 11. 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 the names are not available (perhaps due to having 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` (you can also use `#p0` or `#p<#arg>` notation as an alias).

Name	Location	Description	Example
`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`

The `@CachePut` Annotation

When the cache needs to be updated without interfering with the method execution, you can use the `@CachePut` annotation. That is, the method is always executed and its result is 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. The following example uses the `@CachePut` annotation:

```
@CachePut(cacheNames="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

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 declarations should be avoided. Note also that such conditions should not rely on the result object (that is, the `#result` variable), as these are validated up-front to confirm the exclusion.

The `@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. As opposed to `@Cacheable`, `@CacheEvict` demarcates methods that perform cache eviction (that is, methods that act as triggers for removing data from the cache). Similarly to its sibling, `@CacheEvict` requires specifying one or more caches that are affected by the action, allows a custom cache and key resolution or a condition to be specified, and features an extra parameter (`allEntries`) that indicates whether a cache-wide eviction needs to be performed rather than just an entry eviction (based on the key). The following example evicts all entries from the `books` cache:

```
@CacheEvict(cacheNames="books", allEntries=true) ①
public void loadBooks(InputStream batch)
```

① Using the `allEntries` attribute to evict all entries from the cache.

This option comes in handy when an entire cache region needs to be cleared out. Rather than evicting each entry (which would take a long time, since it is inefficient), all the entries are removed in one operation, as the preceding example shows. Note that the framework ignores any key specified in this scenario as it does not apply (the entire cache is evicted, not only one entry).

You can also indicate whether the eviction should occur after (the default) or before the method executes by using 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 always occur before the method is invoked. This is useful in cases where the eviction does not need to be tied to the method outcome.

Note that `void` methods can be used with `@CacheEvict` - as the methods act as a trigger, the return values are ignored (as they do not interact with the cache). This is not the case with `@Cacheable` which adds or updates data into the cache and, thus, requires a result.

The `@Caching` Annotation

Sometimes, 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` lets multiple nested `@Cacheable`, `@CachePut`, and `@CacheEvict` annotations be used on the same method. The following example uses two `@CacheEvict` annotations:

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key=
"#p0") })
public Book importBooks(String deposit, Date date)
```

The `@CacheConfig` annotation

So far, we have seen that caching operations offer many customization options and that you can set these options for each operation. 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 can be replaced by a single class-level definition. This is where `@CacheConfig` comes into play. The following examples uses `@CacheConfig` to set the name of the cache:

```
@CacheConfig("books") ①
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) {...}
}
```

① Using `@CacheConfig` to set the name of the cache.

`@CacheConfig` is a class-level annotation that allows sharing the cache names, the custom `KeyGenerator`, the custom `CacheManager`, and the custom `CacheResolver`. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization always overrides a customization set on `@CacheConfig`. Therefore, this gives three levels of customizations for each cache operation:

- Globally configured, available for `CacheManager`, `KeyGenerator`.
- At the class level, using `@CacheConfig`.
- At the operation level.

Enabling 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 you can 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
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
https://www.springframework.org/schema/cache/spring-cache.xsd">

        <cache:annotation-driven/>
</beans>
```

Both the `cache:annotation-driven` element and the `@EnableCaching` annotation let you specify various options that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of `@Transactional`.

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.

For more detail about advanced customizations (using Java configuration) that are required to implement `CachingConfigurer`, see the javadoc.

Table 12. Cache annotation settings

XML Attribute	Annotation Attribute	Default	Description
`cache-manager`	N/A (see the `CachingConfigurer` javadoc)	`cacheManager`	The name of the cache manager to use. A default `CacheResolver` is 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.
`cache-resolver`	N/A (see the `CachingConfigurer` javadoc)	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 needs to be specified only as an alternative to the 'cache-manager' attribute.
`key-generator`	N/A (see the `CachingConfigurer` javadoc)	`SimpleKeyGenerator`	Name of the custom key generator to use.

XML Attribute	Annotation Attribute	Default	Description
`error-handler`	N/A (see the `CachingConfigurer` javadoc)	`SimpleCacheErrorHandler`	The name of the custom cache error handler to use. By default, any exception thrown during a cache related operation is thrown back at the client.
`mode`	`mode`	`proxy`	The default mode (`proxy`) processes annotated beans to be proxied by using Spring's AOP framework (following proxy semantics, as discussed earlier, 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`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, 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 AOP advice, see Advice Ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

`<cache:annotation-driven/>` looks for `@Cacheable/@CachePut/@CacheEvict/@Caching` only on beans in the same application context in which it is defined. This means that, if you put `<cache:annotation-driven/>` in a `WebApplicationContext` for a `DispatcherServlet`, it checks for beans only in your controllers, not your services. See the MVC section for more information.

Method visibility and cache annotations

When you use 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 using AspectJ (see the rest of this section) 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 use interface-based proxies. The fact that Java annotations are not inherited from interfaces means that, if you use class-based proxies (`proxy-target-class="true"`) or the weaving-based aspect (`mode="aspectj"`), the caching settings are not recognized by the proxying and weaving infrastructure, and the object is not wrapped in a caching proxy.

In proxy mode (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 that calls another method of the target object) does not lead to actual caching at runtime even if the invoked method is marked with `@Cacheable`. Consider using the `aspectj` mode in this case. Also, the proxy must be fully initialized to provide the expected behavior, so you should not rely on this feature in your initialization code (that is, `@PostConstruct`).

Using Custom Annotations

Custom annotation and AspectJ

This feature works only with the proxy-based approach but can be enabled with a bit of extra effort by 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 lets you 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, which is especially useful if the key or condition are specified or if the foreign imports (`org.springframework`) are not allowed in your code base. Similarly to the rest of the stereotype annotations, you can use `@Cacheable`, `@CachePut`, `@CacheEvict`, and `@CacheConfig` as meta-annotations (that is, annotations that can annotate other annotations). In the following example, we replace a common `@Cacheable` declaration with our own custom annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(cacheNames="books", key="#isbn")
public @interface SlowService {
}
```

In the preceding example, 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)
```

The following example shows the custom annotation with which we can replace the preceding code:

```
@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 earlier, annotation-driven behavior needs to be enabled.

JCache (JSR-107) Annotations

Since version 4.1, Spring's caching abstraction fully supports the JCache standard annotations: `@CacheResult`, `@CachePut`, `@CacheRemove`, and `@CacheRemoveAll` as well as the `@CacheDefaults`, `@CacheKey`, and `@CacheValue` companions. You can use these annotations even 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).

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 13. 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.

Spring	JSR-107	Remark
`@CachePut`	`@CachePut`	While Spring updates the cache with the result of the method invocation, JCache requires that it be passed it as an argument that is annotated with `@CacheValue`. Due to this difference, JCache allows updating the cache before or after the actual method invocation.
`@CacheEvict`	`@CacheRemove`	Fairly similar. `@CacheRemove` supports conditional eviction when the method invocation results in an exception.
`@CacheEvict(allEntries=true)`	`@CacheRemoveAll`	See `@CacheRemove`.
`@CacheConfig`	`@CacheDefaults`	Lets you configure the same concepts, in a similar fashion.

JCache has the notion of `javax.cache.annotation.CacheResolver`, which is identical to the Spring's `CacheResolver` interface, except that JCache supports only 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. See the javadoc of `@CacheResult#cacheName()` for more information.

`CacheResolver` instances are retrieved by a `CacheResolverFactory`. It is possible to customize the factory for each cache operation, as the following example shows:

```
@CacheResult(cacheNames="books", cacheResolverFactory=MyCacheResolverFactory.class) ①
public Book findBook(ISBN isbn)
```

① Customizing the factory for this operation.

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, the following are identical operations, one using Spring's abstraction and the other using JCache:

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")
public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

You can also specify the `CacheKeyResolver` on the operation, similar to how you can specify 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. 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 a parameter). The following caches the exception so that further calls with the same, invalid, ISBN throw the cached exception directly instead of invoking the method again:

```
@CacheResult(cacheName="books", exceptionCacheName="failures"
             cachedExceptions = InvalidIsbnNotFoundException.class)
public Book findBook(ISBN isbn)
```

Enabling JSR-107 Support

You need do nothing specific to enable the JSR-107 support alongside Spring's declarative annotation support. Both `@EnableCaching` and the `cache:annotation-driven` element automatically enable the JCache support if both the JSR-107 API and the `spring-context-support` module are present in the classpath.

Depending on your use case, the choice is basically yours. You can even mix and match services by using the JSR-107 API on some and using Spring's own annotations on others. However, if these services impact the same caches, you should use a consistent and identical key generation implementation.

Declarative XML-based Caching

If annotations are not an option (perhaps due to having no access to the sources or no external code), you can use XML for declarative caching. So, instead of annotating the methods for caching, you can specify the target method and the caching directives externally (similar to the declarative transaction management advice). The example from the previous section can be translated into the following example:

```
<!-- 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 preceding configuration, the `bookService` is made cacheable. The caching semantics to apply are encapsulated in the `cache:advice` definition, which causes the `findBooks` method to be used for putting data into the cache and the `loadBooks` method for evicting data. Both definitions work 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 preceding example, all methods from the `BookService` are considered and the cache advice is applied to them.

The declarative XML caching supports all of the annotation-based model, so moving between the two should be fairly easy. Furthermore, 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 that have 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, you can use the AspectJ pointcut to cherry pick the target methods and apply the appropriate caching functionality. However, through XML, it is easier to apply package or group or interface-wide caching (again, due to the AspectJ pointcut) and to create template-like definitions (as we did in the preceding example by defining the target cache through the `cache:definitions cache` attribute).

Configuring the Cache Storage

The cache abstraction provides several storage integration options. To use them, you need to declare an appropriate `CacheManager` (an entity that controls and manages `Cache` instances and that can be used to retrieve these for storage).

JDK `ConcurrentMap`-based Cache

The JDK-based `Cache` implementation resides under `org.springframework.cache.concurrent` package. It lets you use `ConcurrentHashMap` as a backing `Cache` store. The following example shows how to configure two caches:

```
<!-- 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 preceding snippet 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, persistence capabilities, or eviction contracts.

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 in the `org.springframework.cache.ehcache` package. Again, to use it, you need to declare the appropriate `CacheManager`. The following example shows how to do so:

```
<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 the Spring IoC (through the `ehcache` bean), which is then wired into the dedicated `CacheManager` implementation. Note that the entire Ehcache-specific configuration is read from `ehcache.xml`.

Caffeine Cache

Caffeine is a Java 8 rewrite of Guava's cache, and its implementation is located in the `org.springframework.cache.caffeine` package and provides access to several features of Caffeine.

The following example configures a `CacheManager` that creates the cache on demand:

```
<bean id="cacheManager"
        class="org.springframework.cache.caffeine.CaffeineCacheManager"/>
```

You can also provide the caches to use explicitly. In that case, only those are made available by the manager. The following example shows how to do so:

```
<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 custom `Caffeine` and `CacheLoader`. See the Caffeine documentation for more information about those.

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), see the Spring Data GemFire reference documentation.

JSR-107 Cache

Spring's caching abstraction can also use JSR-107-compliant caches. The JCache implementation is located in the `org.springframework.cache.jcache` package.

Again, to use it, you need to declare the appropriate `CacheManager`. The following example shows how to do so:

```
<bean id="cacheManager"
        class="org.springframework.cache.jcache.JCacheCacheManager"
        p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup  -->
<bean id="jCacheManager" .../>
```

Dealing with Caches without a Backing Store

Sometimes, when switching environments or doing testing, you might have cache declarations without having an actual backing cache configured. As this is an invalid configuration, an exception is thrown at runtime, since the caching infrastructure is unable to find a suitable store. In situations like this, rather than removing the cache declarations (which can prove tedious), you can wire in a simple dummy cache that performs no caching — that is, it forces the cached methods to be executed every time. The following example shows how to do so:

```
<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` in the preceding chains multiple `CacheManager` instances and, through the `fallbackToNoOpCache` flag, adds a no-op cache for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either `jdkCache` or `gemfireCache` (configured earlier in the example) is handled by the no-op cache, which does not store any information, causing the target method to be executed every time.

Plugging-in Different Back-end Caches

Clearly, there are plenty of caching products out there that you can use as a backing store. To plug them in, you need to provide a `CacheManager` and a `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 adapters that map the caching abstraction framework on top of the storage API, as the `ehcache` classes do. Most `CacheManager` classes can use the classes in the `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.

How can I Set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is an abstraction, not a cache implementation. The solution you use might support various data policies and different topologies that other solutions do not support (for example, the JDK `ConcurrentHashMap` — exposing that in the cache abstraction would be useless because there would no backing support). Such functionality should be controlled directly through the backing cache (when configuring it) or through its native API.

Appendix

XML Schemas

This part of the appendix lists XML schemas related to integration technologies.

The `jee` Schema

The `jee` elements deal with issues related to Java EE (Java Enterprise Edition) configuration, such as looking up a JNDI object and defining EJB references.

To use the elements 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 elements in the `jee` 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:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
https://www.springframework.org/schema/jee/spring-jee.xsd">

    <!-- bean definitions here -->

</beans>
```

<jee:jndi-lookup/> (simple)

The following example shows how to use JNDI to look up a data source without the `jee` schema:

```
<bean id="dataSource" 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="dataSource"/>
</bean>
```

The following example shows how to use JNDI to look up a data source with the `jee` schema:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

`<jee:jndi-lookup/>` (with Single JNDI Environment Setting)

The following example shows how to use JNDI to look up an environment variable without `jee`:

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

The following example shows how to use JNDI to look up an environment variable with `jee`:

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>ping=pong</jee:environment>
</jee:jndi-lookup>
```

`<jee:jndi-lookup/>` (with Multiple JNDI Environment Settings)

The following example shows how to use JNDI to look up multiple environment variables without `jee`:

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="sing">song</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

The following example shows how to use JNDI to look up multiple environment variables with `jee`:

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties
format) -->
    <jee:environment>
        sing=song
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

`<jee:jndi-lookup/>` **(Complex)**

The following example shows how to use JNDI to look up a data source and a number of different properties without `jee`:

```
<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.DefaultThing"/>
    <property name="proxyInterface" value="com.myapp.Thing"/>
</bean>
```

The following example shows how to use JNDI to look up a data source and a number of different properties with `jee`:

```
<jee:jndi-lookup id="simple"
        jndi-name="jdbc/MyDataSource"
        cache="true"
        resource-ref="true"
        lookup-on-startup="false"
        expected-type="com.myapp.DefaultThing"
        proxy-interface="com.myapp.Thing"/>
```

`<jee:local-slsb/>` **(Simple)**

The `<jee:local-slsb/>` element configures a reference to a local EJB Stateless Session Bean.

The following example shows how to configures a reference to a local EJB Stateless Session Bean without `jee`:

```
<bean id="simple"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

The following example shows how to configures a reference to a local EJB Stateless Session Bean with `jee`:

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"/>
```

`<jee:local-slsb/>` (Complex)

The `<jee:local-slsb/>` element configures a reference to a local EJB Stateless Session Bean.

The following example shows how to configures a reference to a local EJB Stateless Session Bean and a number of properties without `jee`:

```
<bean id="complexLocalEjb"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.example.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
</bean>
```

The following example shows how to configures a reference to a local EJB Stateless Session Bean and a number of properties with `jee`:

```
<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/>` element configures a reference to a `remote` EJB Stateless Session Bean.

The following example shows how to configures a reference to a remote EJB Stateless Session Bean without `jee`:

```
<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>
```

The following example shows how to configures a reference to a remote EJB Stateless Session Bean with `jee`:

```
<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">
```

The `jms` Schema

The `jms` elements deal with configuring JMS-related beans, such as Spring's Message Listener Containers. These elements are detailed in the section of the JMS chapter entitled JMS Namespace Support. See that chapter for full details on this support and the `jms` elements themselves.

In the interest of completeness, to use the elements 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 elements 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
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jms
https://www.springframework.org/schema/jms/spring-jms.xsd">

    <!-- bean definitions here -->

</beans>
```

Using `<context:mbean-export/>`

This element is detailed in Configuring Annotation-based MBean Export.

The `cache` Schema

You can use the `cache` elements to enable support for Spring's `@CacheEvict`, `@CachePut`, and `@Caching` annotations. It it also supports declarative XML-based caching. See Enabling Caching Annotations and Declarative XML-based Caching for details.

To use the elements 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 elements in the `cache` 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:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
https://www.springframework.org/schema/cache/spring-cache.xsd">

    <!-- bean definitions here -->

</beans>
```