Marshalling XML by Using Object-XML Mappers

Marshalling XML by Using Object-XML Mappers

Introduction

This chapter, describes Spring's Object-XML Mapping support. Object-XML Mapping (O-X mapping for short) is the act of converting an XML document to and from an object. This conversion process is also known as XML Marshalling, or XML Serialization. This chapter uses these terms interchangeably.

Within the field of O-X mapping, a marshaller is responsible for serializing an object (graph) to XML. In similar fashion, an unmarshaller deserializes the XML to an object graph. This XML can take the form of a DOM document, an input or output stream, or a SAX handler.

Some of the benefits of using Spring for your O/X mapping needs are:

- Ease of configuration

- Consistent Interfaces
- Consistent Exception Hierarchy

Ease of configuration

Spring's bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, and so on. You can configure the marshallers as you would any other bean in your application context. Additionally, XML namespace-based configuration is available for a number of marshallers, making the configuration even simpler.

Consistent Interfaces

Spring's O-X mapping operates through two global interfaces: `Marshaller` and `Unmarshaller`. These abstractions let you switch O-X mapping frameworks with relative ease, with little or no change required on the classes that do the marshalling. This approach has the additional benefit of making it possible to do XML marshalling with a mix-and-match approach (for example, some marshalling performed using JAXB and some by XStream) in a non-intrusive fashion, letting you use the strength of each technology.

Consistent Exception Hierarchy

Spring provides a conversion from exceptions from the underlying O-X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. These runtime exceptions wrap the original exception so that no information is lost.

`Marshaller` and `Unmarshaller`

As stated in the introduction, a marshaller serializes an object to XML, and an unmarshaller deserializes XML stream to an object. This section describes the two Spring interfaces used for this purpose.

Understanding `Marshaller`

Spring abstracts all marshalling operations behind the `org.springframework.oxm.Marshaller` interface, the main method of which follows:

```
public interface Marshaller {

    /**
     * Marshal the object graph with the given root into the provided Result.
     */
```

```
    void marshal(Object graph, Result result) throws XmlMappingException, IOException;
}
```

The `Marshaller` interface has one main method, which marshals the given object to a given `javax.xml.transform.Result`. The result is a tagging interface that basically represents an XML output abstraction. Concrete implementations wrap various XML representations, as the following table indicates:

Result implementation	Wraps XML representation
`DOMResult`	`org.w3c.dom.Node`
`SAXResult`	`org.xml.sax.ContentHandler`
`StreamResult`	`java.io.File`, `java.io.OutputStream`, or `java.io.Writer`

NOTE

Although the `marshal()` method accepts a plain object as its first parameter, most `Marshaller` implementations cannot handle arbitrary objects. Instead, an object class must be mapped in a mapping file, be marked with an annotation, be registered with the marshaller, or have a common base class. Refer to the later sections in this chapter to determine how your O-X technology manages this.

Understanding `Unmarshaller`

Similar to the `Marshaller`, we have the `org.springframework.oxm.Unmarshaller` interface, which the following listing shows:

```
public interface Unmarshaller {

    /**
     * Unmarshal the given provided Source into an object graph.
     */
    Object unmarshal(Source source) throws XmlMappingException, IOException;
}
```

This interface also has one method, which reads from the given `javax.xml.transform.Source` (an XML input abstraction) and returns the object read. As with `Result`, `Source` is a tagging interface that has three concrete implementations. Each wraps a different XML representation, as the following table indicates:

Source implementation	Wraps XML representation
`DOMSource`	`org.w3c.dom.Node`
`SAXSource`	`org.xml.sax.InputSource`, and `org.xml.sax.XMLReader`
`StreamSource`	`java.io.File`, `java.io.InputStream`, or `java.io.Reader`

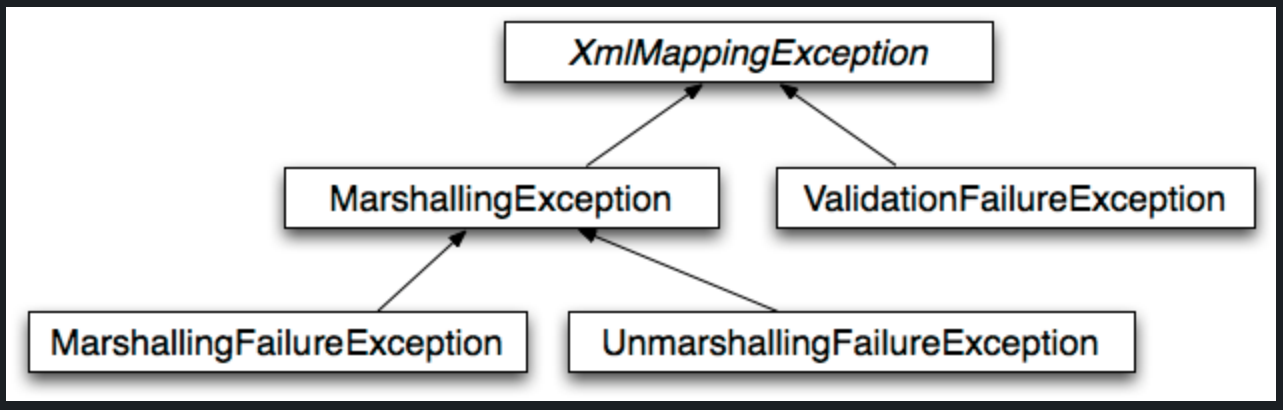
Even though there are two separate marshalling interfaces (`Marshaller` and `Unmarshaller`), all implementations in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and as an unmarshaller in your `applicationContext.xml`.

Understanding `XmlMappingException`

Spring converts exceptions from the underlying O-X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. These runtime exceptions wrap the original exception so that no information will be lost.

Additionally, the `MarshallingFailureException` and `UnmarshallingFailureException` provide a distinction between marshalling and unmarshalling operations, even though the underlying O-X mapping tool does not do so.

The O-X Mapping exception hierarchy is shown in the following figure:



Using `Marshaller` and `Unmarshaller`

You can use Spring's OXM for a wide variety of situations. In the following example, we use it to marshal the settings of a Spring-managed application as an XML file. In the following example, we use a simple JavaBean to represent the settings:

Java Kotlin

```
public class Settings {

    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

The application class uses this bean to store its settings. Besides a main method, the class has two methods: `saveSettings()` saves the settings bean to a file named `settings.xml`, and `loadSettings()` loads these settings again. The following `main()` method constructs a Spring application context and calls these two methods:

Java | Kotlin

```
public class Application {

    private static final String FILE_NAME = "settings.xml";
    private Settings settings = new Settings();
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;

    public void setMarshaller(Marshaller marshaller) {
        this.marshaller = marshaller;
    }

    public void setUnmarshaller(Unmarshaller unmarshaller) {
        this.unmarshaller = unmarshaller;
    }

    public void saveSettings() throws IOException {
        try (FileOutputStream os = new FileOutputStream(FILE_NAME)) {
            this.marshaller.marshal(settings, new StreamResult(os));
        }
    }

    public void loadSettings() throws IOException {
        try (FileInputStream is = new FileInputStream(FILE_NAME)) {
            this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));
        }
    }
```

```
    public static void main(String[] args) throws IOException {
        ApplicationContext appContext =
                new ClassPathXmlApplicationContext("applicationContext.xml");
        Application application = (Application) appContext.getBean("application");
        application.saveSettings();
        application.loadSettings();
    }
}
```

The `Application` requires both a `marshaller` and an `unmarshaller` property to be set. We can do so by using the following `applicationContext.xml`:

```
<beans>
    <bean id="application" class="Application">
        <property name="marshaller" ref="xstreamMarshaller" />
        <property name="unmarshaller" ref="xstreamMarshaller" />
    </bean>
    <bean id="xstreamMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller"/>
</beans>
```

This application context uses XStream, but we could have used any of the other marshaller instances described later in this chapter. Note that, by default, XStream does not require any further configuration, so the bean definition is rather simple. Also note that the `XStreamMarshaller` implements both `Marshaller` and `Unmarshaller`, so we can refer to the `xstreamMarshaller` bean in both the `marshaller` and `unmarshaller` property of the application.

This sample application produces the following `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

XML Configuration Namespace

You can configure marshallers more concisely by using tags from the OXM namespace. To make these tags available, you must first reference the appropriate schema in the preamble of the XML configuration file. The following example shows how to do so:

```
<?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:oxm="http://www.springframework.org/schema/oxm" (1)
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/oxm
        https://www.springframework.org/schema/oxm/spring-oxm.xsd"> (2)
```

(1) Reference the `oxm` schema.

(2) Specify the `oxm` schema location.

The schema makes the following elements available:

- `jaxb2-marshaller`
- `jibx-marshaller`

Each tag is explained in its respective marshaller's section. As an example, though, the configuration of a JAXB2 marshaller might resemble the following:

```
<oxm:jaxb2-marshaller id="marshaller"
contextPath="org.springframework.ws.samples.airline.schema"/>
```

JAXB

The JAXB binding compiler translates a W3C XML Schema into one or more Java classes, a `jaxb.properties` file, and possibly some resource files. JAXB also offers a way to generate a schema from annotated Java classes.

Spring supports the JAXB 2.0 API as XML marshalling strategies, following the `Marshaller` and `Unmarshaller` interfaces described in `Marshaller` and `Unmarshaller`. The corresponding integration classes reside in the `org.springframework.oxm.jaxb` package.

Using `Jaxb2Marshaller`

The `Jaxb2Marshaller` class implements both of Spring's `Marshaller` and `Unmarshaller` interfaces. It requires a context path to operate. You can set the context path by setting the `contextPath` property. The context path is a list of colon-separated Java package names that contain schema derived classes. It also offers a `classesToBeBound` property, which allows you to set an array of classes to be supported by the marshaller. Schema validation is performed by specifying one or more schema resources to the bean, as the following example shows:

```
<beans>
    <bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
```

```
        <property name="classesToBeBound">
            <list>
                <value>org.springframework.oxm.jaxb.Flight</value>
                <value>org.springframework.oxm.jaxb.Flights</value>
            </list>
        </property>
        <property name="schema" value="classpath:org/springframework/oxm/schema.xsd"/>
    </bean>

    ...

</beans>
```

XML Configuration Namespace

The `jaxb2-marshaller` element configures a `org.springframework.oxm.jaxb.Jaxb2Marshaller`, as the following example shows:

```
<oxm:jaxb2-marshaller id="marshaller"
contextPath="org.springframework.ws.samples.airline.schema"/>
```

Alternatively, you can provide the list of classes to bind to the marshaller by using the `class-to-be-bound` child element:

```
<oxm:jaxb2-marshaller id="marshaller">
    <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Airport"/>
    <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Flight"/>
    ...
</oxm:jaxb2-marshaller>
```

The following table describes the available attributes:

Attribute	Description	Required
`id`	The ID of the marshaller	No
`contextPath`	The JAXB Context path	No

JiBX

The JiBX framework offers a solution similar to that which Hibernate provides for ORM: A binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the bind-

ing and compiling the classes, a JiBX binding compiler enhances the class files and adds code to handle converting instances of the classes from or to XML.

For more information on JiBX, see the JiBX web site. The Spring integration classes reside in the `org.springframework.oxm.jibx` package.

Using `JibxMarshaller`

The `JibxMarshaller` class implements both the `Marshaller` and `Unmarshaller` interface. To operate, it requires the name of the class to marshal in, which you can set using the `targetClass` property. Optionally, you can set the binding name by setting the `bindingName` property. In the following example, we bind the `Flights` class:

```
<beans>
    <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
    </bean>
    ...
</beans>
```

A `JibxMarshaller` is configured for a single class. If you want to marshal multiple classes, you have to configure multiple `JibxMarshaller` instances with different `targetClass` property values.

XML Configuration Namespace

The `jibx-marshaller` tag configures a `org.springframework.oxm.jibx.JibxMarshaller`, as the following example shows:

```
<oxm:jibx-marshaller id="marshaller" target-
class="org.springframework.ws.samples.airline.schema.Flight"/>
```

The following table describes the available attributes:

Attribute	Description	Required
`id`	The ID of the marshaller	No
`target-class`	The target class for this marshaller	Yes
`bindingName`	The binding name used by this marshaller	No

XStream

XStream is a simple library to serialize objects to XML and back again. It does not require any mapping and generates clean XML.

For more information on XStream, see the XStream web site. The Spring integration classes reside in the `org.springframework.oxm.xstream` package.

Using `XStreamMarshaller`

The `XStreamMarshaller` does not require any configuration and can be configured in an application context directly. To further customize the XML, you can set an alias map, which consists of string aliases mapped to classes, as the following example shows:

```
<beans>
    <bean id="xstreamMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller">
        <property name="aliases">
            <props>
                <prop key="Flight">org.springframework.oxm.xstream.Flight</prop>
            </props>
        </property>
    </bean>
    ...
</beans>
```

WARNING

WARNING

By default, XStream lets arbitrary classes be unmarshalled, which can lead to unsafe Java serialization effects. As such, we do not recommend using the `XStreamMarshaller` to unmarshal XML from external sources (that is, the Web), as this can result in security vulnerabilities.

If you choose to use the `XStreamMarshaller` to unmarshal XML from an external source, set the `supportedClasses` property on the `XStreamMarshaller`, as the following example shows:

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="supportedClasses" value="org.springframework.oxm.xstream.Flight"/>
    ...
</bean>
```

Doing so ensures that only the registered classes are eligible for unmarshalling.

Additionally, you can register custom converters to make sure that only your supported classes can be unmarshalled. You might want to add a `CatchAllConverter` as the last converter in the list, in addition to converters that explicitly support the domain classes that should be supported. As a result, default XStream converters with lower priorities and possible security vulnerabilities do not get invoked.

NOTE

Note that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As a result, it is rather unsuitable for usage within Web Services.