16. Marshalling XML using O/X Mappers

16.1 Introduction

In this chapter, we will describe Spring's Object/XML Mapping support. Object/XML Mapping, or 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. Spring's bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, etc. The marshallers can be configured as any other bean in your application context. Additionally, XML Schema-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: the `Marshaller` and `Unmarshaller` interface. These abstractions allow you to switch O/X mapping frameworks with relative ease, with little or no changes 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 (e.g. some marshalling performed using JAXB, other using XMLBeans) in a non-intrusive fashion, leveraging 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. As can be expected, these runtime exceptions wrap the original exception so no information is lost.

16.2 Marshaller and Unmarshaller

As stated in the introduction, a *marshaller* serializes an object to XML, and an *unmarshaller* deserializes XML stream to an object. In this section, we will describe the two Spring interfaces used for this purpose.

16.2.1 Marshaller

Spring abstracts all marshalling operations behind the `org.springframework.oxm.Marshaller` interface, the main methods of which is listed below.

```
public interface Marshaller {

    /**
     * Marshals 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`. Result is a tagging interface that basically represents an XML output abstraction: concrete implementations wrap various XML representations, as indicated in the table below.

`Result` implementation	Wraps XML representation
`DOMResult`	`org.w3c.dom.Node`

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

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, marked with an annotation, registered with the marshaller, or have a common base class. Refer to the further sections in this chapter to determine how your O/X technology of choice manages this.

16.2.2 Unmarshaller

Similar to the `Marshaller`, there is the `org.springframework.oxm.Unmarshaller` interface.

```
public interface Unmarshaller {

    /**
     * Unmarshals 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 indicated in the table below.

`Source` implementation	Wraps XML representation

`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`

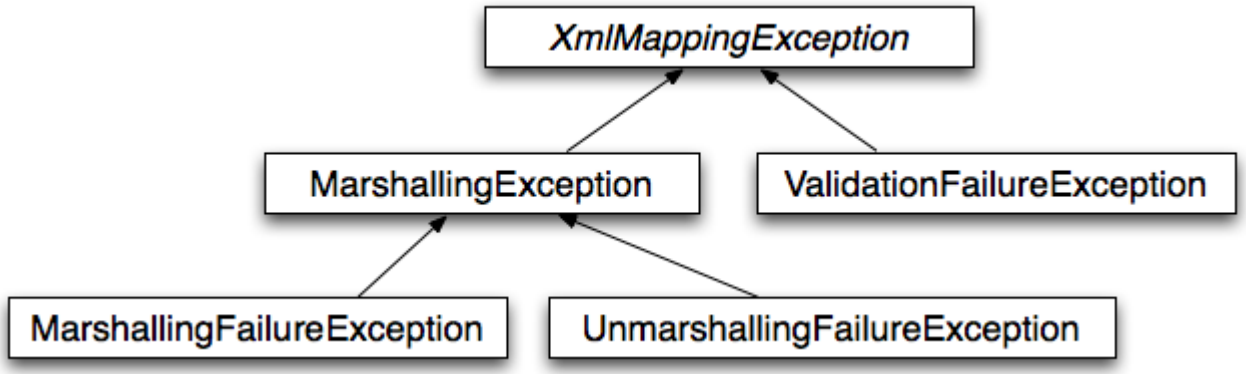
Even though there are two separate marshalling interfaces (`Marshaller` and `Unmarshaller`), all implementations found in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and an unmarshaller in your `applicationContext.xml`.

16.2.3 XmlMappingException

Spring converts exceptions from the underlying O/X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. As can be expected, these runtime exceptions wrap the original exception so 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:



O/X Mapping exception hierarchy

16.3 Using Marshaller and Unmarshaller

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

```
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. A `main()` method constructs a Spring application context, and calls these two methods.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

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 {
        FileOutputStream os = null;
        try {
            os = new FileOutputStream(FILE_NAME);
            this.marshaller.marshal(settings, new StreamResult(os));
        } finally {
            if (os != null) {
                os.close();
            }
        }
    }

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

```
    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 unmarshaller property to be set. We can do so using the following `applicationContext.xml`:

```
<beans>
    <bean id="application" class="Application">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
</beans>
```

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

16.4 XML Schema-based Configuration

Marshallers could be configured more concisely using tags from the OXM namespace. To make these tags available, the appropriate schema has to be referenced first in the preamble of the XML configuration file. Note the 'oxm' related text below:

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

Currently, the following tags are available:

- `jaxb2-marshaller`
- `xmlbeans-marshaller`
- `castor-marshaller`
- `jibx-marshaller`

Each tag will be explained in its respective marshaller's section. As an example though, here is how the configuration of a JAXB2 marshaller might look like:

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

16.5 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 Section 16.2, “Marshaller and Unmarshaller”. The corresponding integration classes reside in the

org.springframework.oxm.jaxb package.

16.5.1 Jaxb2Marshaller

The `Jaxb2Marshaller` class implements both the Spring `Marshaller` and `Unmarshaller` interface. It requires a context path to operate, which you can set using 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 resource to the bean, like so:

```
<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 Schema-based Configuration

The `jaxb2-marshaller` tag configures a `org.springframework.oxm.jaxb.Jaxb2Marshaller`. Here is an example:

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

Alternatively, the list of classes to bind can be provided to the marshaller via the `class-to-be-bound` child tag:

```
<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>
```

Available attributes are:

Attribute	Description	Required
`id`	the id of the marshaller	no
`contextPath`	the JAXB Context path	no

16.6 Castor

Castor XML mapping is an open source XML binding framework. It allows you to transform the data contained in a java object model into/from an XML document. By default, it does not require any further configuration, though a mapping file can be used to have more control over the behavior of Castor.

For more information on Castor, refer to the *Castor web site*. The Spring integration classes reside in the org.springframework.oxm.castor package.

16.6.1 CastorMarshaller

As with JAXB, the `CastorMarshaller` implements both the `Marshaller` and `Unmarshaller` interface. It can be wired up as follows:

```
<beans>

    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
```

```
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
    ...

</beans>
```

16.6.2 Mapping

Although it is possible to rely on Castor's default marshalling behavior, it might be necessary to have more control over it. This can be accomplished using a Castor mapping file. For more information, refer to Castor XML Mapping.

The mapping can be set using the mappingLocation resource property, indicated below with a classpath resource.

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" >
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

XML Schema-based Configuration

The `castor-marshaller` tag configures a `org.springframework.oxm.castor.CastorMarshaller`. Here is an example:

```
                    <oxm:castor-marshaller id="marshaller" mapping-location="classpath:org/springframework/oxm
```

The marshaller instance can be configured in two ways, by specifying either the location of a mapping file (through the mapping-location property), or by identifying Java POJOs (through the target-class or target-package properties) for which there exist corresponding XML descriptor classes. The latter way is usually used in conjunction with XML code generation from XML schemas.

Available attributes are:

Attribute	Description	Required
`id`	the id of the marshaller	no
`encoding`	the encoding to use for unmarshalling from XML	no
`target-class`	a Java class name for a POJO for which an XML class descriptor is available (as generated through code generation)	no
`target-package`	a Java package name that identifies a package that contains POJOs and their corresponding Castor XML descriptor classes (as generated through code generation from XML schemas)	no
`mapping-location`	location of a Castor XML mapping file	no

16.7 XMLBeans

XMLBeans is an XML binding tool that has full XML Schema support, and offers full XML Infoset fidelity. It takes a different approach to that of most other O/X mapping frameworks, in that all classes that are generated from an XML Schema are all derived from `XmlObject`, and contain XML binding information in them.

For more information on XMLBeans, refer to the *XMLBeans web site* . The Spring-WS integration classes reside in the org.springframework.oxm.xmlbeans package.

16.7.1 XmlBeansMarshaller

The `XmlBeansMarshaller` implements both the `Marshaller` and `Unmarshaller` interfaces. It can be configured as follows:

```
<beans>

    <bean id="xmlBeansMarshaller" class="org.springframework.oxm.xmlbeans.XmlBeansMarshaller" />
```

```
    ...

</beans>
```

Note that the `XmlBeansMarshaller` can only marshal objects of type `XmlObject`, and not every `java.lang.Object`.

XML Schema-based Configuration

The `xmlbeans-marshaller` tag configures a `org.springframework.oxm.xmlbeans.XmlBeansMarshaller`. Here is an example:

```
<oxm:xmlbeans-marshaller id="marshaller"/>
```

Available attributes are:

Attribute	Description	Required
`id`	the id of the marshaller	no
`options`	the bean name of the XmlOptions that is to be used for this marshaller. Typically a `XmlOptionsFactoryBean` definition	no

16.8 JiBX

The JiBX framework offers a solution similar to that which JDO provides for ORM: a binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the binding 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, refer to the *JiBX web site*. The Spring integration classes reside in the org.springframework.oxm.jibx package.

16.8.1 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 using the bindingName property. In the next sample, 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>

    ...
```

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

XML Schema-based Configuration

The `jibx-marshaller` tag configures a `org.springframework.oxm.jibx.JibxMarshaller`. Here is an example:

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

Available attributes are:

Attribute	Description	Required
`id`	the id of the marshaller	no

Attribute	Description	Required
`target-class`	the target class for this marshaller	yes
`bindingName`	the binding name used by this marshaller	no

16.9 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, refer to the *XStream web site*. The Spring integration classes reside in the org.springframework.oxm.xstream package.

16.9.1 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:

```
<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>
```

By default, XStream allows for arbitrary classes to be unmarshalled, which can result in security vulnerabilities. As such, it is recommended to set the supportedClasses property on the `XStreamMarshaller`, like so:

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

This will make sure 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 that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As such, it is rather unsuitable for usage within Web services.