Apache Isis Maven plugin

Table of Contents

Chapter 1. Apache Isis Maven plugin

This reference guide describes the goals provided by Apache Isis' Maven plugin.

1.1. Other Guides

Apache Isis documentation is broken out into a number of user, reference and "supporting procedures" guides.

The user guides available are:

- Fundamentals
- Wicket viewer
- Restful Objects viewer
- DataNucleus object store
- Security
- Testing
- Beyond the Basics

The reference guides are:

- Annotations
- Domain Services
- Configuration Properties
- Classes, Methods and Schema
- Apache Isis Maven plugin (this guide)
- Framework Internal Services

The remaining guides are:

- Developers' Guide (how to set up a development environment for Apache Isis and contribute back to the project)
- Committers' Guide (release procedures and related practices)

Chapter 2. Introduction

The Apache Isis Maven plugin defines three goals:

- `validate`

  Use to verify at build time that the metamodel of an application is valid. This runs the `MetaModelValidator` that is also run when the application is started up.

- `swagger`

  Uses the `SwaggerService` to generate Swagger spec files that describe the public and/or private RESTful APIs exposed by the RestfulObjects viewer.

- `xsd`

  Uses the `JaxbService` to generate XSD schemas from any JAXB-annotated view models/DTOs.

  This is instead of and preferable to using the JAXB schemagen tool, because it uses the framework's support (via `@XmlJavaTypeAdapter`) to translate any references to domain objects into `OidDto`s (as defined by the Apache Isis common schema).

The `validate` goal is by default bound to the `test` phase, and the `swagger` goal is by default bound to the `package` phase; both are typically of your application's `dom` sub-module. The `xsd` goal meanwhile defaults to the `generate-resources` phase, and this is generally used in a completely separate sub-module. An example can be found in the (non-ASF) Isis addons' todoapp example app; the separate submodule that uses the `xsd` goal is (also) called `todoapp-xsd`.

All of these goals require an `AppManifest` to point the plugin at, so that it knows how to bootstrap an Isis runtime. This is discussed below, followed by sections on configuring the two goals.

2.1. `AppManifest`

As noted in the introduction, all the goals require an `AppManifest` to point the plugin at, so that it knows how to bootstrap an Isis runtime.

This can be extremely minimal; it isn't necessary to use the main `AppManifest` (in the `app` module) used to bootstrap the application, you can instead use a cut-down one. This then allows the plugins to be run during the build of the `dom` module, rather than having to run in the context of the `integtest` module.

For example, the SimpleApp's manifest is:

```
package domainapp.dom;
...
public class DomainAppDomManifest implements AppManifest {
    @Override
    public List<Class<?>> getModules() {
        return Arrays.asList(
                DomainAppDomainModule.class  // domain (entities and repositories)
        );
    }
    @Override
    public List<Class<?>> getAdditionalServices() { return Collections.emptyList(); }
    @Override
    public String getAuthenticationMechanism() { return null; }
    @Override
    public String getAuthorizationMechanism() { return null; }
    @Override
    public List<Class<? extends FixtureScript>> getFixtures() { return null; }
    @Override
    public Map<String, String> getConfigurationProperties() { return null; }
}
```

where `DomainAppDomainModule` simply identifies the package for the manifest to search under:

```
package domainapp.dom;
public final class DomainAppDomainModule { }
```

The downside of using a minimal `AppManifest` in the `dom` module is that any contributed actions/associations will be ignored.

We recommend the following:

- run the `validate` goal in the `dom` submodule; this will give early warning if there are any syntactic errors in the model, eg orphaned supporting methods
- run the `swagger` goal in the `integtest` submodule; this ensures that the generated Swagger schema definition files correctly include any contributed actions/associations.
- run the `xsd` plugin in a new `xsd` submodule; contributed actions are irrelevant for this particular goal; having a separate submodule allows the configuration of both the `xsd` goal (to generate the XSD schemas) and any other XSD-related configuration to be kept in a single place.

The SimpleApp archetype reflects these recommendations for the `validate` and `swagger` goals. You can find an example of the `xsd` plugin in the (non-ASF) Isis addons' todoapp application.

Chapter 3. `validate` goal

The Apache Isis programming model requires that a number of naming conventions are followed.

For example, the validator will detect any orphaned supporting methods (eg `hideXxx()`) if the corresponding property or action has been renamed or deleted but the supporting method was not also updated. Another example is that a class cannot have a title specified both using `title()` method and also using `@Title` annotation.

When running the application these are enforced by the `MetaModelValidator` component that detects these errors, failing fast.

The purpose of the `validate` goal of the `isis-maven-plugin` is to enforce these naming conventions at build time, typically enforced by way of a continuous integration server.

The `validate` goal defines a single property:

- `appManifest` - fully qualified class name for the app manifest used to bootstrap the application (see discussion above)

The sections below explain how to configure the plugin within an app.

The instructions given here relate to `1.10.0`. This goal was also released for `1.9.0`, but with a slightly different configuration; see the final section for differences.

3.1. `dom` submodule

Update the `pom.xml` (we recommend in your project's `dom` module, and with a minimal AppManifest):

```
<profile>
    <id>isis-validate</id>
    <activation>
        <property>
            <name>!skip.isis-validate</name>
①
        </property>
    </activation>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.isis.tool</groupId>
                <artifactId>isis-maven-plugin</artifactId>
                <version>${isis.version}</version>
②
                <configuration>
                    <appManifest>domainapp.dom.DomainAppDomManifest</appManifest>
③
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>simpleapp-dom</artifactId>
④
                        <version>${project.version}</version>
                    </dependency>
                    <!-- workaround to avoid conflict with plexus-default -->
                    <dependency>
                        <groupId>com.google.guava</groupId>
                        <artifactId>guava</artifactId>
                        <version>16.0.1</version>
                    </dependency>
                </dependencies>
                <executions>
                    <execution>
                        <phase>test</phase>
                        <goals>
                            <goal>validate</goal>
⑤
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

① the profile is active by default, though can be disabled using `-Dskip.isis-validate`

② set to `1.10.0` (or any later version)

③ the manifest discussed previously; adjust as required

④ the `dom` module for the project; adjust as required

⑤ binds the plugin's `validate` goal to the Maven `test` lifecycle phase (ie the goal will be called when `mvn test` is run).

3.2. To run

The plugin is activated by default, so is run simply using:

```
mvn test
```

This will run any tests, and then also - because the plugin is activated by the `isis-validate` property and bound to the `test` phase, will run the plugin's `validate` goal.

If for any reason you want to disable the validation, use:

```
mvn test -Dskip.isis-validate
```

3.3. Example of failure

In the SimpleApp application the `SimpleObject` defines an `updateName` action. This has a supporting method:

```
public SimpleObject updateName( ... ) { ... }
public String default0UpdateName() { ... }
```

We can introduce an error by misspelling the supporting method, for example:

```
public String default0XUpdateName() { ... }
```

Running `mvn test` then generates this output:

```
[error]
[error]
[error]
[error] domainapp.dom.simple.SimpleObject#default0XUpdateName: has prefix default, is
probably a supporting method for a property, collection or action.  If the method is
intended to be an action, then rename and use @ActionLayout(named="...") or ignore
completely using @Programmatic
[error]
[error]
[error]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Simple App ..................................... SUCCESS [  0.087 s]
[INFO] Simple App DOM ................................. FAILURE [  4.182 s]
[INFO] Simple App Fixtures ............................ SKIPPED
[INFO] Simple App Application ......................... SKIPPED
[INFO] Simple App Integration Tests ................... SKIPPED
[INFO] Simple App Webapp .............................. SKIPPED
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] ...
[ERROR] Failed to execute goal org.apache.isis.tool:isis-maven-plugin:1.13.0:validate
(default) on project simpleapp-dom: 1 problems found. -> [Help 1]
```

If one were to attempt to run the application, the same error would appear in the log files on startup (and the application would not boot).

3.4. Custom validation rules

It is also possible to customize the validation, explained here. For example, you could enforce project-specific conventions by implementing a custom `MetaModelValidator`, and registering using a configuration property.

To support this using `AppManifest`s, override its `getConfigurationProperties()` method:

```
public class DomainAppDomManifest implements AppManifest {
    ...
    public Map<String, String> getConfigurationProperties() {
        final Map<String, String> map = Maps.newTreeMap();

map.put("isis.reflector.validator","com.mycompany.myapp.MyMetaModelValidator");
        return map;
    }
}
```

3.5. 1.9.0 version

The `1.9.0` version of the plugin requires slightly different configuration. Rather than using an `AppManifest`, instead the configuration directory containing `isis.properties` is specified:

```
<profile>
    <id>isis-validate</id>
    <activation>
        <activeByDefault>false</activeByDefault>
①
    </activation>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.isis.tool</groupId>
                <artifactId>isis-maven-plugin</artifactId>
                <version>1.9.0</version>
                <configuration>
                    <isisConfigDir>../webapp/src/main/webapp/WEB-INF</isisConfigDir>
②
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>org.apache.isis.example.application</groupId>
                        <artifactId>simpleapp-dom</artifactId>
                        <version>1.9.0</version>
                    </dependency>
                    <!-- workaround to avoid conflict with plexus-default -->
                    <dependency>
                        <groupId>com.google.guava</groupId>
                        <artifactId>guava</artifactId>
                        <version>16.0.1</version>
                    </dependency>
                </dependencies>
                <executions>
                    <execution>
                        <phase>test</phase>
                        <goals>
                            <goal>validate</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

① activated using the `-P` profile flag rather than a `-D` system property

② specify the `isisConfigDir` directory (containing the `isis.properties` file).

To use the `1.9.0` version, use:

```
mvn -P isis-validate test
```

Note that the `isisConfigDir` property was removed in `1.10.0`; only the `AppManifest` approach is supported.

Chapter 4. `swagger` goal

The `swagger` goal of the `isis-maven-plugin` uses the `SwaggerService` to generate Swagger spec files to describe the public and/or private RESTful APIs exposed by the RestfulObjects viewer.

These spec files, once generated, can then be used in the build pipeline to generate client-side stubs, typically using Swagger's own swagger-codegen-maven plugin.

The `swagger` goal defines the following properties:

- `appManifest` - fully qualified class name for the app manifest used to bootstrap the application (see discussion above)
- `fileNamePrefix` - (optional) a prefix to the generated file names (is suffixed by the requested visibilities, see below).

  Defaults to `swagger`.

- `visibilities` - (optional) list of required visibilities.

  Defaults to [`PUBLIC`, `PRIVATE`] (meaning that two spec files will be created).

- `format` - (optional) which format to generate, either `JSON` or `YAML`.

  Defaults to `JSON`.

- `output` - (optional) subdirectory under the `target` directory to generate the swagger schema definition files

  Defaults to `generated-resources/isis-swagger`

4.1. `integtest` submodule

Update the `pom.xml` (in your project's `integtest` module):

```
<profile>
    <id>isis-swagger</id>
    <activation>
        <property>
            <name>!skip.isis-swagger</name>                                  ①
        </property>
    </activation>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.isis.tool</groupId>
                <artifactId>isis-maven-plugin</artifactId>
                <version>${isis.version}</version>                           ②
                <configuration>
                    <appManifest>domainapp.app.DomainAppAppManifest</appManifest>  ③
                    <visibilities>                                           ④
                        <visibility>PUBLIC</visibility>
                        <visibility>PRIVATE</visibility>
                    </visibilities>
                    <format>JSON</format>                                    ⑤
                    <fileNamePrefix>swagger</fileNamePrefix>                 ⑤
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>simpleapp-dom</artifactId>               ⑥
                        <version>${project.version}</version>
                    </dependency>
                    <dependency>
                        <groupId>com.google.guava</groupId>
                        <artifactId>guava</artifactId>
                        <version>16.0.1</version>
                    </dependency>
                </dependencies>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>swagger</goal>                             ⑦
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

① the profile is active by default, though can be disabled using `-Dskip.isis-swagger`

② set to `1.11.0` (or any later version)

③ the manifest discussed previously; adjust as required

④ the visibilities to create (one swagger spec file per visibility listed)

⑤ which file format to generate the spec files as.

⑥ the `dom` module for the project; adjust as required

⑦ binds the plugin's `swagger` goal to the Maven `package` lifecycle phase (ie the goal will be called when `mvn package` is run).

4.2. To run

The plugin is activated by default, so is run simply using:

```
mvn package
```

Chapter 5. xsd goal

The xsd goal of the isis-maven-plugin uses the JaxbService to generate XSD schemas from any JAXB-annotated view model/DTOs.

This is instead of and preferable to using the JAXB schemagen tool, because it uses the framework's support (via @XmlJavaTypeAdapter) to translate any references to domain objects into OidDtos (as defined by the Apache Isis common schema).

The xsd goal defines the following properties:

- appManifest - fully qualified class name for the app manifest used to bootstrap the application (see discussion above)
- jaxbClasses - a list of JAXB-annotated view model classes;
- output - (optional) subdirectory under the target directory to generate the XSDs

  Defaults to generated-resources/isis-xsd

- separate - (optional) whether to create separate directories for each JAXB-class.

  Defaults to false. Most DTO classes will reference one another or the common schema. Normally it's fine to merge all these XSDs together. This property, if set, results in each a separate directory for each generation of its XSD or XSDs.

- commonSchemas - (optional) whether to also generate the isis common schema(s).

  Defaults to false; if set then the call to JaxbService will set IsisSchemas.INCLUDE flag.

As a convenience to any (Java) consumers, the XSDs generated from the view models can then in turn be generated into DTOs. The original view models and these DTOs are similar but not identical: while the view models can only be used within the Isis application (they may reference underlying domain entities) whereas the DTO classes generated from the XSDs can be used standalone, eg by a Java subscriber running on an ESB such as Apache Camel.

The rest of this section explains how to configure a new xsd submodule that uses the isis-maven-plugin along with other standard plugins in order to generate both XSDs and DTOs. The pom.xml described below uses Maven profiles to separate out these two responsibilities.

5.1. xsd submodule

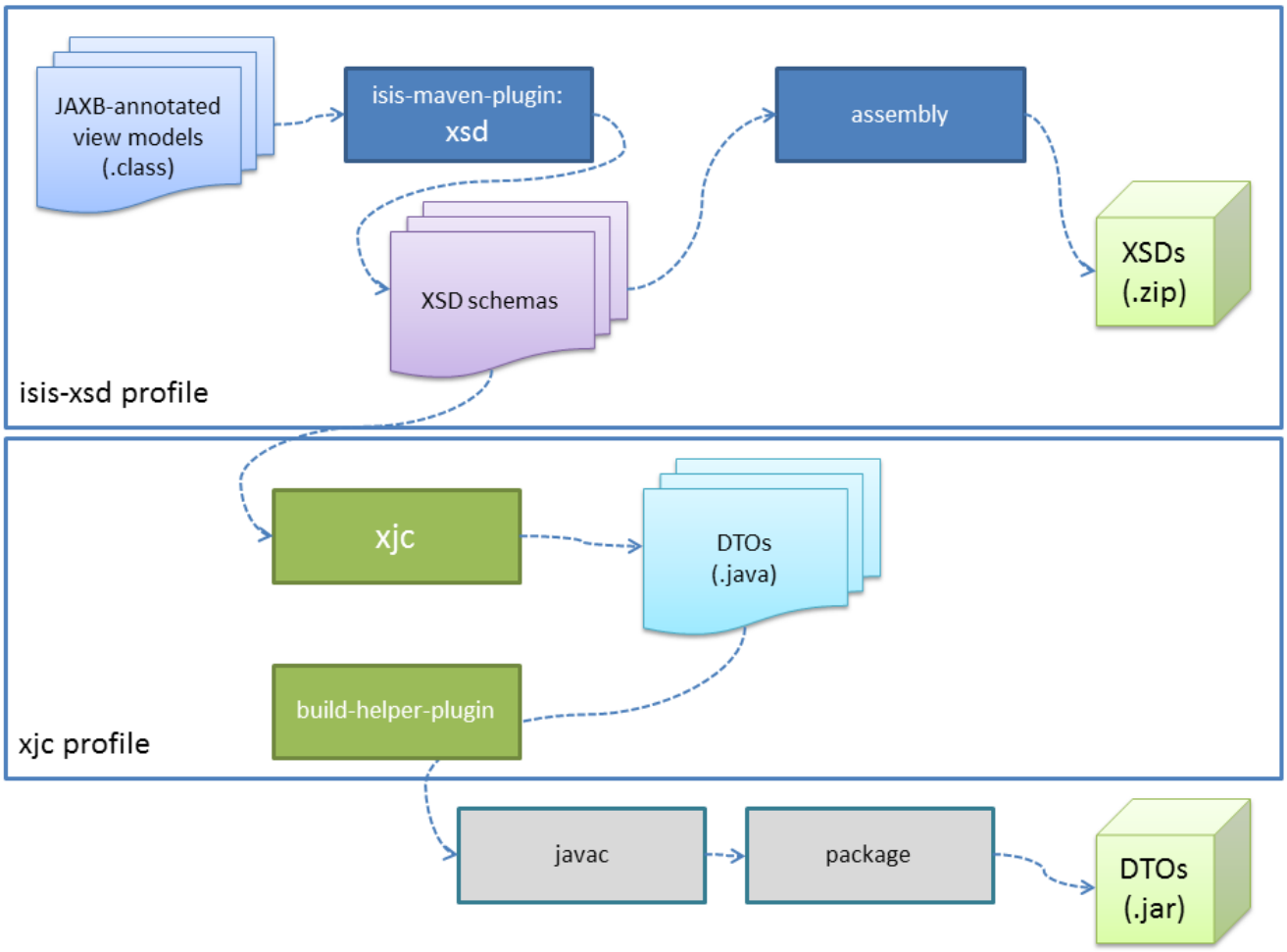
We recommend creating a new submodule that will perform the following build steps:

- run the xsd goal (in the generate-resources phase) to generate the XSDs from the specified view model/DTOs
- use the maven-assembly-plugin to bundle the generated XSD files into a zip file.
- use the xjc-gen to generate corresponding DTO classes from the XSDs.

  These are *not* the same as the original view models; they are provided as a convenience for

subscribers to marshall XML documents into Java classes, but running as a standalone process (not part of the Isis app)

These two main responsibilities can then be placed into separate Maven profiles, for better modularity. The diagram below shows the overall design:



For example, here is the `pom.xml` file for the (non-ASF) Isis addons' todoapp example app's `todoapp-xsd` submodule.

First, the usual boilerplate:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.mycompany</groupId>
        <artifactId>todoapp</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </parent>

    <artifactId>todoapp-xsd</artifactId>
    <name>Isis Addons ToDoApp XSD</name>

    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>todoapp-app</artifactId>    ①
        </dependency>
    </dependencies>

    <profiles>
        <profile>
            <id>isis-xsd</id>                       ②
            ...
        </profile>
        <profile>
            <id>xjc</id>                            ③
            ...
        </profile>
    </profiles>
</project>
```

① depends on the rest of the application's modules

② XSD generation, to run the `xsd` goal and then assemble into a zip file; within a profile for modularity; see section below

③ XJC generation, to run the `xjc` to generate Java DTO classes from XSDs; within a profile for modularity; see section below

The sections below flesh out the gaps.

5.1.1. XSD profile

The `isis-xsd` profile runs the `xsd` goal of the `isis-maven-plugin`; these are then zipped up by the assembly plugin:

```
<profile>
```

```
    <id>isis-xsd</id>
    <activation>
        <property>
            <name>!skip.isis-xsd</name>
①
        </property>
    </activation>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.isis.tool</groupId>
                <artifactId>isis-maven-plugin</artifactId>
                <version>${isis.version}</version>
                <configuration>
                    <appManifest>todoapp.dom.ToDoAppDomManifest</appManifest>
②
                    <jaxbClasses>
③
                        <jaxbClass>
todoapp.app.viewmodels.todoitem.v1_0.ToDoItemDto</jaxbClass>
                        <jaxbClass>
todoapp.app.viewmodels.todoitem.v1_1.ToDoItemDto</jaxbClass>
                    </jaxbClasses>
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>todoapp-dom</artifactId>
                        <version>${project.version}</version>
                    </dependency>
                    <dependency>
④
                        <groupId>com.google.guava</groupId>
                        <artifactId>guava</artifactId>
                        <version>16.0.1</version>
                    </dependency>
                </dependencies>
                <executions>
                    <execution>
                        <phase>generate-sources</phase>
⑤
                        <goals>
                            <goal>xsd</goal>
⑥
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
⑦
```

```
                <version>2.5.3</version>
                <configuration>
                    <descriptor>src/assembly/dep.xml</descriptor>
⑧
                </configuration>
                <executions>
                    <execution>
                        <id>create-archive</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

① enabled *unless* `skip.isis-xsd` property specified

② specify the app manifest to bootstrap the Isis runtime within the maven plugin

③ enumerate all JAXB-annotated view models

④ workaround to avoid conflict with plexus-default

⑤ by default is bound to `generate-resources`, but bind instead to `generate-sources` if also running the `xjc` profile: the XSD are an input to `xjc`, but it is bound by default to `generate-sources` and the `generate-sources` phase runs before the `generate-resources`.

⑥ run the `xsd` goal

⑦ define the assembly plugin

⑧ assembles the XSD schemas into a zip file, as defined by the `dep.xml` file (see below).

```
The `dep.xml` file, referenced by the `assembly` plugin, is defined as:
```

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2
                              http://maven.apache.org/xsd/assembly-1.1.2.xsd">
    <id>xsd</id>
    <formats>
        <format>zip</format>
    </formats>
    <fileSets>
        <fileSet>
            <directory>${project.build.directory}/generated-resources/isis-
xsd</directory>      ①
            <outputDirectory>/</outputDirectory>
        </fileSet>
    </fileSets>
</assembly>
```

① the location that the `xsd` goal writes to.

5.1.2. XJC profile

The `xjc` profile reads the XSD generated by the `xsd` goal, and from it generates Java DTOs. Note that this isn't round-tripping: the original view model is only for use within the Isis app, whereas the DTO generated from the XSDs is for use in a standalone context, eg in a Java subscriber on an event bus.

The `xjc` profile is defined as:

```
<profile>
    <id>xjc</id>
    <activation>
        <property>
            <name>!skip.xjc</name>
①
        </property>
    </activation>
    <build>
        <plugins>
            <plugin>
                <groupId>org.jvnet.jaxb2.maven2</groupId>
                <artifactId>maven-jaxb2-plugin</artifactId>
                <version>0.12.3</version>
                <executions>
                    <execution>
                        <id>xjc-generate</id>
                        <phase>generate-sources</phase>
                        <goals>
                            <goal>generate</goal>
```

```
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <removeOldOutput>true</removeOldOutput>
                    <schemaDirectory>
②
                        target/generated-resources/isis-
xsd/viewmodels.app.todoapp/todoitem
                    </schemaDirectory>
                    <schemaIncludes>
③
                        <schemaInclude>v1_0/todoitem.xsd</schemaInclude>
                        <schemaInclude>v1_1/todoitem.xsd</schemaInclude>
                    </schemaIncludes>
                    <catalog>src/main/resources/catalog.xml</catalog>
④
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
⑤
                <version>1.9.1</version>
                <executions>
                    <execution>
                        <id>add-source</id>
                        <phase>generate-sources</phase>
                        <goals>
                            <goal>add-source</goal>
                        </goals>
                        <configuration>
                            <sources>
                                <source>target/generated-sources/xjc</source>
⑥
                            </sources>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

① enabled *unless* `skip.xjc` property specified

② specifies the directory that the XSD schemas were generated to by the `isis-maven-plugin`

③ specify each of the XSDs to be processed

④ catalog file indicates the location of the referenced common schema XSDs.

⑤ the `build-helper-maven-plugin` adds the Java source generated by the `xjc` plugin so that it can be

compiled and packaged as any other code

⑥ the location that the `xjc` plugin generates its source code.

The referenced `catalog.xml` file instructs the `xjc` plugin how to resolve referenced schema locations. Only a reference for the Apache Isis common schema is likely to be needed:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog
            PUBLIC "-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN"
            "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <public publicId="http://isis.apache.org/schema/common"
            uri="http://isis.apache.org/schema/common/common.xsd"/>
①
</catalog>
```

① resolve the common schema from the Apache Isis website

5.2. To run

The plugin is activated by default, so is run simply using:

```
mvn package
```

This will generate the XSDs, the DTOs from the XSDs, and package up the XSDs into a ZIP file and the generated DTO class files into a regular JAR package.

If for any reason you want to disable the generation of the DTOs, use:

```
mvn package -Dskip.xjc
```

If you want to disable the generation of both the XSDs and the DTOs, use:

```
mvn package -Dskip.xjc -Dskip.isis-xsd
```