**Administrative Office of the Courts**

**Supreme Court of New Mexico**

****

**Judicial Information Division (JID)**

**Odyssey API Library Generator**

**Overview**

**Version 2.0**

Update: 10/2018

Contents

[Table of Figures 3](#_Toc526431697)

[Introduction 4](#_Toc526431698)

[How do APIs work? 4](#_Toc526431699)

[Implementing the Library 4](#_Toc526431700)

[POM (maven build) files 6](#_Toc526431701)

[Finding Conflicts in XSD Schema files 7](#_Toc526431702)

[Resolving Conflicts found in Odyssey release 2017 APIs (XSDs) 7](#_Toc526431703)

[Resolving Conflicts Automatically 9](#_Toc526431704)

[Builder (Conflict Solver) 10](#_Toc526431705)

[Hand Written Classes (invoker) 12](#_Toc526431706)

[Final Notes 13](#_Toc526431707)

# Table of Figures

[Figure 1 - API Library Building Process 5](#_Toc526169760)

[Figure 2 - MULTI MODULE MAVEN PROJECT 5](#_Toc526169761)

[Figure 3 - CONFLICTS BETWEEN ADDADHOCHEARING.XSD AND ADDATTORNEY.XSD 7](#_Toc526169762)

[Figure 4 - Nested Element Naming Conflict on AddProsecutorCaseChargeSharedTypes.xsd 7](#_Toc526169763)

[Figure 5 - Type Definitions that Include White Spaces 8](#_Toc526169764)

[Figure 6 - Missed Type Definition on AmendJudgmentEventResult.xsd 8](#_Toc526169765)

[Figure 7 - Missed Include Statements on AddProsecutorCaseChargeSharedTypes.xsd 8](#_Toc526169766)

[Figure 8 - Calculated Paths from AddProsecutorCaseChargeSharedTypes.xsd 9](#_Toc526169767)

[Figure 9 - Identifying Dependencies across Schema Files 10](#_Toc526169768)

[Figure 10 - Episode Objects Associated to Schema Files 10](#_Toc526169769)

[Figure 11 - Modifications for BaseTypes.xsd 10](#_Toc526169770)

[Figure 12 - Modifications for AddCaseEvent.xsd 11](#_Toc526169771)

[Figure 13 - Adding Prefixes 11](#_Toc526169772)

[Figure 14 - FindCaseByCaseNumber Created by the API Library 12](#_Toc526169773)

[Figure 15 - FindCaseByCaseNumber Message Generated by the Invoker 12](#_Toc526169774)

# Introduction

Odyssey’s Application Programming Interfaces (APIs) are web services that allow users to add, update, and query Odyssey data by using message based interfaces. Basically, each Odyssey API has two parts: ***Request*** and ***Response*** messages. ***Request*** messages are used invoke an API in Odyssey, whereas ***Response*** messages return requested data and/or a message to provide status regarding whether or not an action was successfully executed in Odyssey. Each API specification is provided by a schema (a pair of XSD files; one for the request and one for the response) provided by Tyler Technologies.

Building programs using Odyssey APIs requires significant XML manipulation, time, and effort from the programmer. Having a shared API library in a JAR file to facilitate this tedious low-level coding will reduce the programming effort. Using the shared library described in this document, creating Odyssey integrations should be easier, faster, and cheaper. Also, the tools provided include everything needed to produce new versions of the shared API library; the builder will facilitate creating new versions of the shared library e.g. when Tyler Technologies releases new sets of XSD files for future Odyssey releases.

# How do APIs work?

APIs essentially provide the language and contract for interacting with large system like Odyssey. Tyler’s APIs use SOAP to access web services, and rely on XML for transferring messages. To use Odyssey APIs, an endpoint is created to represent one end of a communication channel with Odyssey. Endpoint creation requires parsing web services description language files (WSDL files) for both the APIs WSDL and the WSDL of the Odyssey Integration (UDDI). These WSDLs are located by URLs and they provide descriptions of how the services provided by Tyler’s Odyssey APIs can be called. The ***wsimport*** command provided by the Java API for XML Web Services **(JAX-WS)** is able to process WSDLs and generates web service artifacts that include client support for accessing APIs web services.

Besides WSDL processing, the shared library will facilitate the exchange of XML messages (parameters) and returned data structures as described in XSD files (or schema) for a given Odyssey API. Thus, it is important to be able to read and write messages that adhere to these XSD specifications. The ***xjc*** command included in the Java Architecture for XML Binding ***(JAXB)*** allows Java programmers to streamline XML processing by mapping Java classes to XML representations and vice versa. It is able to marshal objects into XML messages and also it performs the inverse process – known as *unmarshalling* -- XML messages are automatically mapped into Java objects.

In short, our library will leverage JAX-WS to process Odyssey API and Integration WSDLs URLs in order to create client-side web service classes. These client-side web service classes can be used to invoke Odyssey APIs using the SOAP protocol over http. In addition, the shared library will leverage JAXB to generate classes that can marshall/unmarshall XML messages to/from java objects, so as to facilitate sending API requests (and receiving responses) from Odyssey.

# Implementing the Library

To create the API library, we use a two-step process as depicted in Figure 1. During the first-step, the ***wsimport*** command (JAX-WS) is used on the Odyssey WSDL URLs to create the web service client support, and the ***xjc*** command (JAXB) is used on Tyler’s schema files (XSD files) to create binding classes that are able to marshal and un-marshal objects for XML messages. Processing Odyssey’s WSDLs is a very straightforward process that should be free of errors, but processing Odyssey API XSD files requires some intermediate steps, as there are collisions and conflicts across XSD files, and they must be corrected before the ***xjc*** command can be successfully invoked. The second step basically collects all classes generated from the first step (***wsimport***) and stores them into a single Jar file in the local Maven repository. Notice that some hand-written Java classes are included in the API library because resolving conflicts in the original XSD files introduces some other minor problems that must be corrected at run-time, when API messages are exchanged between user applications and Odyssey.

Step 1 –OdyApiGen-Builder

Step 2 – OdyApiGen-Library

Local Maven Repository

Odyssey API Library (jar)

Hand-written

Java Classes (invoker)

Builder

(conflict solver)

Odyssey API

XSD files

Included as

Maven

Dependency

Generated Java Classes

(JAXB annotated)

Customized API

XSD files

xjc

OdyApiGen-Example Project

Generated Java Classes

(JAX-WS bindings)

Odyssey WSDL URLs

wsimport

Figure - API Library Building Process

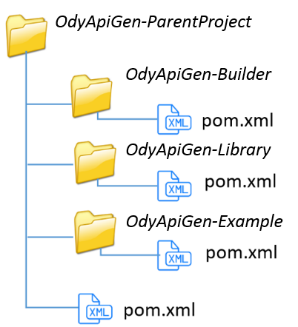
For implementing the API library a two module Maven project was used. One module per step; the first *module (or project)* is labeled ***OdyApiGen-Builder*** whereas the second is called ***OdyApiGen-Library***. Besides these two modules, another project was included, called ***OdyApiGen-Example***, which is for testing and demonstration of the API library. Figure 2 shows the distribution of the two modules and the ***OdyApiGen-Example*** project inside the main Maven project called ***OdyApiGen-ParentProject***; notice how each module/project holds a POM file.

Figure - MULTI MODULE MAVEN PROJECT

# POM (maven build) files

The ***POM*** file under the ***OdyApiGen-ParentProject*** directory not only defines global properties, dependencies, and plug-ins used by the modules but it also has (build) targets for cleaning up ***OdyApiGen-Builder*** and ***OdyApiGen-Library*** modules during the maven *clean* phase. Within the pom.xml file, there is a section ***Parameters to be configured.*** This section is where the directory of the Odyssey API schema files (i.e. the location of Odyssey XSD), URI of the API WSDL, and URI of the Odyssey Integration Kit are specified. Users of the ***OdyApiGen-ParentProject*** must configure these properties according to their local Odyssey installations/environments prior to executing the maven build.

The ***OdyApiGen-Builder POM*** file copies resources during the Maven compile phase. During the maven build’s *generates-sources* phase, ***wsimport*** is invoked to parse the API WSDL URL and generate required files (JAX-WS portable artifacts) for clients to access Odyssey APIs via web services calls. The generated files are stored on the ***OdyApiGen-Builder*** module project and copied to the ***OdyApiGen-Library*** module project. A similar action is performed by ***wsimport*** on the Odyssey WSDL integration URL. Then, the ***builder POM*** compiles and runs a custom program, during Maven’s *compile* and *install* phases respectively, to generate the classes that support marshalling/unmarshalling of XML messages for working wih the Odyssey APIs. This custom program (gov.nmcourts.api.builder.main.***Builder.java***) also modifies the Odyssey schema structure (XSD files) by replacing *include* statements with *import* statements and adding prefixes to eliminate collisions and allow a progressive building process with JAXB (details are presented in the next sections), and the generated files are stored in the target folder of the ***OdyApiGen-Library*** project.

The goal of the ***OdyApiGen-Library POM*** file is the creation of the API library JAR file that aggregates all files generated by the ***OdyApiGen-Builder POM*** file. Like ***OdyApiGen-Builder POM***, the ***OdyApiGen-Library POM*** during the Maven *clean* phase cleans up temporary files and previous JAR files, and during the Maven *install* phase not only creates the JAR file but also installs it inside the local Maven repository. To ensure a proper generation and use of the library make sure the version of the library on the ***OdyApiGen-Library POM*** –line 13, matches version of the dependency on the ***example POM*** –line 38. Notice, that JAR file is also available from the target folder of the ***OdyApiGen-Library*** module project. Thus, the JAR file can be included in another JAVA project.

The ***OdyApiGen-Example POM*** will compile and run a program for testing and/or demonstration of some of the Odyssey APIs. It includes the JAR file generated by the ***OdyApiGen-Library POM*** as a dependency. When the program runs it will present an graphical UI that is used to setup parameters that are unique to your Odyssey environment.

**IMPORTANT NOTE:** ***you should use a TEST instance of Odyssey as the example program modifies Odyssey data***. In the GUI that is displayed, you will provide a valid Odyssey case number with at least one case event that has a PDF document attached to it. If everything was done correctly, XML messages will appear on the console, and changes will be done in the (TEST!) Odyssey instance used by the program.

# Finding Conflicts in XSD Schema files

JAXB allows Java programmers to map Java classes to XSD representations, and it is able to marshal objects into XML messages, and also performs the inverse process – XML messages are stored into Java objects (unmarshalling). These tasks are accomplished by using classes created by JAXB based on Odyssey’s API XSD files. However, using JAXB requires a well defined (i.e. syntactically and semantically correct) schema (i.e. set of XSD files) that are free of conflicts.

We have found no issues using this build process when utilizing the API XSD files from Odyssey versions 2013 and 2014, but we have determined that Odyssey 2017 API XSD specifications have a few minor errors that must be corrected. Some of the errors in Odyssey 2017 can be automatically corrected by the builder software, while some require manual intervention. So, if you are using Odyssey version 2017, please see the next section for more information on some of the errors that will be automatically resolved by the builder, as well as a couple of errors you must resolve yourself for this version of Odyssey.

# Resolving Conflicts found in Odyssey release 2017 APIs (XSDs)

Passing all of Odyssey 2017 API XSD files to JAXB for generating the library is not possible without first solving the following conflicts:

* **Conflict 1** Each API is defined by a pair of XSD files where all ***Requests*** and ***Responses*** have as main elements ***Message*** and ***Result***, respectively. The first ***Request*** XSD file will declare ***Message*** as its main element and then the next ***Request*** XSD file will do the same, producing a collision; ***Message*** can be defined only once in the same namespace. Figure 3 shows how the conflict on the ***Message*** element takes place between AddAdHocHearing.xsd and AddAttorney.xsd.

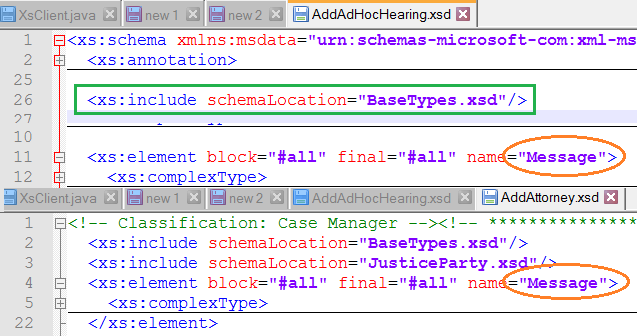


Figure - CONFLICTS BETWEEN ADDADHOCHEARING.XSD AND ADDATTORNEY.XSD

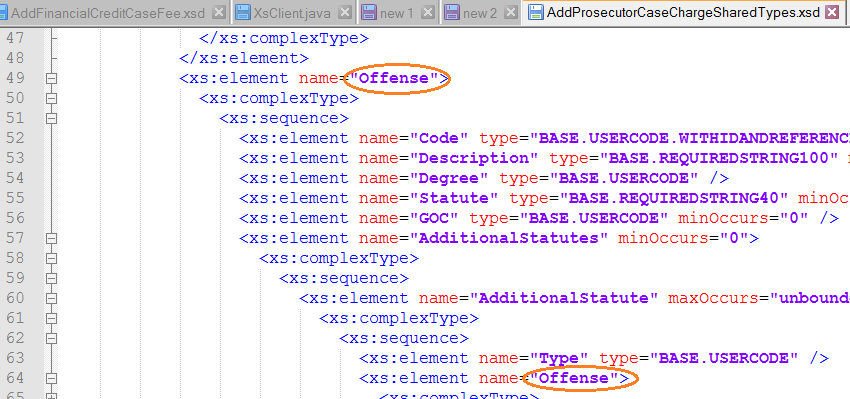
* **Conflict 2** XSD files use include statements to use definitions from other XSD files (type definitions) and all of them belong to the same or no namespace. Therefore, if several XSD files include the same file it will create a collision because those type definitions will be defined multiple times in the same or no namespace. Figure 3 illustrates the conflict when BaseTypes.xsd file is imported by AddAdHocHearing.xsd and AddAttorney.xsd files.
* **Conflict 3** JAXB is not able to create the expected classes if there are nested elements with the same name. Figure 4 shows this condition for element ***Offense***.

Figure - Nested Element Naming Conflict on AddProsecutorCaseChargeSharedTypes.xsd

* **Conflict 4** JAXB is not able to process XSD files where elements or types include white spaces in their definitions as shown in Figure 5. These problems are easy to fix by a program that parses the schemas files and deletes (i.e. trims) white spaces.

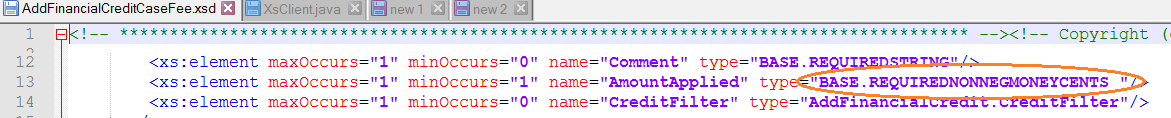


Figure - Type Definitions that Include White Spaces

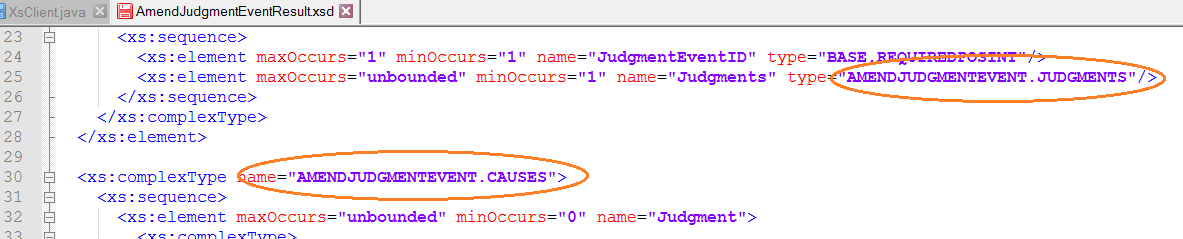
* **Conflict 5** Type definitions are missing. This type of conflict arises when there is a typo in the definition of a complex type as presented in Figure 6. Line 26 defines “Judgments” as “AMENDJUDMENTEVENT.JUDGMENTS” type. However, that type is not defined either in the API XSD file nor in the included file (BaseTypes.xsd). Line 30 should be changed to <xs:complexType name="AMENDJUDGMENTEVENT.JUDGMENTS">

Figure - Missed Type Definition on AmendJudgmentEventResult.xsd

* **Conflict 6** Include statements are missing. As presented in Figure 7, several elements were defined with types defined in BaseTypes.xsd and JusticeCharges.xsd. However, include statements were missing. For fixing the problems the following two statements should be added to the schema on line 21:

<xs:include schemaLocation="BaseTypes.xsd" />

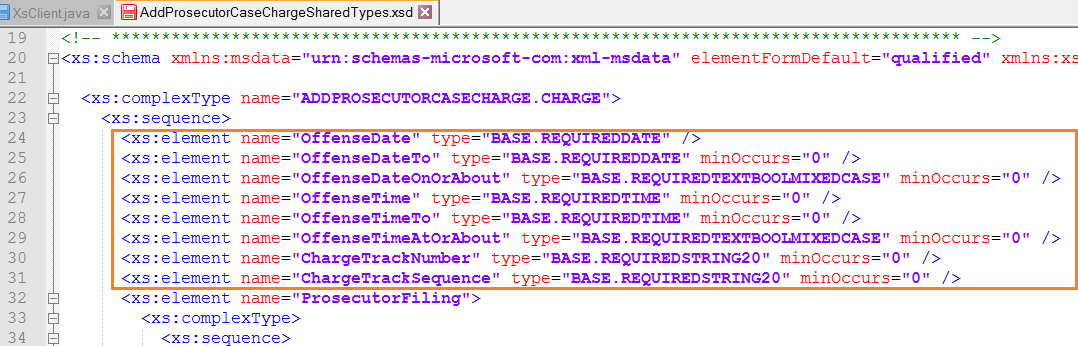
 <xs:include schemaLocation="JusticeCharges.xsd" />

Figure - Missed Include Statements on AddProsecutorCaseChargeSharedTypes.xsd

# Resolving Conflicts Automatically

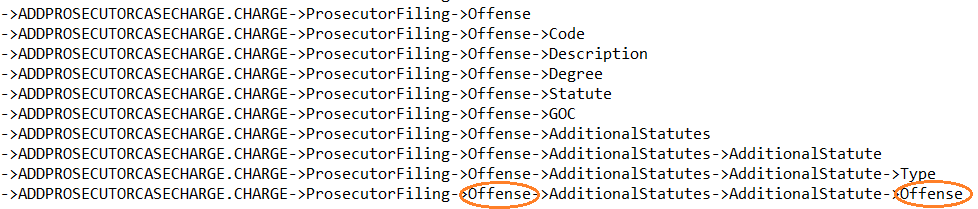
***The Odyssey API Generator project is able to automatically resolve all of Odyssey 2017’s conflicts, with the exception of Conflict 5 and Conflict 6.*** These two conflicts involve typos and/or missing data types (included in other XSD files), which will be fixed in the near future by Tyler Technologies (in some version of Odyssey 2018 or above, we are told). However, if you must use Odyssey version 2017 and cannot yet get the fixes from Tyler in 2018, you can resolve these conflicts manually as we will describe later. ***Conflict 4*** is fixed automatically, by reading all elements from XSD files and trimming white spaces. ***Conflict 3*** is fixed automatically, by parsing each schema file, after which child nodes are extracted and paths are calculated as presented in Figure 8.

Figure - Calculated Paths from AddProsecutorCaseChargeSharedTypes.xsd

For finding nested conflicts, duplicate names are detected in every path across every XSD file, and if duplicates are found, then the last one is renamed by appending a number to the name. For example, the second ***Offense*** will be renamed as ***Offense1***. This process is repeated until no more conflicts are found. The ***OdyApiGen-Builder (conflict solver)*** depicted in Figure 1 includes a method (called ***handleCollisions***) to perform this process. Conflicts *1* and *2* are also fixed automatically by the ***OdyApiGen-Builder*** and the solution is presented in the next section.

# Builder (Conflict Solver)

For solving ***conflicts 1 and 2*** (produced by having multiple elements with the same identifiers in the same or no namespace or by having schema files using include statements) the ***OdyApiGen-Builder*** program changes the schema structure from a monolithic schema in the same or no namespace to multiple sub namespace or prefixes where each XSD file will have one unique prefix.

Organizing the schema in that fashion eliminates conflicts because each API’s main elements (***Message*** and ***Result)*** reside within their own namespace. Also ***include statements*** are replaced by ***import statements*** in the definition of the schema to prevent a schema file from being regenerated again. JAXB *episodes* are used in the building process to ensure each schema file is processed once and only once.

One of the firsts task performed by the ***OdyApiGen-Builder*** is parsing XSD files to find a hierarchy between them; if a schema file includes other schema file, then the last one must be processed first. Finding dependencies is important to establish the processing order for JAXB. Figure 9 presents the dependencies for eight XSD files. BaseTypes.xsd (nodeA) should be the first episode to be built, then Justice.Event.xsd (nodeB) which will import definitions from BaseTypes.xsd, then JusticeCharges.xsd (nodeC) must be processed, followed by AddCaseEvent.xsd (nodeE), AddChargeCaseFilling.xsd (nodeF), AddAdHocHearing.xsd (nodeD), AddAdHocHearingResults.xsd (nodeG), and AddAdultConditionsSentenceComponent.xsd (nodeH). Notice, also, that as soon as nodeA is processed (or constructed) nodes D, G, and H can be generated, there is no need to wait for nodes B, C, E, or F.

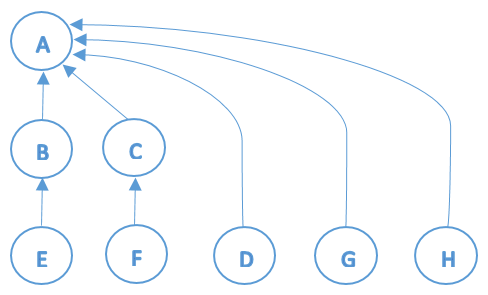
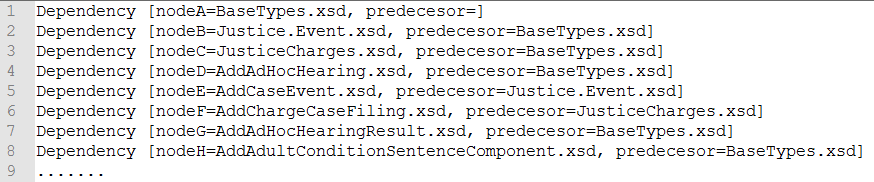


Figure - Identifying Dependencies across Schema Files

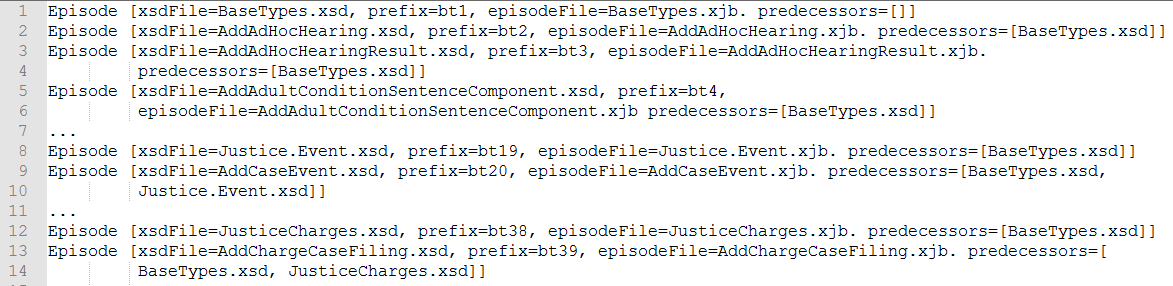
For finding schema dependencies a directed graph was constructed and paths to all nodes were calculated starting from the root node (nodeA). An object is associated with each schema file called an ***Episode*** for storing not only the predecessors (prior nodes on the path to the root node) but also prefixes as presented on Figure 10. Prefixes are needed later to identify a sub namespace for each XSD file. Notice, also, that each XSD file will have an episode (or binding file) that matches the name of the XSD file with extension XJB.

Figure - Episode Objects Associated to Schema Files

The next step for changing the schema structure is to update XSD files by including in each one a default namespace (which is defined for all non-prefixed elements) and a targetNamespace (that defines the namespace that the schema is intended to target or validate).

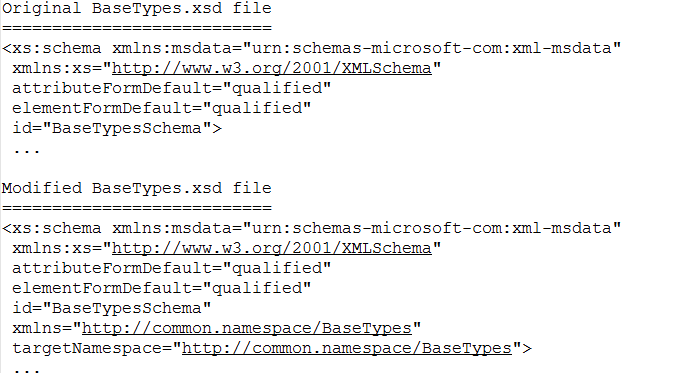


Figure - Modifications for BaseTypes.xsd

Figure 11 shows how to acomplish these tasks in root node, BaseTypes.xsd. All schema files will start with the default namespace and targetNamespace <http://common.namespace/> followed by the name of the XSD file. The XML messages will include the targetNamespace, however it will be automatically removed, at run-time, immediately before messages are sent from the endpoint to Odyssey; this run-time manipulation is performed by the hand-written Java Classes that are distributed with the shared library (within the invoker) described by Figure 1.

The rest of the schema files need more additions because they have dependencies described in their include statements; for example, Figure 12 shows modifications needed for AddCaseEvent.xsd. Besides the default namespace and the targetNamespace two more namespaces with a prefix were included (bt1 and bt19) for importing elements coming from AddCaseEvent’s dependencies (BaseTypes.xsd and Justice.Event.xsd). Include statements were replaced with import statements and their namespaces were defined.

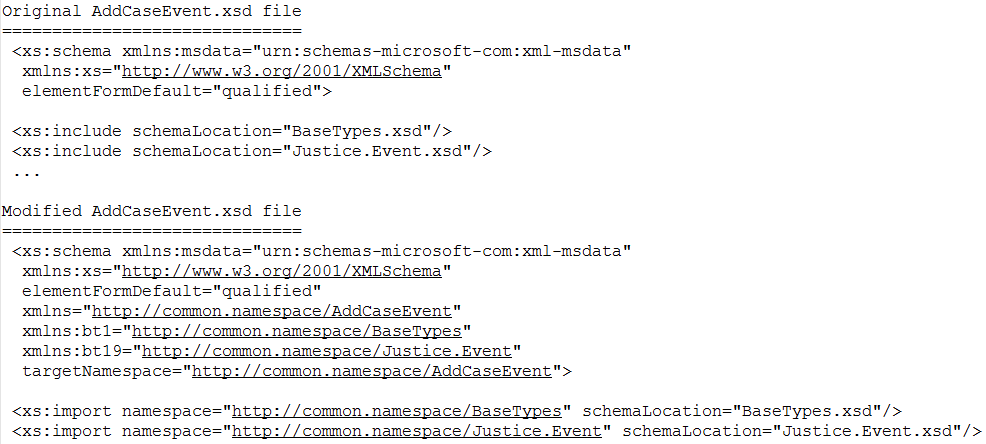


Figure - Modifications for AddCaseEvent.xsd

Notice also BaseTypes.xsd and Justice.xsd prefixes were added to elements that belong to a dependency. For instance, ***BASE.REQUIRESPOSINT*** was replaced with ***bt1:BASE.REQUIRESPOSINT*** as presented in Figure 13:

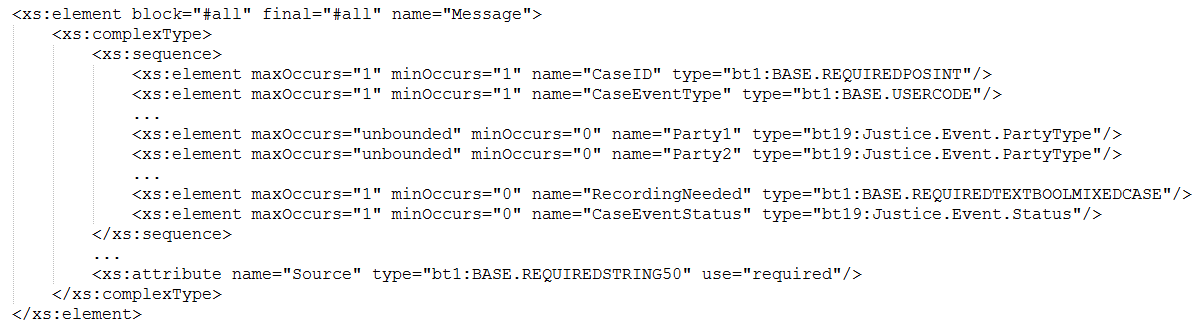


Figure - Adding Prefixes

Keep in mind all imported types in Figure 12 must be validated against their dependency files. Meaning all types in AddCaseEvent.xsd (the ones replaced with bt1 as prefix) must exist in the BaseTypes.xsd, and this is the reason why the ***OdyApiGen-Builder*** project is parsing dependency files. Otherwise, the program will generate an error if this step had been absent.

This conversion from include to import files is repeated in each schema file according to the order determined in the previous steps. In addition to the modified XSD files that are created during the build/conflict resolution, there are also binding files (one per API) that are produced; these are used to generate xjc episodes. When the conversion finished new XSD files and binding files will be ready to be processed by JABX to create the binding classes.

# Hand Written Classes (invoker)

Figure 1, step 2 shows how some hand-written classes were added to the API library because the schema structure was changed from what Odyssey is expecting. If the API library is used to generate ***Request*** messages (i.e. for ***FindCaseByCaseNumber API***) a namespace will be included on the XML message as presented on Figure 14, and it will produce an error the moment it hits Odyssey, since Odyssey isn’t expecting such a namespace.

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<Message MessageType="FindCaseByCaseNumber" NodeID="1" ReferenceNumber="TEST\_REFNUM\_1234" UserID="4847" Source="TEST\_SOURCE" **xmlns="http://common.namespace/FindCaseByCaseNumber">**

Figure - FindCaseByCaseNumber Created by the API Library

In order to fix this problem an intermediate piece of code was added to the library called invoker, which not only will marshal and un-marshal API messages but also it will get rid of namespaces created by the use of import statements during the compilation process of the API library. Figure 15 presents the correct message generated by the invoker for the FindCaseByCaseNumber API.

<?xml version="1.0" ?>

<Message MessageType="FindCaseByCaseNumber" NodeID="1" ReferenceNumber="TEST\_REFNUM\_1234" UserID="4847" Source="TEST\_SOURCE">

Figure - FindCaseByCaseNumber Message Generated by the Invoker

# Final Notes

The ***OdyApiGen-Example*** project includes a file called ExampleAPIs.java that provides a series of examples to get familiar with the use of the API library.