OPC UA

SDK for Java

Code Generator Manual

Version 4.0.0

Table of Contents

1. Introduction

The Prosys OPC UA SDK for Java Code Generator (later just 'Codegen') can be used to reduce manual workload when using custom information models in the Prosys OPC UA SDK for Java. It creates Java classes based on given information models provided in the OPC UA NodeSet2 XML format.

2. About Information Models

Codegen will work with most information models. If you encounter a model that does not generate, let us know.

2.1. Obtaining Information Models

Codegen requires OPC UA information models in the NodeSet2 format for the generation procedure. The NodeSet2 format stores an information model in XML file(s) conforming to the Information Model XML Schema of the OPC UA specification (v1.04, Part 6, Annex B).

A popular software for creating and managing information models is the OPC UA Modeler that also enables exporting the information model in the NodeSet2 format. The exported models can then be used as the input for Codegen.

If your model depends on other information models, you will need to obtain and generate those as well.

2.2. Known Issues

Codegen is currently supporting OPC UA Specification version 1.04 or earlier. Note that version 1.04 is continuing to receive amendments. A model build on top of those amendments might require updates to the Codegen or the SDK in order to work.

Possible name clashes related to method names

- Codegen will generate a setter and getter for each UA Variable of an UA Object, e.g. `getVar()` and `setVar(…)`. If you define an UA Method with the same name, e.g. `getVar` or `setVar`, and the same argument type inside the same UA Object, then you will get a name clash. Additionally same kind of conflict can happen if a model defines names used by the SDK base API which the generated classes extend.
- The generated Java methods do not include the UA namespace in any way. This means that if an UA type defines multiple components with the same BrowseNames that only differ by their NamespaceIndices, then it will result in a name clash.

In both cases currently either the model has to be changed, or one of the components be excluded (starting from 3.1.8) from generation. In the case where there is a conflict between an UA Method and a component, we recommend excluding the component, as excluding a Method will also remove it’s handling logic. See Excluding sub-nodes under a Type node from generation for more information.

Multiple types having same name within the same Namespace

The specification is not exactly clear is this allowed, but anyway it is technically possible to create such NodeSet file that it contains multiple types that would output as same fully-qualified Java class.

Codegen as of 3.1.8 will halt the generation if such model is detected. As a workaround you can add/edit the SymbolicNames to the model manually to differentiate the types, but fixing the names to be unique is preferred.

Missing SymbolicNames when BrowseNames are not suitable names to be used in generated code

The specification makes the option to define a SymbolicName in the model when the normal (Browse)Name is not usable as a name in generated code. Some models are missing these. Codegen as of 3.1.8 will try to guess a suitable one if none is found, but fixing the model is preferred.

3. Using Codegen

Codegen can be currently run in two ways:

1. From command line. See chapter Running from Command Line.
2. With Maven. See chapter Maven Integration.

In both cases, the generated code will compile and work in Java 6 or later.

3.1. Running from Command Line

For running Codegen from command line you will need the `java` command to be available from PATH. Codegen requires (exactly) Java 8 runtime to operate (but the generated code can be compiled in Java 6 or later).

The parameters of the code generation can be modified through a configuration XML file. Please see the example configuration files in the 'configexamples' folder. Also see the Configuration chapter for more general information on configuring the code generation procedure.

Codegen is designed to be launched via the supplied startup scripts found in the 'codegen/bin' directory: 'codegen.bat' for Windows and 'codegen.sh' for other operating systems.

The following arguments can be passed to the scripts:

Table 1. Codegen command line arguments

Argument	Info
-c <path>	Path to the configuration XML file WARNING: This is **NOT** the Information Model XML, but the configuration for the Codegen itself. See Configuration chapter for more information.
-v	Output debug log
-h	Display help

On Windows, run Codegen with:

```
codegen.bat -c path_to_configuration_file
```

For example, on Windows PowerShell running the Codegen for generating the SampleTypes.xml model (which is in the `commandline/models` folder) the call could look like (in working directory `commandline`):

```
.\bin\codegen.bat -c .\configexamples\sampletypesconf.xml
```

On other operating systems, run Codegen with:

```
./codegen.sh -c path_to_configuration_file
```

3.2. Maven Integration

If you use Apache Maven as your build tool, you can use the included maven plugin for Codegen.

> This tutorial does not cover Maven usage in general. Please, visit Apache Maven website for more information.

The Maven plugin for Codegen is provided in the `codegen/maven-integration/maven-install-helper` Maven project. It can be installed to your `.m2` local repository by running `mvn install` in the 'maven-install-helper' folder (see the pom.xml in that folder for more information). Then you can look at the 'maven-integration-sample' project and run `mvn generate-sources` for it.

> The generated classes are not added to the build path automatically. If needed, you can use the Build Helper Maven Plugin to achieve that.

The parameters of code generation can be modified by configuring the Codegen plugin in the associated 'pom.xml' file. See the Configuration chapter for more general information on configuring the code generation procedure.

4. Configuration

This section explains some of Codegen's configuration and usage concepts.

4.1. Mapping NamespaceUris to Java Packages

OPC UA has the concept of namespaces where each node is in some namespace. Namespaces organize OPC UA nodes according to naming authorities and allow different namespaces to contain same type names. Different namespaces are distinguished by their NamespaceUris. In an OPC UA server, each namespace is also identified by individual indices in the NamespaceArray of the server. They are used instead of the NamespaceUris to allow for a more compact data transfer. However, because the indices are defined during server runtime, they cannot be utilized in the code generation process and instead the NamespaceUris are used to identify namespaces.

Java programming language has the concept of packages which are used to organize classes. Therefore, in Codegen, each OPC UA namespace is mapped to a unique Java package.

4.1.1. Namespace Prefix

Additionally, a prefix parameter can be defined for each generated namespace. The prefix is used to prepend the generated class names of `Ids`, `ClientInformationModel`, `ServerInformationModel`, `DataTypeDictionaryHelper` and `Serializers`. This helps with identifying the different implementations of these classes when you are registering them in your client or server application. If you don't use the prefix, you can still classify the classes with their fully qualified class names (including the Java package name).

4.2. Generation Targets

Codegen generates Java classes from types (ObjectTypes, VariableTypes, ReferenceTypes and DataTypes) defined in OPC UA information models.

If your models include instances (Objects and Variables), Codegen ignores these, since it can only generate Java classes for UA types. However, when you load the model to your server at startup, these instances will be generated by the SDK. See the Server Tutorial for more information.

Codegen can create the following Java interfaces and classes for each OPC UA ObjectType and VariableType that is defined in the models (using the UA AnalogItemType as an example):

- Common *interface* that has the same name as the UA type: `AnalogItemType`
- Base *class* for the client side: `AnalogItemTypeImplBase`
- Actual *class* for the client side: `AnalogItemTypeImpl`
- Base *class* for the server side: `AnalogItemTypeNodeBase`
- Actual *class* for the server side: `AnalogItemTypeNode`

The base classes should not be modified after generation. The actual classes can be modified, but note that Codegen will override the changes when you regenerate. So you should copy the actual classes to version control to protect them from unwanted changes..

For each UA Method in an ObjectType, the following Java files are created (using the Resend Method of the ServerType as an example):

- Parameter object *class* for each Method that has more than one output argument. It is used as the return value for the method. It is generated as a nested class of the Common interface.
- Helper *interface* for providing an external method implementation: ServerTypeResendMethod.

Additionally, the following classes are generated per model:

- Structure *class* for each UA Structure DataType
- An *enum* for each UA Enumeration DataType
- Serializers *class*, which will contain one Serializer per Structure
- DataTypeDictionaryHelper *class*, which is used internally by the SDK to handle custom structures
- ClientInformationModel *class* that is used to register the generated classes on the client side
- ServerInformationModel *class* that is used to register the generated classes on the server side
- Ids *class* which contains the identifiers (as ExpandedNodeId) for each generated type and their subcomponents

If a namespace is mapped with a *prefix*, it will be used to prepend the Ids and InformationModel classes. For example, for a prefix Test the output would be TestIds, TestClientInformationModel and ServerInformationModel.

For Optional Structures and Unions, both of which are specialized forms of Structures, introduced in OPC UA version 1.03, the EncodingMask and SwitchValue are handled automatically by the generated code (in Serializers). Please see OPC UA Specification (v.1.03) Part 6 sections 5.2.6 and 5.2.7 for more information.

4.2.1. Target Configuration

Codegen enables you to configure which targets you wish to generate by editing the configuration files. For this purpose, the following target names are defined:

Table 2. Possible generation targets

Target	Description
common	Generate Java interfaces for each type in the model plus Ids, Serializers, Structures, Enumerations and helper classes. *These are not designed for editing.*
client_base	Generate client-side base class for each type. *Not designed for editing.*
client_impl	Generate client-side actual class for each type (extends base classes, **can be edited**)

Target	Description
server_base	Generate server-side base class for each type *Not designed for editing.*
server_impl	Generate server-side actual class for each type (extends the base classes, **can be edited**)
server_model	**Special** Copy the used information model file to the server-side package
client_model_provider	**Special** Writes META-INF/services descriptor files com.prosysopc.ua.client.ClientCodegenModelProvider com.prosysopc.ua.server.ServerCodegenModelProvider. Used for automatic detection of generated models for the SDK. See Automatic Discovery of Generated Models for more information. [NOTE] ==== As there can be only one of such files per output folder, Codegen appends to that file (unless it already had the provider that would be appended). ====
server_model_provider	

Table 3. Additional generation targets that are combinations of the previous targets

Target	Description
client_all	Combination of client_base, client_impl.
server_all	Combination of server_base, server_impl.
base_all	Combination of common, client_base, server_base.
impl_all	Combination of client_impl, server_impl.
all_code	Combination of everything except server_model, i.e. all .java output.
all_resources	Combination of client_model_provider, server_model_provider and server_model, i.e. all other than .java output.
all	Combination of every target

If you generate the base and actual (i.e. "impl") targets in different code generation configurations, remember that both are always required.

4.3. Method Implementations

Instead of defining method implementations in the actual classes, the SDK also supports implementing them out of the actual class. Therefore, the Codegen also generates a method interface which can be implemented and plugged into the actual class. Please see the generated classes to learn how this can be done.

4.4. Object Initializations

Additionally, an initializer can be defined for each type, for example to provide initial values. Please see the generated classes to learn how this can be done.

4.5. Configuration Parameters

Codegen can be configured with a few configuration parameters. In the command line version of Codegen, the changes need to be implemented in a configuration XML file. In the Maven plugin version, the parameters are defined in the POM XML file. Please also see the provided examples for more complete examples of the configuration process. The examples for the command line version are in the 'configexamples' folder and the 'maven-integration-sample' Maven project provides examples for configuring the Maven integration.

4.5.1. Model Directories

The modelSources parameters define the directories (or files directly) where Codegen is searching for information models that are used in the code generation. The files must conform to the OPC UA NodeSet2 XML format.

In the command line version, the models are defined in the following way:

```
<modelSources>
   <modelSource>${app.home}/models</modelSource>
</modelSources>
```

The Maven plugin is configured the same way (but you would use e.g. `${project.basedir}/models` as the setting value.

4.5.2. Namespace Mapping Parameters

The namespace mapping parameters define how the OPC UA namespaces that are defined in the information models are mapped to their respective Java packages. See chapter Mapping NamespaceUris to Java Packages for an explanation of the concept. Each mapping consists of the following parameters:

Parameter	Description
uri	The NamespaceUri associated with the information model used as the input of the code generation
packageName	The name of the Java package where the Java code output for the information model is placed
prefix	**Optional** The prefix used in the name of the `ServerInformationModel` and `Ids` classes, e.g., prefix value 'Di' outputs the class `DiServerInformationModel`

In both command line version and the Maven plugin, the namespace mapping parameters are defined in the following way:

```
<namespaceMappings>
   <namespaceMapping>
      <uri>http://ua.prosysopc.com/SampleTypes</uri>
      <packageName>example.packagename</packageName>
      <prefix>Sample</prefix>
   </namespaceMapping>
</namespaceMappings>
```

4.5.3. Generation Target Parameters

The concept of the generation target is explained in more detail in the chapter Generation Targets.

Parameter	Description
targets	Defines what is the output of the code generation, a comma separated list of Generation Targets
uri	The NamespaceUri associated with the information model used as the input of the code generation
outputs	Definition of output folders (see below examples)

In the command line version, the generation target parameters are defined in the following way:

```
<generates>
   <generate>
      <targets>all</targets>
      <uris>
         <uri>http://ua.prosysopc.com/SampleTypes</uri>
      </uris>
      <outputs>
         <code>${app.home}/sampletypes/output_code</code>
         <resources>${app.home}/sampletypes/output_resources</resources>
      <outputs>
   </generate>
<generates>
```

In the Maven plugin, the configuration is the same, but typically you would define the output locations as properties (e.g. `${codegen.output.code}` and `${codegen.output.resources}`).

4.5.4. Excluding sub-nodes under a Type node from generation

Some models might result in generated code that conflicts the base API the SDK provides with UaNodes or name clashes because of one or more models, and thus might not compile. Currently the only workaround (outside of editing the model itself) is to exclude those nodes when generating get/set/getXXXNode methods for the type. An excluded sub-node behaves in the context of the generation as-if it were never present in the Type under which it is.

In the both versions they can be excluded:

```
<excludes>
   <instanceDeclarations>
      <instanceDeclaration>namespace_uri:name_part_of_BrowseName</instanceDeclaration>
   </instanceDeclarations>
</excludes>
```

For example, in order to exclude the PowerInput subnode (that is used in ValveObjectType in the SampleTypes model), you would use the following configuration:

```
<excludes>
   <instanceDeclarations>
      <instanceDeclaration>http://ua.prosysopc.com/SampleTypes:PowerInput</instanceDeclaration>
   </instanceDeclarations>
</excludes>
```

Excluding a component does not remove it from the AddressSpace of the server, it just means that the generated accessor for it wont be available. The equivalent operation can be done "manually" by getting the component node (UaNode.getComponent) and reading/writing values to that directly. Excluded Methods do not create the handling logic, and must be done via a listener (see MyMethodManagerListener in the samples).

Currently the exclude excludes it from all types, as the main use case is to avoid name clashes with the SDK core API.

4.6. Automatic Discovery of Generated Models

Normally the generated InformationModel classes must be registered to the SDK (via UaClient.registerModel() or UaServer.registerModel) in order to make the SDK aware of the generated classes (please see the Client or Server tutorial for more information). If you use the client_model_provider (for client side) or server_model_provider (for server-side) Generation Targets, you can enable the SDK to discover them automatically. Both targets outputs a so called "service provider file" named com.prosysopc.ua.client.ClientCodegenModelProvider or com.prosysopc.ua.server.ServerCodegenModelProvider under META-INF/services in the configured resources output. These files are then used by the SDK via standard Java ServiceLoader mechanism to instantiate and load all generated InformationModel classes.

The generated files (com.prosysopc.ua.client.ClientCodegenModelProvider or com.prosysopc.ua.server.ServerCodegenModelProvider under META-INF/services) must be visible in the java classpath for the SDK in order for this system to work. This might not be the case if you are using e.g. OSGi or some other environment which uses custom ClassLoaders, or they might need extra configuration.

If you try to use the mechanism yourself, note that the Standard and GDS models that are bundled inside the SDK are not in the list which Java ServiceLoader would return. SDK handles their registration explicitly.

4.7. Example configuration files

This is a basic example of a configuration file for the command line version. Note that the same file is available from the codegen/commandline/configurationexamples folder

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

  <!-- Mapping of UA Namespace -> Java package -->
  <namespaceMappings>
    <!-- Add mapping to your model(s) namespace(s) here -->
    <!-- Each mapping must have unique java package name -->
    <namespaceMapping>
      <uri>http://ua.prosysopc.com/SampleTypes</uri>
      <packageName>example.packagename</packageName>
    </namespaceMapping>
  </namespaceMappings>

  <!-- Defines where models are loaded from. Use absolute paths, 'app.home'
    is set by the launcher scripts to parent directory of the launch scripts
    and can be used to make relative paths. Entries can be files or folders.
    All xml files from the folder are assumed to be model files -->
  <modelSources>
    <modelSource>${app.home}/models</modelSource>
  </modelSources>

  <!-- These define which namespace should be generated and which targets
    to use. See the Codegen Manual about the possible Generation Targets. -->
  <generates>
    <!-- Multiple generate blocks are possible -->
    <generate>
      <targets>all</targets>
      <uris>
        <!-- Multiple uris possible -->
        <uri>http://ua.prosysopc.com/SampleTypes</uri>
      </uris>
```

```
        <outputs>
          <code>${app.home}/sampletypes/output_code</code>
          <resources>${app.home}/sampletypes/output_resources</resources>
        </outputs>
      </generate>
    </generates>

    <enhancements>
      <!-- Optionally define a header to generate for each file. Every line is
        prepended with '//'. -->
      <fileHeader>
        <line>Generated from SampleTypes</line>
        <line>by Prosys OPC UA Java SDK Codegen</line>
      </fileHeader>
    </enhancements>
  </codegenConfiguration>
```

Here is another example with the maven plugin version. Please refer to the full pom.xml in the codegen/maven-integration/maven-integration-sample. Please note that you must have taken steps explained in Maven Integration part before the plugin works.

```xml
<plugin>
   <groupId>com.prosysopc.ua.codegen</groupId>
   <artifactId>prosys-opc-ua-java-sdk-codegen-maven-plugin</artifactId>
   <version>${codegen.plugin.version}</version>
   <executions>
      <execution>
         <id>run-codegen</id>
         <phase>generate-sources</phase>
         <goals>
            <goal>generate</goal>
         </goals>
         <configuration>
            <modelSources>
               <modelSource>${project.basedir}/models</modelSource>
            </modelSources>
            <generates>
               <generate>
                  <targets>all</targets>
                  <uris>
                     <uri>http://ua.prosysopc.com/SampleTypes</uri>
                  </uris>
                  <outputs>
                     <code>${codegen.output.code}</code>
                     <resources>${codegen.output.resources}</resources>
                  </outputs>
               </generate>
            </generates>
            <namespaceMappings>
               <namespaceMapping>
                  <uri>http://ua.prosysopc.com/SampleTypes</uri>
                  <packageName>example.packagename</packageName>
               </namespaceMapping>
            </namespaceMappings>
         </configuration>
      </execution>
   </executions>
</plugin>
```

5. Using Generated Classes in Applications

5.1. Client-side Applications

For instructions on using the generated classes in client-side applications developed with the Prosys OPC UA SDK for Java, please refer to the Prosys OPC UA SDK for Java Client Tutorial in the 'tutorial' folder of the distribution package.

5.2. Server-side Applications

For instructions on using the generated classes in server-side applications developed with the Prosys OPC UA SDK for Java, please refer to the Prosys OPC UA SDK for Java Server Tutorial in the 'tutorial' folder of the distribution package.