MICRO FOCUS® | CORBA® An OMG Standard

Discover the Future of CORBA

Micro Focus® | CORBA® Add-on for REST 1.1.0

Installation and User Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

2020-12-17

Contents

Preface

This Guide describes the Micro Focus® | CORBA® Add-on for REST. It describes how to install and set up the product.

In This Guide

This manual contains the following chapters:

- "Introduction" describes some of the concepts of the CORBA Add-on for REST.
- "Installing the CORBA Add-on for REST" gives installation instructions.
- "Getting Started" tells you how to build and run the Typetest demonstration program.
- "IDL-RS Annotations" describes an extension of the IDL annotations concept for a REST API.
- "IDL Type Serialization" tells you how to serialize IDL types into JSON and XML data formats:
- "IDL Request and Response Wrapping" describes using wrappers to create single request and response objects.
- "Advanced HTTP Integration" describes the support provided for HTTP Cross Origin Resource Sharing and for HTTP Caching.
- "Open API Support" describes how CORBA Add-on for REST uses the OpenAPI description format for REST APIs to generate and use an OpenAPI model.
- "System Exceptions" describes system exception mapping.
- "Configuration" describes configuration properties and how they can be used and overridden.
- "idl2rest Options" describes command line flags that the **idl2rest** tool will accept.
- "Open API Block Comments Reference" describes the Block Comments facility used in the OpenAPI model.
- "Connector Options" describes the command line flags used by the REST Connector.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.

- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:

Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact Information

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

Introduction

This chapter introduces the Micro Focus® | CORBA® Add-on for REST (CORBA Add-on for REST).

What is the CORBA Add-on for REST?

The CORBA Add-on for REST provides a standard and interoperable mechanism to enable CORBA objects to be exposed as REST services.

With the CORBA Add-on for REST, pure REST client applications can use these exposed CORBA services transparently, without having any knowledge or awareness that these services are ultimately implemented by CORBA objects.

The advantages offered by the CORBA Add-on for REST include the ability to:

- Selectively annotate the IDL definitions corresponding to the CORBA objects to be exposed to REST client applications, in order to incrementally extend the reach and use of existing CORBA assets.
- Provide a standard mechanism to decorate IDL constructs with IDL-RS annotations to clearly and unambiguously define REST representations of CORBA services, which strive to comply with the Representational State Transfer (REST) architectural style.
- Enable REST client applications to utilize readily available REST software stack (such as an HTTP library and JSON or XML library) to access the CORBA services exposed to REST, without requiring CORBA run-time technology or tooling on the client side.
- Enable REST developers to build new REST client applications that interact with a REST API façade, defined by IDL-RS annotations, to CORBA objects, defined by IDL. This approach allows to leverage existing CORBA assets without requiring CORBA expertise for the client-side developers, who only need to use tools and technology that REST client developers are already familiar with.
- Leverage the established approaches that REST over HTTP client applications already employ to achieve load-balancing, NAT traversal, and firewall traversal.
- Enable REST developers to consume a generated OpenAPI model representing the REST API defined from the annotated IDL file. This OpenAPI model can then be readily consumed with various freely available OpenAPI tools including code generators and REST API documentation tools. See the “Open API Support” chapter for further information.

Components

The CORBA Add-on for REST contains the following components:

- An IDL compiler tool (**idl2rest**) that can translate existing CORBA IDL along with IDL-RS Annotations, in order to generate a custom REST-to-CORBA mapping.
- REST run-time and support libraries that can be used together with the **idl2rest** generated code for the REST Connector.
- Support for the following Micro Focus CORBA ORBs:
  - Orbix 6.3.11
  - VisiBroker 8.5.6
- Product HotFixes for the ORBs mentioned above, that enable their respective IDL compilers to correctly parse IDL-RS annotations. This simplifies the development workflow for the CORBA products listed above, by allowing existing IDL files to be annotated in place. This means that only a single copy of the IDL files needs to be maintained.

Prerequisites

Each component has its own requirements.

- **Installer:** The installer requires JDK 1.7 (or later) to be installed. The installer is available for both Windows and Linux operating systems.
- **ORB Installation:** The installer requires an existing CORBA installation to upgrade.

Licensing

All CORBA Add-on for REST licenses are locked to the system on which they are applied and activated. You cannot copy these licenses to another system, and they cannot be accessed from a CORBA Add-on for REST product running on another system. If you reinstall the software on the same system, you will need to reactivate the license.

CORBA Add-on for REST node-locked licenses must be registered and activated before using the product. To apply the licenses you will need the following:

- **The Serial Number and Key, or the activation file for each license.** These will arrive in an email from Micro Focus.
- **A username and password.** This is needed for product registration. Note that the main **bdn.borland.com** address is no longer in use. However:
  - If you have an existing Borland Developer Network account, you can use the credentials for that account.
  - If you have a Borland Developer Network login, but have forgotten your password, go to https://my.embarcadero.com and select the Change/Forgot Password link.
  - You can alternatively create an account using the CORBA Add-on for REST Product Registration Wizard, during the registration/activation process described in "Licensing the Product". Make sure that you record the user name and password that you create.

If you are applying a license to a remote system, you must have Telnet access to that system and some way to copy (that is, FTP) the activation file to that system.

Using curl

Some of the examples included in this document use curl commands to send messages to URLs. Only one example of a curl command, for UNIX installations, is shown in each case.

The curl command is not a standard part of Windows installations. Support for curl on Windows is available through third-party products, and the exact command options used may vary between implementations. The commands given in this manual should be regarded as examples, and may not be the exact commands that your installation will require.

For more information about curl, see for example https://curl.haxx.se/.

Installing the CORBA Add-on for REST

This chapter gives installation information for the Micro Focus® | CORBA® Add-on for REST.

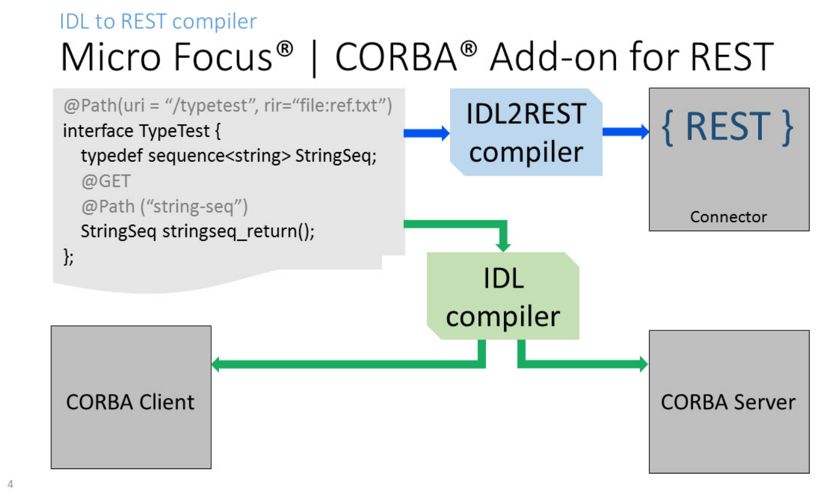
The CORBA Add-on for REST supports the following CORBA products:

- Orbix 6.3.11
- VisiBroker 8.5.6

The CORBA Add-on for REST product includes an **idl2rest** compiler which takes as its input an IDL file marked with IDL-RS annotations. The compiler uses this input to generate a custom REST connector component.

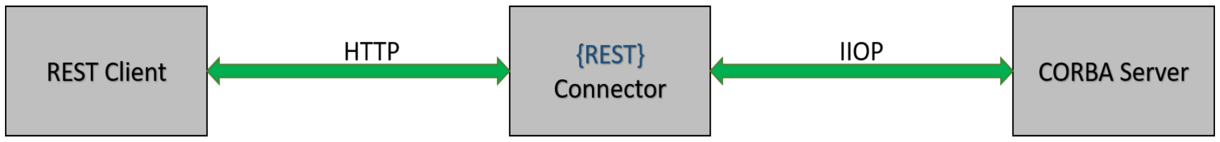
In addition, installing the CORBA Add-on for REST product updates the existing IDL compilers, enabling them to generate CORBA code in the same way as before, identifying and ignoring the IDL-RS annotations.

The diagram below demonstrates how the product can be used to annotate (using the new IDL-RS annotations) an existing CORBA IDL file, and how by using the **idl2rest** tool you can generate a custom component called the **Rest connector** that can be used to allow REST clients to communicate with CORBA servers.



```
@Path(uri = "/typetest", rir="file:ref.txt")
interface TypeTest {
   typedef sequence<string> StringSeq;
   @GET
   @Path ("string-seq")
   StringSeq stringseq_return();
};
```

The diagram below demonstrates how a typical deployment of the CORBA Add-on for REST may appear. At the left of the diagram a REST client is communicating via HTTP with a **Rest Connector** component or server; it is this component that will then act as a CORBA client to the backend CORBA server.



Installation of the CORBA Add-on for REST

In order to deploy the CORBA Add-on for REST product solution for your ORB installation, run the installer and during the installation choose the location of your existing ORB installation (VisiBroker 8.5.6 or Orbix 6.3.11).

CORBA Add-on for REST installation footprint

The content of the installer will be installed into a sub folder called `REST` within the ORB installation location:

`<`*`ORB_install`*`>/REST/bin/`	Directory containing the **idl2rest** tool, and some other scripts.
`<`*`ORB_install`*`>/REST/lib/`	Directory containing the run-time components of the product.
`<`*`ORB_install`*`>/REST/demos/`	
`<`*`ORB_install`*`>/REST/doc/license_agreement.txt`	
`<`*`ORB_install`*`>/REST/doc/notices.txt`	
`<`*`ORB_install`*`>/REST/uninstall/`	
`<`*`ORB_install`*`>/REST/etc/`	
`<`*`ORB_install`*`>/REST/license/`	
`<`*`ORB_install`*`>/REST/var/`	

Prerequisites

What you need for this installation:

- The CORBA Add-on for REST installer
- A license for the **idl2rest** compiler
- An existing ORB installation, which will need to be upgraded to work with REST. The currently supported ORB installations are:
  - Orbix 6.3.11 or higher
  - VisiBroker 8.5.6 or higher
- The CORBA Add-on for REST HotFixes for your existing ORB client and server installation machines (to be downloaded from Micro Focus Support).

  ORB HotFixes must match the platforms (operating system, compiler, bitness) that your ORB installations are deployed on.

Installation Instructions

In order to install the CORBA Add-on for REST, download the installer into a temporary directory (for example, `\temp` on Windows, or `/tmp` on Linux).

You can then either install using the GUI, or choose silent installation (see "Silent Installation").

Installing with the GUI

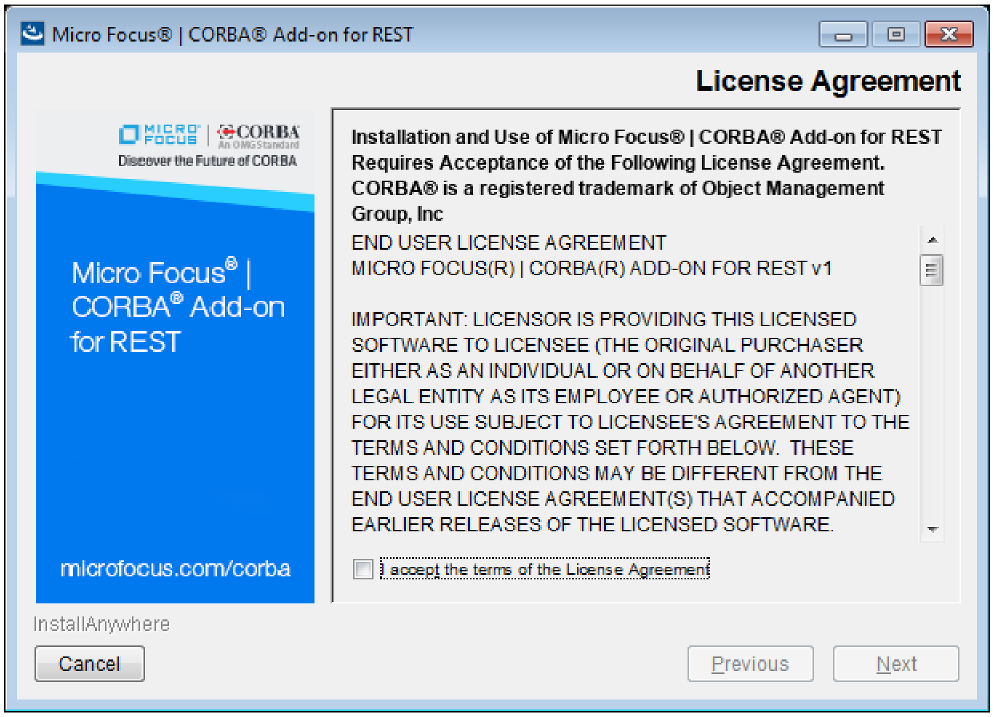
To install via the GUI, download the installer as described above. Then:

1 Run the installer to launch InstallAnywhere.

On Windows, `mf_rest_corba_addon_`*`<version>`*`_win.exe`

On Linux, `mf_rest_corba_addon_`*`<version>`*`_unix.bin`

2 The installer will run through a series of screens. The **License Agreement** screen is the first to display.


Micro Focus® | CORBA® Add-on for REST
License Agreement
MICRO FOCUS | CORBA An OMG Standard
Discover the Future of CORBA
Micro Focus® | CORBA® Add-on for REST
microfocus.com/corba
InstallAnywhere
Cancel
Previous
Next

Installation and Use of Micro Focus® | CORBA® Add-on for REST Requires Acceptance of the Following License Agreement. CORBA® is a registered trademark of Object Management Group, Inc

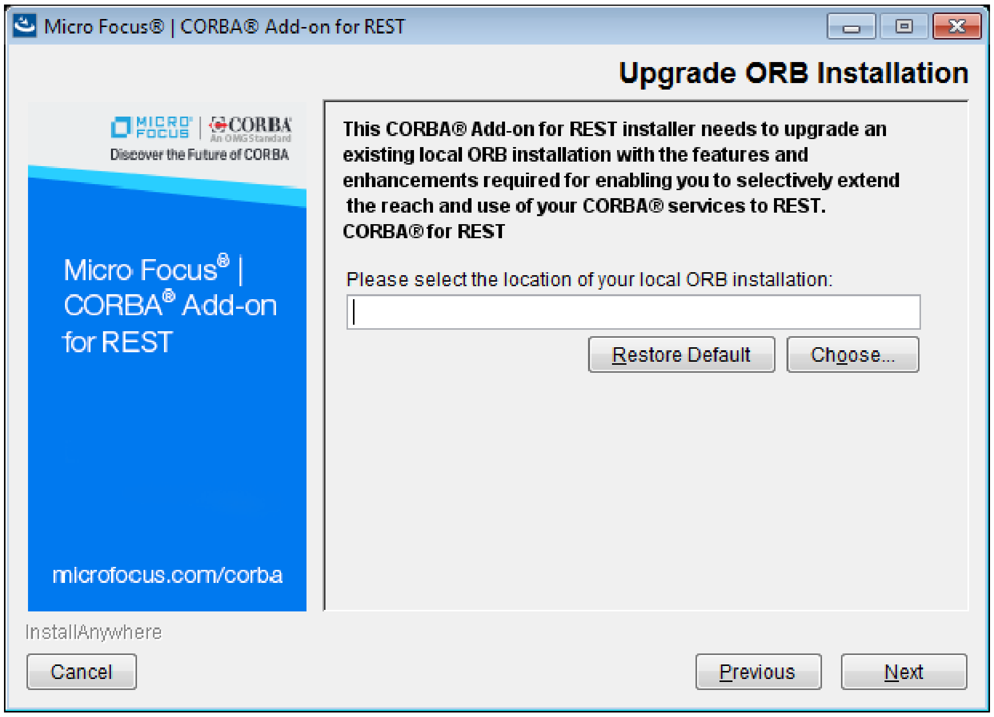
END USER LICENSE AGREEMENT
MICRO FOCUS(R) | CORBA(R) ADD-ON FOR REST v1

IMPORTANT: LICENSOR IS PROVIDING THIS LICENSED SOFTWARE TO LICENSEE (THE ORIGINAL PURCHASER EITHER AS AN INDIVIDUAL OR ON BEHALF OF ANOTHER LEGAL ENTITY AS ITS EMPLOYEE OR AUTHORIZED AGENT) FOR ITS USE SUBJECT TO LICENSEE'S AGREEMENT TO THE TERMS AND CONDITIONS SET FORTH BELOW. THESE TERMS AND CONDITIONS MAY BE DIFFERENT FROM THE END USER LICENSE AGREEMENT(S) THAT ACCOMPANIED EARLIER RELEASES OF THE LICENSED SOFTWARE.

☐ I accept the terms of the License Agreement

Read and agree the terms of the license agreement. Check **I accept the terms of the License Agreement** and click **Next**. If you do not accept the license, you cannot proceed further.

3 Choose the location of the ORB installation that will be upgraded. Either accept the default offered or click **Choose** to browse to the correct location.

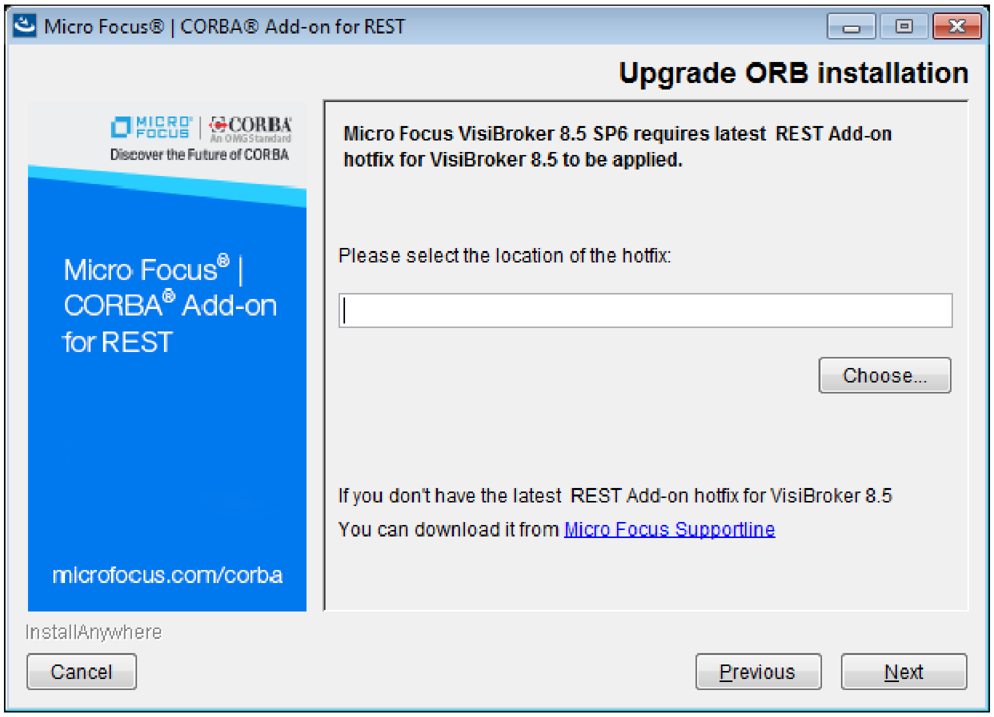

Micro Focus® | CORBA® Add-on for REST

Upgrade ORB Installation

This CORBA® Add-on for REST installer needs to upgrade an existing local ORB installation with the features and enhancements required for enabling you to selectively extend the reach and use of your CORBA® services to REST. CORBA® for REST

Please select the location of your local ORB installation:

4 The installer then asks for the location of the CORBA Add-on for REST HotFix. The text shown on the screen depends on the ORB that is being upgraded. The example illustrated below shows a VisiBroker installation; if you are upgrading an Orbix 6 ORB, the text would change accordingly.


Micro Focus® | CORBA® Add-on for REST

Upgrade ORB installation

Micro Focus VisiBroker 8.5 SP6 requires latest REST Add-on hotfix for VisiBroker 8.5 to be applied.

Please select the location of the hotfix:

If you don't have the latest REST Add-on hotfix for VisiBroker 8.5

You can download it from Micro Focus Supportline

If you have not yet downloaded the necessary HotFix, click on the Micro Focus Supportline link provided on the screen, and download from there to your local machine. You can now select the HotFix and proceed with the installation.

5 The remaining installer screens will be:

   a **Pre-installation summary**

   b **Installation** (this will show a progress of the installation as it occurs)

   c **Final post-installation screen** mentioning the ORB installation that was upgraded, and how to install the CORBA Add-on for REST license (which is required before use).

Silent Installation

As an alternative to the GUI installation described above, you can install silently.

A silent installation runs without user interaction, and is typically used to automate installation across multiple machines. Download the installer as described in "Installation Instructions". Instead of specifying the installation parameters via the screens of the GUI, the parameters are stored in an installer properties file.

Sample silent installer properties file

A sample properties file for the CORBA Add-on for REST silent installer would look as follows:

```
#
ORB_INSTALLATION=<path to orb installation>
REST_ADDON_HOTFIX=<path to rest add-on hotfix>
INSTALLER_UI=silent
```

Performing silent installation

To perform a silent installation, specify silent mode by using the `-i` switch on the command line.

- **Windows**: `mf_rest_corba_addon_<version>_win.exe -i silent`
- **Linux**: `mf_rest_corba_addon_<version>_unix.bin -i silent`

If the installer properties file is named `installer.properties` and is in the current directory, it will be automatically picked up. To specify a file with a different filename or in a different location, use the `-f` command line switch.

- **Windows**: `mf_rest_corba_addon_<version>_win.exe -i silent -f c:\temp\installer_win.properties`
- **Linux**: `mf_rest_corba_addon_<version>_unix.bin -i silent -f /tmp/installer_linux.properties`

Licensing the Product

Before using the CORBA Add-on for REST product, you need to register and activate the license you received for your product. The license may be in the form of an email from Micro Focus listing one or more serial numbers and license keys, or it may be a license key file sent to you in an email from Micro Focus.

To license the product we need to run the licensing script:

- **Windows:** `<`*`ORB-installation`*`>\REST\bin\rest_lmadm.bat`
- **Linux:** `<`*`ORB-installation`*`>/REST/bin/rest_lmadm.sh`

The script will open up a GUI wizard. Follow the on-screen instructions to register your license. You can register the product in one of the following ways:

1. **Serial Number**: Where a serial number and activation key are available, please follow these steps to register the product:
   - a **Product Registration Wizard**: Select **Have Serial Number**.
   - b **Serial Number**: Enter the 20-character serial number and the 6-character license key supplied with your installation into the respective fields.
   - c **Select Registration Method**: Choose **Direct**.

     This is the recommended registration method, but note that for direct registration you need internet access from the system on which your product is installed; if you do not have access, choose one of the other methods.
   - d **Account Information**: This is specified as a prerequisite for installing CORBA Add-on for REST. Choose **I have an account**. If you have not yet set up a developer network account, select **I do not have an account**. The wizard prompts you for the information to create an account.
   - e **Account Information**: Enter your developer network account information: login name, email and password.
   - f **Proxy settings**: If you have any proxies configured, check **I use a proxy** and enter your proxy information. If you do not use proxies, just click **Next**.
   - g **Information Summary**: Check the information you have entered and if it is correct click **Next**. Click **Back** if you want to change anything.
   - h **Direct Registration**: Assuming you did select **Direct** earlier, this dialog shows how your registration is progressing. When registration is complete, the **Done** button is enabled. Click **Done**.
2. **Activation File**: If a product activation file or slip file are available, then select **Have Activation File** and follow the on-screen prompts to have the activation file installed and registered.

For help obtaining one of the above, or if you encounter any issues with licensing the product, please contact your local representative via Micro Focus Support.

Troubleshooting

To view debug information from the installer, do the following:

- **Windows:** Hold down the `Ctrl` key immediately after launching the installer until a console window appears.
- **Linux:** To send the debug output to the console, run the installer as follows:

  ```
  LAX_DEBUG=true ./mf_rest_corba_addon_<version>_unix.bin
  ```

Getting Started

This chapter describes using the CORBA Add-on for REST.

Setting the Environment

The first thing is to ensure that your ORB's environment is correctly set up.

Run the Deployment Environment Script

- **Orbix 6**: Before using the CORBA Add-on for REST with Orbix 6, you must have deployed an Orbix domain, and need to have run the deployment environment script that was generated during the deployment process.

  The deployment environment scripts are typically found in the `etc/bin` sub-folder of an Orbix 6 installation. See the ***Orbix 6 Deployment Guide*** for further information.
- **VisiBroker**: Run the `vbroker` script in the `bin` folder of the VisiBroker installation. See the ***VisiBroker Installation Guide*** for information on this script.

The installation process described in the previous chapter ("Installing the CORBA Add-on for REST") created a sub-folder called `REST` inside the ORB installation folder. This sub-folder contains the CORBA Add-on for REST installation.

Run the rest_env Script

To source the environment for the CORBA Add-on for REST, run the script called `rest_env`, which is located in the `bin` folder.

Now your environment is correctly set up and you can start using the product.

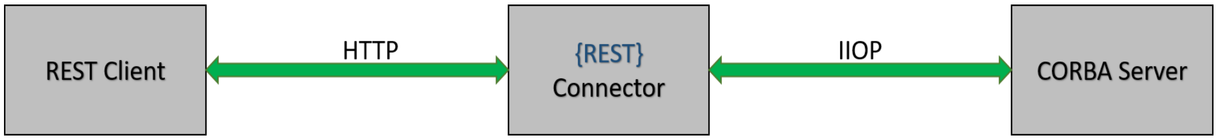
The Typetest Example

This chapter describes how to create, build and run the `typetest` example that is included in the CORBA Add-on for REST product. The demonstration program can be found under the `demos` folder in the product installation.

The accompanying `README.txt` file contains detailed information on how to run the demonstration program. The demonstration program contains a CORBA server, a custom-generated REST Connector component, and some sample REST clients demonstrating how clients written using a simple REST framework can be used to talk to a CORBA server, via the REST connector component.

Before you run the example program, it is important to understand how the program uses the new IDL-RS annotations along with the **idl2rest** tool to generate a custom REST connector. See the chapter "IDL-RS Annotations" for details.

The following diagram shows the different parts of the demo and how they work together.



There are three main components:

- **REST Client**: This is a client which can be written in any number of languages and frameworks. The typical requirement is a HTTP library, and the data serialization library (such as XML or JSON).
- **REST Connector**: This is the custom-generated component which has been generated with the **idl2rest** tool, and translates or maps your HTTP messages into CORBA messages that the CORBA server can understand.
- **CORBA Server**: This is your CORBA server, which may be one of the Micro Focus ORBs listed in "Components".

The `typetest` demonstration program uses the IDL-RS annotations to create a REST API from the CORBA IDL.

The excerpt below shows the TypeTest interface, and some annotated IDL operations. This shows how, by using a few IDL-RS annotations, it is possible to add a REST API that can in turn be mapped into the desired CORBA calls to invoke on the CORBA server.

```
@Path(uri = "/typetest", rir = "file:../typetest_objref.txt")
interface TypeTest
{
    enum Beer { Wheat, Lambic, Bitter, Stout, Porter };

    @Path(enum_inout)
    @POST
    void enum_inout (inout Beer beerEnum);

    @Path("sysexc_op")
    @POST
    void sysexc_op();

    @HTTPStatus(responseCode = 414,
              description = "A new user exception")
    exception UserExc
    {
        long    m1;
        boolean m2;
        string  m3;
    };

    @Path("userexc_op")
    @POST
    void userexc_op() raises(UserExc);
...
...
...
};
```

If you look at the interface definition, you can see that the `MF_TypeTest::Typetest` interface is annotated with the `@Path` annotation. This will bind the interface to the URI `/typetest` and refer to the backend CORBA Object contained in the file `typetest_objref.txt`.

For example, to call to the `TypeTest::sysexc_op()` operation, you could issue the following HTTP Request to the URI:

```
    <base_uri>/typetest/sysexc_op
```

with the HTTP `POST` method.

The structure of the demonstration

The directory and file structure of the `typetest` example, as it is installed, is shown below. Inside the demonstration folder are the following important folders:

- `idl` contains the IDL that has been annotated with IDL-RS annotations.
- `rest_clients` contains a number of different clients that have been written with the following REST frameworks:
  - JAX-RS/REST
  - Python
- `rest_connector` contains the build scripts and README files required to build and run the connector.
- `corba_server` contains the build scripts, source and README files to run the CORBA server.

The typical folder layout for the `typetest` demonstration is as follows:

```
<REST_HOME>/demos/classes
<REST_HOME>/demos/typetest
<REST_HOME>/demos/typetest/idl
<REST_HOME>/demos/typetest/rest_clients/jax_rs
<REST_HOME>/demos/typetest/rest_clients/jax_rs/classes
<REST_HOME>/demos/typetest/rest_clients/jax_rs/src
<REST_HOME>/demos/typetest/rest_clients/python
<REST_HOME>/demos/typetest/rest_connector/idl2rest_output
<REST_HOME>/demos/typetest/rest_connector/<build-scripts>
<REST_HOME>/demos/typetest/rest_connector/<readme files>
<REST_HOME>/demos/typetest/corba_server/src/
<REST_HOME>/demos/typetest/corba_server/java_output
<REST_HOME>/demos/typetest/corba_server/<build-scripts>
<REST_HOME>/demos/typetest/corba_server/<readme files>
```

Building the Demonstration

This section gives instructions for building this demonstration for Orbix 6 and VisiBroker. These instructions build the CORBA Server, and the generated REST connector.

Orbix 6

The Orbix 6 demo is run with the `itant` tool that is distributed with Orbix 6.

Windows:

```
cd %REST_HOME%\demos\typetest\corba_server
itant
```

UNIX:

```
cd $REST_HOME/demos/typetest/corba_server
itant
```

VisiBroker

The VisiBroker demonstration follows the demonstration build system that VisiBroker demos use, which is a Makefile for UNIX platforms, and a batch file script on Windows systems.

Windows:

```
cd %REST_HOME%\demos\typetest\corba_server
vbmake.bat
```

UNIX:

```
cd $REST_HOME/demos/typetest/corba_server
make all
```

Running the Demonstration

This consists of:

- Running the CORBA Server
- Scenarios for Deploying the REST Connector

Running the CORBA Server

Orbix 6

Windows:

```
cd %REST_HOME%\demos\typetest\corba_server
java -classpath .\classes;"%CLASSPATH%" typetest.MF_TypeTest.Server
```

UNIX:

```
cd $REST_HOME/demos/typetest/corba_server
java -classpath classes:$CLASSPATH typetest.MF_TypeTest.Server
```

VisiBroker

Windows:

```
cd %REST_HOME%\demos\typetest\corba_server
vbj -classpath .\classes;%CLASSPATH% typetest.MF_TypeTest.Server
```

UNIX:

```
cd $REST_HOME/demos/typetest/corba_server
vbj -classpath ./classes:$CLASSPATH typetest.MF_TypeTest.Server
```

Scenarios for Deploying the REST Connector

The section discusses how to take your existing CORBA server deployment and deploy the REST Connector in three different ways. Each of the sections describes deploying a new REST Connector instance. These demonstrate how REST Connectors can be deployed in different circumstances, such as:

- Deploying the REST Connector insecurely
- Deploying the REST Connector securely
- Deploying the REST Connector to serve a versioned REST API

Deploying the REST Connector insecurely

At this point your CORBA server is compiled and running, and the remaining component to run is the REST Connector.

In the `bin` folder of the installation is a shell script called:

- **Windows:** `rest_connector.bat`
- **UNIX:** `rest_connector.sh`

For details on the options that can be passed into the `rest_connector` script, see “Connector Options”.

Orbix 6

Windows:

```
cd %REST_HOME%\demos\typetest\rest_connector
itant connector.compile
```

UNIX:

```
cd $REST_HOME/demos/typetest/rest_connector
itant connector.compile
```

VisiBroker

Windows:

```
cd %REST_HOME%\demos\typetest\rest_connector
vbmake.bat
```

UNIX:

```
cd $REST_HOME/demos/typetest/rest_connector
make -e all
```

At this point the build system has taken the annotated IDL file, and compiled it with the **idl2rest** tool. The resulting generated code has been compiled.

You are now ready to run the REST Connector. The `rest_connector` script takes an argument that is the package name for the connector to scan for generated code packages at runtime.

Windows:

```
%REST_HOME%\bin\rest_connector.bat --hostname REST-HOST
--config-file generated_config\connector.properties
```

UNIX:

```
$REST_HOME/bin/rest_connector.sh --hostname REST-HOST
--config-file generated_config/connector.properties
```

The excerpt below shows a sample output of what the Connector outputs when it runs:

```
Nov 11, 2019 9:43:15 AM
com.microfocus.rest4corba.config.FlatPropertiesFileConfigurationH
andler loadProperties
INFO: Loaded properties from generated_config\
connector.properties: property count = 1
uriBasename: http://REST-HOST
Nov 11, 2019 9:43:18 AM
org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [REST-HOST:8080]
Nov 11, 2019 9:43:18 AM
org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
```

Once the connector is running you can run the REST clients against it. The sources to the REST clients are located in the `rest_clients` sub-folder in the demonstration (see “The structure of the demonstration”). See the `README` files located in these folders for more information on running the provided REST clients.

The excerpt below invokes an API using a curl command to send a HTTP request to a URI. This example invokes a `POST` method on the `enum_inout` URI. It also informs the Connector what format the request and response messages are to contain. This is controlled via HTTP headers. In this example, the request body is sending JSON, and XML is asked for in the response message.

Request:

```
curl -s http://REST-HOST:8080/typetest/enum_inout -H
'Content-Type: application/json' -H 'Accept: application/xml' -X
POST -d '{"val": {"kind": "Porter"}}'
```

Response:

```
<enum_inoutResponseWrapper xmlns=""><val><kind>Wheat</kind></
val></enum_inoutResponseWrapper>
```

Note:
This request example is for a UNIX installation. For considerations about using curl on Windows, see "Using curl".

Java (JAX-RS):

The following excerpt is taken from the `JAXRSClientJson` class that is provided in the typetest demonstration. It uses the JAX-RS APIs to build up and parse the request and response.

```
WebTarget target = client.target(baseUrl + "enum_inout");
String inputData = "{\"val\":{\"kind\":\"Porter\"}}";
Response response = target.request("application/
json").post(Entity.json(inputData));
if(response.getStatus()!=200){
    System.out.println("Did not get expected http response for
POST enum_inout expected 200");
    throw new RuntimeException("HTTP Error: "+
response.getStatus());
}
System.out.println("Correct OK Response from the Server for POST
enum_inout ");
String result = response.readEntity(String.class);

if(result.contains("Wheat") == true)
    System.out.println("Correct value returned for enum_inout
request " + result);
else {
    System.out.println("Wrong value returned for enum_inout " +
result + " not contain Wheat");
    System.exit(1);
}
```

Python:

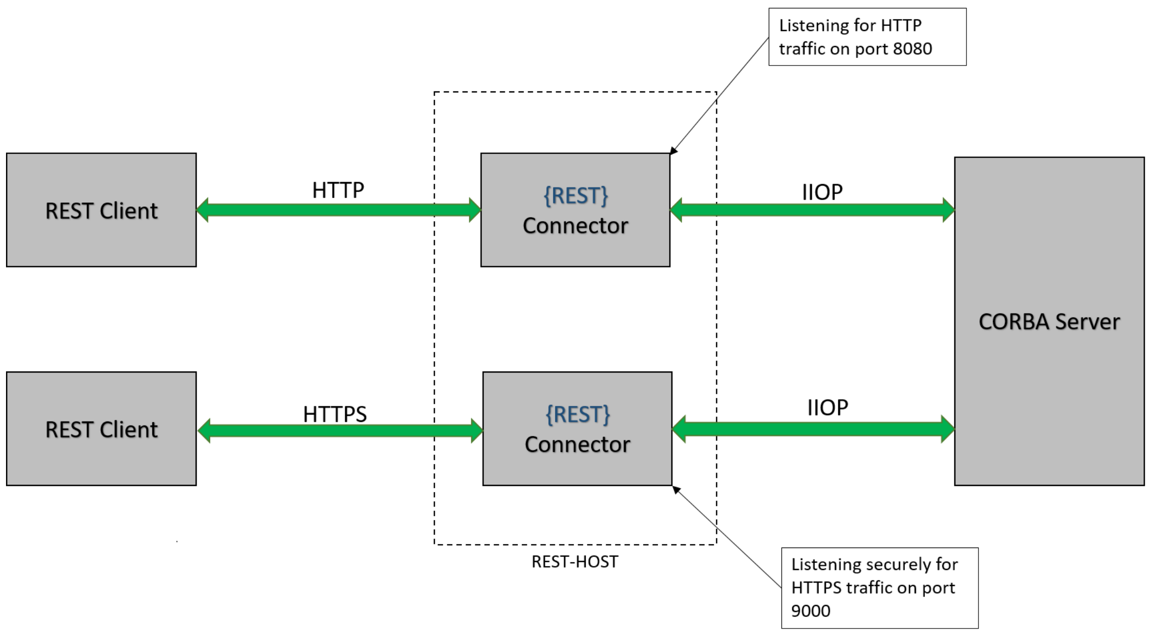
The following excerpt taken from the Python client for the same API uses an HTTP and a JSON library to build up and parse the request and response.

```
url_enum_inout = baseurl + "enum_inout"
r =  session.post(url_enum_inout, json =
{'val':{'kind':'Porter'}})
if r.status_code == 200:
    print ("GOOD http response for enum_inout     200 OK")
else:
    exit("Exiting as bad return value for enum_inout  " +
r.text);
check_string = str(r.text)
if check_string.find("Wheat") == -1:
    exit("Exiting as bad return value for enum_inout\n");
else:
    print ("enum_inout successful " + r.text + "\n")
```

Deploying the REST Connector securely

Until this point, we have described running the REST Connector and the REST clients against a HTTP endpoint. This section describes how to turn on TLS security so that you can run the connector to serve secure HTTP traffic (https).

This section builds upon the previous deployment of an insecure REST Connector, which you deployed on the host REST-HOST on port 8080, serving the insecure HTTP traffic. You can now build upon this to run another instance of the REST Connector, this time only serving secure HTTPS traffic on port 9000. The diagram below visualizes this deployment. The REST client-side application and the CORBA server application can reside on the same host or on different hosts.



Generating TLS certificates

For ease of use with secure running, a script is provided in the `etc` sub-folder of the installation for generating the TLS certificates used for the secure demonstration. These TLS certificates must be generated because the X509 extension `SubjectAlternateName` needs to be used to provide alternate hostnames and IP addresses that are host-specific.

See the `README.txt` file in the `etc` subfolder of the CORBA Add-on for REST installation for a detailed background and instructions for generating and installing these TLS certificates.

Note:
If you are running the `rest_connector` and the REST client(s) on separate machines, it is necessary to generate the certificates on the connector machine and then copy the required certificates to the appropriate location on the client machine(s).

Connector properties

Once the certificates are installed, ensure that you pass the `--secure` option to the `rest_connector` script. You must also pass the `--hostname` option to the script, and the hostname must match one of the hostnames present in the TLS certificate's `SubjectAlternateName` (as described in "Generating TLS certificates").

The `--hostname` and `--secure` flags are also required when running the REST clients provided with this demonstration. These flags ensure that the correct TLS certificates are configured. See the README files in the `rest_clients/jax_rs` and `rest_clients/python` sub-folders for detailed instructions on how to do this.

This section describes and demonstrates this approach, and assumes that you already have your TLS certificates generated at this point (see "Generating TLS certificates" for details).

As no change is needed to the IDL, you do not need to re-run the **idl2rest** tool.

```
mkdir secure_rest_connector
cd secure_rest_connector
```

Copy the previous `connector.properties` file to the current folder, and rename it `secure_connector.properties`. You can edit the contents of this copy to enable security.

Now configure the certificates. Open the `secure_connector.properties` configuration file and uncomment these four TLS configuration items:

- `REST_KEYSTORE_SERVER_FILE`
- `REST_KEYSTORE_SERVER_PWD`
- `REST_TRUSTSTORE_SERVER_FILE`
- `REST_TRUSTSTORE_SERVER_PWD`
- `REST_CONNECTOR_SECURE` and set this to `true`

To set the port number that the connector will use, uncomment the configuration variable `REST_CONNECTOR_PORT` and change its value to **9000**.

The following example shows the main points that need configuring highlighted in bold:

```
# The port number the HTTP server will listen for HTTP traffic, by default the
# HTTP server will listen on port: 8080
#
REST_CONNECTOR_PORT = 9000
# eg: REST_CONNECTOR_PORT = 8080
######################################################################
# Begin Security deployment settings for the HTTP Server
#
#
# HTTP server SSL configuration
# Use of SSL is controlled via the URI base.
#
#
# Turn on HTTPs security, by default this is set to false.
#
# REST_CONNECTOR_SECURE = <true|false>
REST_CONNECTOR_SECURE = true
#
# Authenticate the client?
#
# false:  The client authenticates the server.
# true:   The client authenticates the server,
#         and the server authenticates the client.
#
REST_AUTHENTICATE_CLIENT = true
#
# Relative paths are based off the directory in which
# the rest_connector script was launched.
#
REST_KEYSTORE_SERVER_FILE  = C://REST/etc/sslconfig/keystore_server.jks
REST_KEYSTORE_SERVER_PWD  = keystoreserverpass
REST_TRUSTSTORE_SERVER_FILE= C://REST/etc/sslconfig/truststore_server.jks
REST_TRUSTSTORE_SERVER_PWD  = truststoreserverpass
#
#
# End of security properties
###################################################################
```

Now you can run the connector securely using the following command:

Windows:

```
%REST_HOME%\bin\rest_connector.bat --hostname REST-HOST
--config-file secure_connector.properties
```

UNIX:

```
$REST_HOME/bin/rest_connector.sh --hostname REST-HOST
--config-file secure_connector.properties
```

```
Nov 12, 2019 12:02:54 PM
com.microfocus.rest4corba.config.FlatPropertiesFileConfigu
rationHandler loadProperties
INFO: Loaded properties from secure_connector.properties:
property count = 7
uriBasename: https://REST-HOST
Nov 12, 2019 12:02:57 PM
org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [REST-HOST:9000]
Nov 12, 2019 12:02:57 PM
org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Nov 12, 2019 12:02:57 PM
com.microfocus.rest4corba.CorbaRestStandaloneServer run
INFO: Started server on: https://REST-HOST:9000
```

You can run a curl command to simulate a client sending HTTPS requests to your secure Connector instance as follows:

Request:

```
curl -s --cacert <PATH_TO_CERTS>/caserver_export.pem
https://REST_HOST:9000/typetest/enum_inout -H
'Content-Type: application/json' -H 'Accept: application/
xml' -X POST -d '{"val": {"kind":"Porter"}}'
```

Response:

```
<enum_inoutResponseWrapper xmlns=""><val><kind>Wheat</
kind></val></enum_inoutResponseWrapper>
```

Note:
This request example is for a UNIX installation. For considerations about using curl on Windows, see "Using curl".

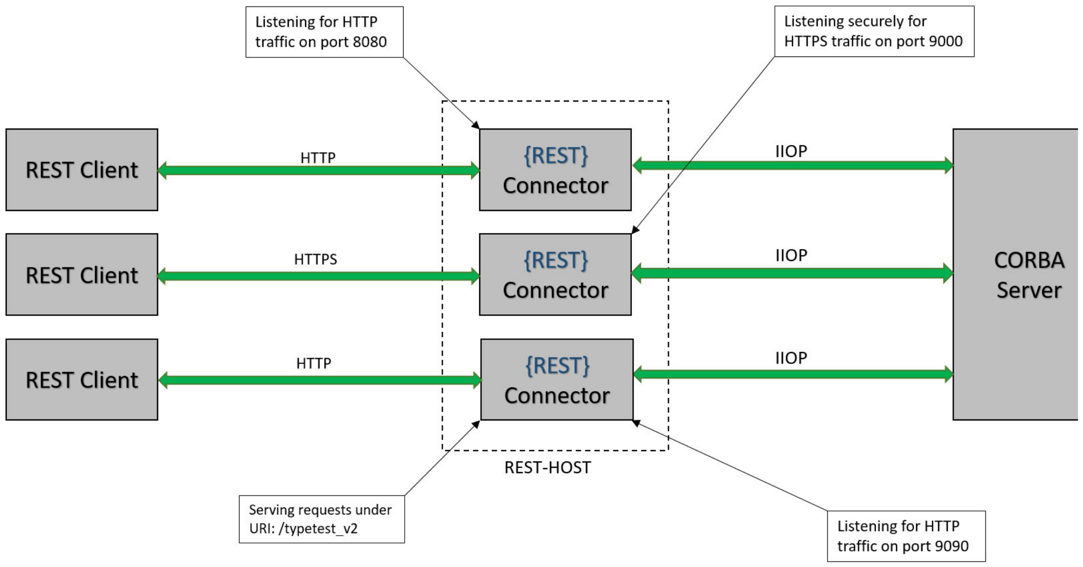
The JAX-RS and Python clients can also be run securely against the new securely-deployed Connector, serving requests from port 9000 on the host `REST-HOST`. Consult the `README` files in the respective rest client folders on the exact command line options that are required to configure the clients to run with the client-side TLS demo certificates.

Deploying the REST Connector to serve a versioned REST API

The previous section described how you could run a single Connector instance, and run some REST clients to talk to the Connector.

This section shows how to extend the existing CORBA IDL file that was previously annotated with IDL-RS annotations. It will demonstrate that by just changing some of the IDL-RS annotation values, you can construct a different REST API. In addition you will root the entry point of the modified API at a different URI. The example that follows changes the root URI from: `/typetest` to `/typetest_v2`.

The following illustration shows the structure of the revised deployment.


Listening for HTTP traffic on port 8080
Listening securely for HTTPS traffic on port 9000
REST Client
HTTP
{REST} Connector
IIOP
REST Client
HTTPS
{REST} Connector
IIOP
CORBA Server
REST Client
HTTP
{REST} Connector
IIOP
REST-HOST
Serving requests under URI: /typetest_v2
Listening for HTTP traffic on port 9090

During this example the CORBA server does not need to be brought down, or reconfigured. The existing REST connectors that were used in the previous sections can also be left running.

This section describes:

- Extending the IDL to include more IDL-RS annotations
- Compiling the IDL by running the **idl2rest** tool but compiling the generated code into a separate package, `typetest_v2`
- Running the connector and configuring it with the `typetest_v2`. The connector will be running on a separate port
- Running the REST clients against the new connector, showing how you can point the clients at the updated modified API

Extending the IDL

First, you can extend the previous IDL file. The changes are highlighted in bold text below.

```
@Path(uri = "/typetest_v2", rir = "file:../typetest_objref.txt")
interface TypeTest
{
    enum Beer { Wheat, Lambic, Bitter, Stout, Porter };

    @Path(enum_inout)
    @POST
    void enum_inout (inout Beer beerEnum);

    @Path("sysexc_op")
    @POST
    void sysexc_op();

     @HTTPStatus(responseCode = 408,
              description = "A new user exception")
    exception UserExc
    {
        long    m1;
        boolean m2;
        string  m3;
    };

    @Path("userexc_op_new")
    @POST
    void userexc_op() raises(UserExc);

    @Path("long_in/{arg1}")
    @POST
    void long_in(
        @PathParam("arg1")
        in long val
    );
```

In the first change, the @Path annotation changes the uri to `/typetest_v2`. This ensures that the connector serves requests from the resource at the `/typetest_v2` URI.

The next change is that the CORBA user exception is now annotated with the `@HTTPStatus` annotation, and the URI for the `userexc_op()` is now changed to `userexc_new_op`.

Compiling the IDL

The IDL can be compiled in the same way as before, with the following change to the demo build-system.

Orbix6

Windows:

```
cd %REST_HOME%\demos\typetest\rest_connector
itant connector.compile -Drest.package=typetest_v2
```

UNIX:

```
cd $REST_HOME/demos/typetest/rest_connector
itant connector.compile -Drest.package=typetest_v2
```

VisiBroker

Windows:

```
cd %REST_HOME%\demos\typetest\rest_connector
vbmake.bat typetest_v2
```

UNIX:

```
cd $REST_HOME/demos/typetest/rest_connector
make -e REST_PACKAGE_NAME=typetest_v2
```

The above commands show how you can generate and compile the modified IDL file (which only has IDL-RS changes) to create a new REST API rooted under `/typetest_v2`.

Running the connector

At this point you can run the connector in a number of different ways. In every case, you must configure two important pieces of information:

1. **The port number:** as you already have a connector running on port 8080, serving the `/typetest` URI, you need to choose a new port.
2. **The REST package** to use when the Connector starts.

The sections "Overriding the Configuration Variables" and "Connector Options" document the various ways that this configuration can be provided to the Connector.

In this section, we will configure the Connector by passing command-line options to the `rest_connector` script.

Windows:

```
%REST_HOME%\bin\rest_connector.bat --hostname REST-HOST
--port 9090 --rest-package typetest_v2 --config-file
generated_config/connector.properties
```

UNIX :

```
$REST_HOME/bin/rest_connector.sh --hostname REST-HOST
--port 9090 --rest-package typetest_v2 --config
generated_config/connector.properties
```

Running the REST clients

You can run your REST clients against the new connector as follows:

```
$ curl http://REST-HOST:8080/typetest/userexc_op -X POST -v
> POST /typetest/userexc_op HTTP/1.1
> User-Agent: curl/7.37.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 414 Request-URI Too Long
< Cache-Control: no-cache
< Content-Type: application/json
< Date: Fri, 22 Nov 2019 11:06:21 GMT
< Connection: close
< Content-Length: 30
<
* Closing connection 0
A new user exception

$ curl http://REST-HOST:9090/typetest_v2/userexc_op_new -X POST
-v

[SNIP..]

> POST /typetest_v2/userexc_op_new HTTP/1.1
> User-Agent: curl/7.37.0
> Host: localhost:9090
> Accept: */*

[SNIP…]

< HTTP/1.1 408 Request Timeout
< Cache-Control: no-cache
< Content-Type: application/json
< Date: Fri, 22 Nov 2019 11:07:14 GMT
< Connection: close
< Content-Length: 30
<
* Closing connection 0
A new user exception
```

In the example above, the first `curl` command sends a request to the URI `/typetest/userexc_op` on port 8080. This returns an HTTP response code of `414` with the message "A new user exception". You can see this by looking at the IDL-RS annotation in the IDL excerpt in the section "The Typetest Example".

The second `curl` command above changes the URI and port, from `/typetest/user_exc_op` on port 8080 to `/typetest/user_exc_op_new` on port 9090. The connector on port 9090 has responded with an HTTP response code of `408`, and the message "A new user exception".

IDL-RS Annotations

The IDL-RS annotations are a new concept; the notion of IDL annotations was recently added in the IDL 4 specification. The IDL-RS annotations follow on from this notion of IDL annotations, to define a set of IDL annotations that are applicable to decorate an IDL file to allow it to describe selective IDL components, and to aid in providing a mapping from certain IDL constructs to a REST API.

Annotations Used

The current list of implemented IDL-RS annotations used by the product are:

- **@Path**: This annotation is used to bind a HTTP URI (Uniform Resource Locator) to a particular IDL construct. The annotation takes two forms:
  - Where the annotation only needs to mention the URI, some examples would be: `@Path("/bank")`, `@Path("/op/{`*`name`*`}")`.

    In the previous example, `@Path("/op/{`*`name`*`}"`, the text `{`*`name`*`}` is referred to as a *URI path template parameter*. This is like a wildcard that is replaced at runtime with a matching URI path segment; for example in this case the URI `/op/my-name` would match. URI path template parameters are typically used in conjunction with the `@PathParam` annotation.
  - Where the annotation needs to be told about a persistent CORBA object, along with the URI. In this form the annotation would look like: `@Path(uri = "/bank", rir = "corbaloc::localhost:3075/ BankRef")`.
- **@PathParam**: This annotation is applied to an IDL operation parameter to provide a binding between a URI path template parameter and an IDL operation parameter. The URI path template parameter is specified with the `@Path` annotation. An example would be:

  ```
  @Path("/op/{arg1}")
  void op(@PathParam("arg1") in string msg);
  ```
- **@QueryParam**: This annotation is applied to an IDL operation parameter to provide a binding between a URI query parameter and an IDL operation parameter. A URI query parameter forms part of a URI query string, which takes the form: *?name-1=value-1&name-2=value2*. An example usage would be:

  ```
  @Path("/op")
  void op2(@QueryParam("id") in long id,
           @QueryParam("name") in string name);
  ```

  For example, the following URI could be used to access the above IDL operation: `/op?id=1&name=Peter`
- **@GET**: Maps to the HTTP GET method.
- **@POST**: Maps to the HTTP POST method.
- **@PUT**: Maps to the HTTP PUT method.
- **@DELETE**: Maps to the HTTP DELETE method.

- **@Consumes**: Can be used to tell the runtime what media-type can be consumed in HTTP requests. The current supported media-type values are:
    - `application/json`
    - `application/xml`
- **@Produces**: Can be used to tell the runtime what media-type can be produced in HTTP responses. The current supported media-type values are:
    - `application/json`
    - `application/xml`
- **@HTTPStatus**: This annotation is used to provide a custom mapping for user exceptions. This mapping can map a user exception to a HTTP response status code, and have the response body contain a custom message.

  An example of the annotation would be: `@HTTPStatus(responseCode = 404, description = "Invalid Operation")`.

Where possible the IDL-RS annotations have tried to stay close to the JAX-RS annotations that Java REST developers would be familiar with. With this in mind, any developers coming from that development background should be very comfortable with using the IDL-RS annotations to annotate an existing IDL.

IDL Type Serialization

This chapter describes using the CORBA Add-on for REST product to serialize IDL types into two different data formats:

- *JSON (JavaScript Object Notation).*
- *XML (Extensive Markup Language).*

These are self-describing data formats that are used extensively when building distributed systems using REST.

JSON Mapping for IDL Types

The mapping for JSON serialization is based on the DDS-JSON OMG specification as a starting point. The serialized content is the actual data payload, and defining a schema for the JSON representation of the original IDL type is less significant.

For more information about JSON see json.org.

JSON Mapping for Primitive IDL Types

Where possible the CORBA Add-on for REST tries to map IDL primitive types to their nearest counterpart in the JSON data format.

Integer types

The following IDL integer types are represented as JSON integer number type:

- `short`
- `unsigned short`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`

As JSON can only represent integral values as decimal, any hexadecimal or octet value will be serialized as their decimal equivalent value.

Floating point types

The following IDL floating-point types are represented as the JSON number type (integer, fraction, or exponent number types):

- `float`
- `double`
- `long double`

Character types

The following IDL character types are represented as the JSON string type:

- `char`
- `wchar`

Boolean type

The IDL `boolean` type is represented in JSON as either a `true` or a `false` value.

Octet type

The IDL `octet` type is represented in JSON as a positive number in the range 0 – 255 inclusive.

JSON Mapping for Arrays and Sequences

JSON has native support for defining lists or array structures within its syntax.

The following sequence in IDL:

```
typedef sequence<long> LongSeq;
```

would be serialized in JSON as:

```
[10, 2, 4, 5]
```

The same mapping would exist for an IDL Array as for an IDL Sequence.

Where the elements of the IDL Array or Sequence are of IDL primitives, they are serialized to their nearest JSON counterpart. See "JSON Mapping for Primitive IDL Types".

JSON Mapping for Complex IDL Types

This section looks at how some of the more complex IDL types are serialized to the JSON data format. Where possible a trade-off has been made between payload size and readability.

Enum types

The following IDL enum type:

```
enum Colors {RED, BLUE, GREEN};
```

will be serialized into the following JSON payload:

```
{"kind": "BLUE"}
```

In the above example the `Colors` enum currently has the value of `BLUE`. The value associated with the "`kind`" key is the stringified representation of the current enum value.

Struct type

The following IDL struct type:

```
struct DateOfBirth {
    octet day;
    octet month;
    unsigned short year
};

struct CustomerType {
    String name;
    DateOfBirth dob;
}
```

will get serialized to the following JSON payload:

```
{
    "name": "Peter",
    "dob": {
        "day": 4,
        "month": 3
        "year": 1970
     },
}
```

Union type

The following IDL union type:

```
enum IPVerEnum {E_IPV4, E_IPV6, E_IP_UNKNOWN};

union IPVersion switch(IPVerEnum) {
  case E_IPV4:
    octet ipv4[4];
  case E_IPV6:
    octet ipv6[16];
  default:
};
```

will be serialized to the following JSON payload:

```
{
    "discriminator": {
        "kind": "E_IPV6"
    },
    "value": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
}
```

XML Mapping for IDL Types

This section defines how types defined in IDL are serialized in XML.

XML Mapping for Primitive IDL Types

IDL primitives are represented in XML as described in the following sections.

Integer types

The value of instances of the following IDL integer data types are represented as an XML element value set to the decimal string representation of the integer:

- `short`
- `unsigned short`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`

For example, an instance of an IDL long type defined in IDL:

```
long return_code
```

will be represented in XML as:

```
<return_code>50000</return_code>
```

Floating point types

The following IDL floating-point types will have their numeric representation serialized as their decimal value inside the corresponding XML element:

- `float`
- `double`
- `long double`

For example the following `long` defined in IDL:

```
float float_val
```

This will be serialized into XML as:

```
<float_val>-1.1225E8</float_val>
```

Character types

The following IDL character types hav ee their character representation serialized inside the corresponding XML element:

- `char`
- `wchar`

For example the following `char` defined in IDL:

```
char char_val
```

This will be serialized into XML as:

```
<char_val>x</char_val>
```

Boolean types

The IDL boolean type will have its representation serialized inside the corresponding XML element. The value of the type will be represented as either `true` or `false`.

For example consider the following boolean defined in IDL:

```
boolean bool_val
```

This will be serialized into XML as:

```
<bool_val>false</bool_val>
```

Octet type

The IDL octet type will have their its representation serialized inside the corresponding XML element. This will contain a serialized value in the range 0-255.

For example the following `octet` defined in IDL:

```
octet octet_val
```

This will be serialized into XML as:

```
<octet_val>254</octet_val>
```

XML Mapping for Arrays and Sequences

IDL sequence and array types are mapped to an XML element with the same name as the sequence or array name. Inside the sequence element there exists a single XML element called `item` for each member in the sequence or array. Each `<item>` element will then be serialized based on its definition.

For example consider the following IDL sequence of octets:

```
typedef sequence<octet> OctetSeq;
```

This will be serialized into XML as:

```
<OctetSeq>
    <item>2</item>
    <item>3</item>
    <item>5</item>
</OctetSeq>
```

XML Mapping for Complex IDL Types

This section looks at how some of the more complex IDL types are serialized to the XML data format. Where possible a trade-off has been made between payload size and readability.

Enum type

The following IDL enum type:

```
enum Colors {RED, BLUE, GREEN};
```

will be serialized as the following JSON payload:

```
<kind>BLUE</kind>
```

In the above example the `Colors` enum currently has the value of `BLUE`. The value associated with the "`kind`" XML node is the stringified representation of the current enum value.

Struct type

The following IDL struct type:

```
struct DateOfBirth {
    octet day;
    octet month;
    unsigned short year
};

struct CustomerType {
    String name;
    DateOfBirth dob;
}
```

will get serialized as the following XML payload:

```
<CustomerType>
    <name>Peter</name>
    <DateOfBirth>
        <day>4</day>
        <month>3</month>
        <year>1970</year>
    </DateOfBirth>
</CustomerType>
```

Union type

The following IDL union type:

```
enum IPVerEnum {E_IPV4, E_IPV6, E_IP_UNKNOWN};

union IPVersion switch(IPVerEnum) {
  case E_IPV4:
    octet ipv4[4];
  case E_IPV6:
    octet ipv6[16];
  default:
};
```

will be serialized as the following XML payload:

```
<IPVersion>
    <discriminator>
         <kind>E_IPV6</kind>
    </discriminator>
    <value>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
        <item>0</item>
    </value>
</IPVersion>
```

IDL Request and Response Wrapping

This chapter describes how the CORBA Add-on for REST uses request and response wrapping to meet HTTP's requirement to deal with a single object.

Wrapped Objects

One of the advantages of using CORBA is its very rich way of sending multiple data items from servers to clients using a combination of response values that can use not only the return type of an IDL operation, but also the `inout` or `out` parameter modes.

While this makes CORBA very powerful and flexible, it does pose an issue with mapping request parameters, and responses. HTTP defines its request and response bodies to contain a **single** entity or object only. A solution to this is to wrap the REST parameters or responses into a wrapped object. This wrapped object can then be serialized into one of the supported data formats (JSON or XML), using the serialization rules defined in "IDL Type Serialization".

The code generation tool **idl2rest** will automatically generate code to wrap request and response data. The generated classes are used to describe how to serialize and unserialize a data representation of the request and response data. They are serialized as follows:

- **JSON**: there is an anonymous JSON object, with each member having the same name as it does in the IDL file. For example, the IDL operation parameter names are preserved.
- **XML**: the root XML element is named as follows:
  - The IDL operation or attribute name as defined in the IDL file
  - The suffix `Request` is used for a request, and `Response` for a response.

Note:

This is a slight change from version 1.0 of the CORBA Add-on for REST, where the suffix was either `RequestWrapper` or `ResponseWrapper`. For interoperability the idl2rest tool provides a command line flag to revert to the old behavior. This flag is described in more detail in "idl2rest Options".

Request Wrappers

In order to support sending multiple IDL parameters in an IDL request, CORBA Add-on for REST needs a means to wrap these parameters together so that it can encode them in the body of a HTTP request.

First, the following code snippet shows an IDL operation annotated with IDL-RS Annotations:

```
@POST
@Path("/op")
boolean op_with_multi_args(
  in string message1,
  in string message2,
  in long long_val,
    inout short short_val
  );
```

The data sent in the HTTP request body would look like this, assuming that it is in JSON data format:

```
  POST /op HTTP/1.1
  [SNIP...]
  {
    "message1" : "contents...",
    "message2" : "more data for our CORBA server",
    "long_val" : 300000,
    "short_val": 2000
  }
```

Or if the same example is in XML format, the HTTP request may look like this:

```
  POST /op HTTP/1.1
  <?xml version="1.0" encoding="UTF-8"?>
  <op_with_multi_argsRequest xmlns="">
    <mesage1>contents...</message1>
    <mesage2>more data for our CORBA server </message2>
    <long_val>300000</long_val>
  </op_with_multi_argsRequest>
```

IDL Parameter Annotations

In this more complex example, the first parameter to the existing IDL operation has been annotated with an IDL-RS annotation.

```
  @POST
  @Path("/op/{id}
  boolean op_with_multi_args(
    @PathParam("id")
    in string message1,
    in string message2,
    in long long_val,
    inout short short_val
  );
```

In this instance the request body would no longer have the key `message1` in its JSON payload. Instead, the content of the `message1` parameter would form part of the URI, as it is annotated with the IDL-RS annotation `@PathParam`:

```
  PUT /op/message1 HTTP/1.1
  [SNIP... ]
  {
    "message2" : "more data for our CORBA server",
    "long_val" : 300000,
    "short_val": 2000
  }
```

Or, in XML:

```
  PUT /op/message1 HTTP/1.1
  [SNIP... ]
  <?xml version="1.0" encoding="UTF-8"?>
  <op_with_multi_argsRequest xmlns="">
    <mesage2>more data for our CORBA server </message2>
    <long_val>300000</long_val>
    <short_val>2000</short_val>
  </op_with_multi_argsRequest>
```

Response Wrappers

If you include the same IDL example as in the previous section:

```
boolean op_with_multi_args(
  in string message1,
  in string message2,
  in long long_val,
  inout short short_val
);
```

This will now result in the following data being written in the HTTP response body (in JSON format):

```
{
  "ret"       : false,
  "short_val" : 24500
}
```

Or in the case of XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<op_with_multi_argsResponse xmlns="">
  <-ret>false</-ret>
  <short_val>24500</short_val>
</op_with_multi_argsResponse>
```

You can see here that irrespective of whether the data format is JSON or XML, the data format contains two pieces of data:

1 The `ret` key or node corresponds to the return value from the IDL operation.

2 The `short_val` key or node corresponds to the IDL parameter that is returned by the CORBA server, as the IDL parameter had the `inout` parameter mode.

As you have seen above, any parameters that have a response-based parameter mode (that is, `inout` and `out`) will be serialized in the HTTP response body.

Advanced HTTP Integration

This chapter looks at some advanced HTTP integration support provided by the CORBA Add-on for REST.

HTTP Cross Origin Resource Sharing support

Cross Origin Resource Sharing (CORS) is a security mechanism available in most modern web browsers. It provides fine-grained control for restricting access to HTTP resources, when requested from a JavaScript-based web application hosted in a different domain. CORS works by setting specific HTTP headers that allow servers to describe which origins are allowed to display HTTP response data in a web browser.

Browsers issuing HTTP requests using HTTP methods other than GET, will first issue what is called the "pre-flight" request. This is usually a HTTP request message with the HTTP method, and the HTTP header origin, set to the URL of the calling HTTP resource's hosting domain. This HTTP request will contain a number of HTTP headers requesting approval from the server; this is accomplished using CORS headers. The server will then respond with pre-flight HTTP response message that may set some CORS response headers. If the correct headers are received by the browser, it will issue the actual HTTP request, and the response data can be displayed in the browser's web application.

For a more detailed look at Cross-Origin Resource Sharing (CORS), see the following online guide:
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

To provide CORS support, the CORBA Add-on for REST product will respond to a browser HTTP request containing the OPTIONS HTTP method, with a HTTP response that will set the following HTTP response headers:

- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Origin`

The CORBA Add-on for REST will set the above CORS response headers, and as such most JavaScript-based web applications that interact with the REST Connector should work without any configuration. If more stringent CORS security configuration is required, it is worth consulting the following configuration variables:

- REST_CORS_HEADER_ALLOW_ ORIGIN
- REST_CORS_HEADER_ALLOW_ HEADERS
- REST_CORS_HEADER_ALLOW_ METHODS

HTTP Caching support

HTTP Caching is a feature in HTTP where a copy of a certain HTTP resource is saved for future use. Any subsequent HTTP requests for that resource will be intercepted and the cached copy of the resource will be sent in the HTTP response.

The CORBA Add-on for REST product allows the HTTP response header `Cache-Control` to be set and configured. For more details on how to configure the header, see:

- REST_CACHE_HEADER_VALUE

Open API Support

This chapter describes how CORBA Add-on for REST supports the OpenAPI Specification and enables you to generate and deploy an OpenAPI model for your REST module.

Introduction

The OpenAPI Specification is an API description format for REST APIs. CORBA Add-on for REST version 1.1 introduces a new feature that enables you to generate a fully compatible OpenAPI v3 model from your annotated IDL.

This powerful feature enables the generated model to be consumed out of the box with popular open source tools such as the OpenAPI Generator tool, and Swagger-UI.

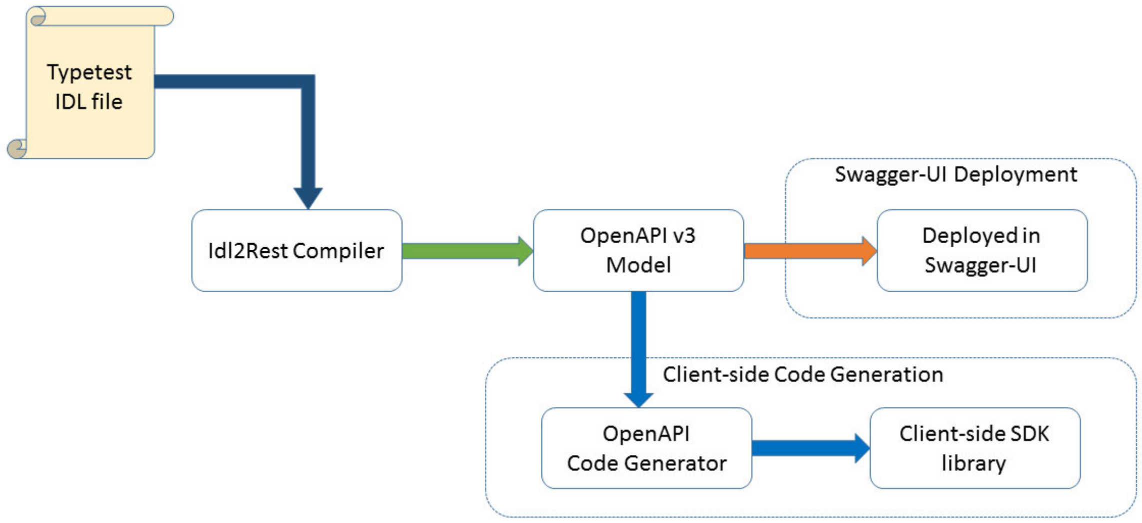
This chapter describes how to use OpenAPI with the CORBA Add-on for REST version 1.1, demonstrating its usage by means of the TypeTest example provided with the product (see "The Typetest Example"). The chapter covers the following areas:

- How to generate the OpenAPI Model, and a brief look at the generated model.
- A demonstration of taking the model to generate a client-side SDK library that greatly simplifies the effort of writing a REST client that can talk to the REST connector.
- Deploying the OpenAPI model into the popular Swagger-UI tool, which provides a rich Web based user interface to visualize and document the OpenAPI model.

For further information:

- For more on the OpenAPI Specification, see https://swagger.io/docs/specification/about/
- For Swagger-UI, see https://swagger.io/tools/swagger-ui/

The following diagram visualizes the workflow of the TypeTest example, demonstrating the two main use cases for the OpenAPI model.



Running the `idl2rest` compiler against an `.idl` file automatically generates the OpenAPI model; no extra command line options or configuration are required.

The generated OpenAPI model is placed in the same folder as the annotated IDL file, and the model will have the same name as the IDL file. For example, after compiling `typetest.idl`, the files `typetest.yaml` and `typetest.json` would be created.

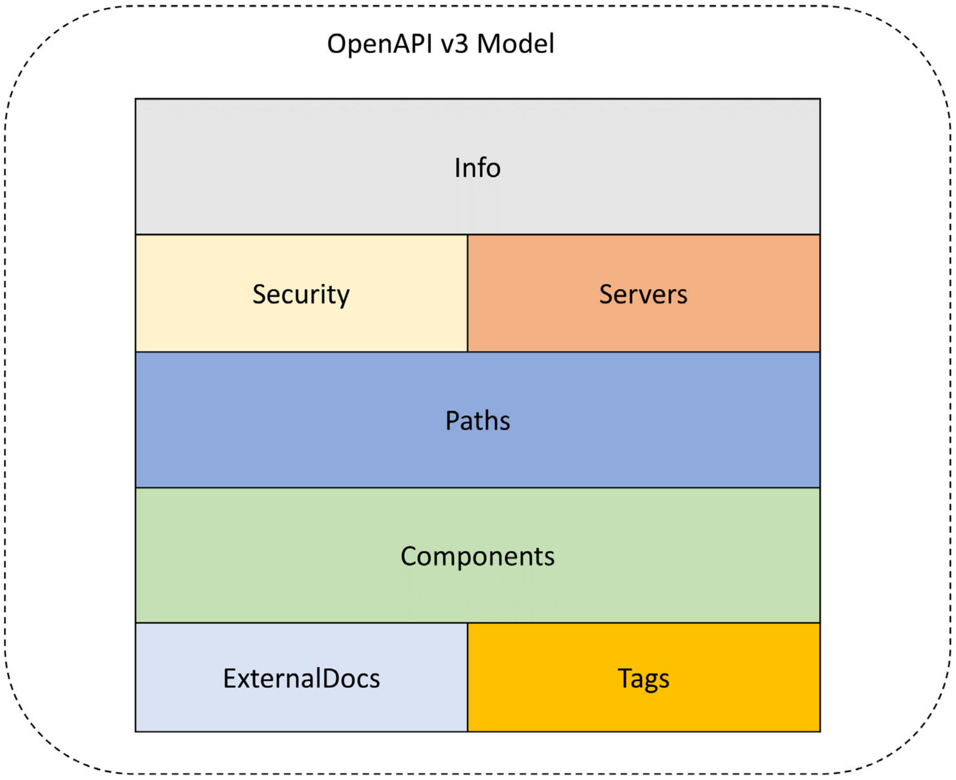
The `idl2rest` tool creates the OpenAPI model in both JSON and Yaml formats. If you only require one format, you can simply ignore or delete the other file.

Note:

The generation of the OpenAPI model is only supported when running the *idl2rest* compiler with Java version 8 or higher. If Java 7 is used, then no OpenAPI files will be generated.

OpenAPI Model structure

Before examining the generation of an OpenAPI model, it is worthwhile looking at the type of information contained in the OpenAPI v3 Model.



The above diagram visualizes how the OpenAPI v3 model is structured.

Without going into too much detail on the structure of the OpenAPI v3 model structure and layout, the model mainly consists of these parts:

- The OpenAPI version.
- The **Info** section contains metadata about the model, such as title of the model, author, license, and a brief textual description of the model and its APIs.
- The **Servers** section contains the URLs of the server or servers that will host the REST APIs that are defined in the Paths section.
- The **Paths** section contains the REST APIs that are defined by the model, and which are implemented by the REST Server (the URLs of these servers should be specified in the Servers section).
- The **Components** section contains any reusable components, which may include schema for data payloads, request bodies, or responses.

For a more detailed look at the OpenAPI Model, see the Specification at https://github.com/OAI/OpenAPI-Specification/blob/3.0.1/versions/3.0.1.md.

Extending the Typetest example to demonstrate OpenAPI integration

Previous sections, starting with "The Typetest Example", took the Typetest CORBA example and generated a custom REST Connector, so that REST clients could talk to the CORBA server via the REST Connector. This section builds upon that, and shows an OpenAPI model generated from the exposed REST APIs.

If you compile the existing TypeTest IDL file with the `idl2rest` tool, and examine the resulting OpenAPI model, the first thing noticeable is that there are a few missing pieces in the model. The excerpt below identifies the missing sections.

```
openapi: 3.0.1
info: {}
servers: {}
security: []
tags: []
paths:
  /typetest/sysexc_op:
    post:
      summary:
      description:
      operationId: sysexc_op_rest
      responses:
        200:
          $ref: '#/components/responses/OkEmptyResponse'
        default:
          description: Default Response for all exceptions
          content:
            application/json:
              schema:
                type: string
            application/xml:
              schema:
                type: string
```

The highlighted text indicates a number of omissions in the model, such as incomplete **info** and **servers** sections. In the **Paths** section, for the API `/typetest/sysexc_op` there are **summary** and **description** fields which contain no values.

The model is therefore very incomplete. However OpenAPI enables developers to centralize APIs and their documentation in one place. If this model was to be imported into a tool like Swagger-UI, one of two things would occur: either it would fail to import (due to missing information from the **info** and **servers** sections); or the generated web page would look very bare and incomplete due to the lack of API documentation (because of the incomplete **summary** and **description** fields).

OpenAPI Block Comments

To address this shortcoming, the `idl2rest` tool supports a special block comment format that looks very similar to Javadoc style comments. This concept gives users the ability to document the IDL constructs that will be turned into REST APIs. This documentation will carry forward and be populated into the OpenAPI model. For detailed reference information on the OpenAPI block comments, see "Open API Block Comments Reference".

The updated `typetest` IDL file shows these block comments:

```
/***
 * @info-title TypeTest OpenAPI Model example
 * @info-contact-name Micro Focus
 * @info-contact-email support@microfocus.com
 * @info-version 1.0.0
 * @info-license-name Micro Focus CORBA Add-on for Rest Proprietary EULA.
 * @info-license-url file://../../../license.txt
 * @server-url http://localhost:8080
 * @server-desc This REST server provides a REST front-end to
 * our TypeTest CORBA server (insecure instance).
 * @server-url https://localhost:8081
 * @server-desc This REST server provides a REST front-end to
 * our TypeTest CORBA server (secure instance).
 */
module MF_TypeTest
{
[SNIP...]
    @Path(uri = "/typetest", rir = "file:../typetest_objref.txt")
    interface TypeTest
    {
        /***
         * @op-summary The method does nothing, its purpose is to show
         * to a CORBA call.
         * @op-desc This method represents an empty operation that neither
         * sends of recieves any data.
         */
        @Path(uri = "null_op", rir = "")
        @POST
        void null_op();

        /***
         * @op-summary This method demonstrates throwing a System
         * Exception, this of course happens at the CORBA end.
         * @op-desc This method will throw a CORBA System Exception,
         * this will be returned in the HTTP response body, with
         * the appropriate HTTP status code set.
         */
        @Path("sysexc_op")
        @POST
        void sysexc_op();

        [SNIP...]
    };
};
```

The code excerpt above shows the TypeTest IDL file largely unchanged from the previous example, except for the new OpenAPI block comments (which are highlighted in green text).

The code excerpt below is from the generated OpenAPI model:

```
openapi: 3.0.1
info:
  title: TypeTest OpenAPI Model example
  contact:
    name: Micro Focus
    email: support@microfocus.com
  license:
    name: Micro Focus CORBA Add-on for Rest Proprietary EULA.
    url: file://../../../license.txt
  version: 1.0.0
servers:
- url: http://localhost:8080
  description: This REST server provides a REST front-end to our TypeTest
    CORBA server (insecure instance).
- url: https://localhost:8081
  description: This REST server provides a REST front-end to our TypeTest
    CORBA server (secure instance).
security: []
tags: []
paths:
  /typetest/sysexc_op:
    post:
      summary: This method demonstrates throwing a System Exception, this
        of course happens at the CORBA end.
      description: This method will throw a CORBA System Exception, this
        will be returned in the HTTP response body, with the appropriate
        HTTP status code set.
      operationId: sysexc_op_rest
      responses:
        200:
          $ref: '#/components/responses/OkEmptyResponse'
        default:
          description: Default Response for all exceptions
          content:
            application/json:
              schema:
                type: string
            application/xml:
              schema:
                type: string
  /typetest/null_op:
    post:
      summary: The method does nothing, its purpose is to show to a CORBA
        call.
      description: This method represents an empty operation that neither
        sends nor receives any data.
      operationId: null_op_rest
      responses:
        200:
          $ref: '#/components/responses/OkEmptyResponse'
        default:
          description: Default Response for all exceptions
          content:
            application/json:
              schema:
                type: string
            application/xml:
              schema:
```

```
type: string
```

In the excerpt from the OpenAPI model above, the lines in **bold text** demonstrate the metadata that has been filled in from the block comments in the IDL file. Without these the model is very much incomplete, and while some textual descriptions can be left out (such as **summary** and **description** fields), the some other fields in the info and servers sections are a necessity. At the very minimum an **info.title** and a **servers.url** are required.

Generating a Client SDK library from the OpenAPI model

The previous sections demonstrated how the TypeTest IDL file can be used to generate an OpenAPI model representing the REST APIs. This section builds upon that by using the TypeTest OpenAPI model as a blueprint to help ease the development of a REST client. Previously without this approach it was necessary to write a REST client based on the assumed REST APIs that was implemented in REST Connector. This can be a little time consuming, and may require some tweaks to get things right.

This section demonstrates taking the generated OpenAPI model and using the OpenAPI code generator tool (package with the product) to generate a client-side SDK library. This library can then be used by writing a simple client that will make calls against this library. This greatly simplifies the process of writing a REST client to talk to the REST Connector.

In the example that follows, the demonstration shows how to generate a Java client SDK, and from that to write a small Java client that uses this generated SDK library.

Generating a Client SDK for Typetest

This section assumes that:

1 The CORBA Add-on for REST version 1.1 has been successfully installed.

2 Both the ORB and REST environment scripts have been run.

3 The Typetest demonstrations REST Connector and CORBA Server have been deployed.

For the below demo, the code generation tool generates the Client SDK library as Java code with a Maven and Gradle build-system. For the purposes of this demonstration, the scripts included with the product are designed to integrate with the Maven build-system. For information on installing and configuring Maven 3 please see the Maven website http://maven.apache.org.

Building the Java Client and the generated Client SDK library

Orbix 6

The Orbix 6 demo is run with the `itant` tool that is distributed with Orbix 6.

Windows:

```
cd %REST_HOME%\demos\typetest\openapi\java_client
```

```
itant
```

UNIX:

```
cd $REST_HOME/demos/typetest/openapi/java_client
itant
```

VisiBroker

The VisiBroker demonstration follows the demonstration build system that VisiBroker demos use, which is a Makefile for UNIX platforms, and a batch file script on Windows systems.

Windows:

```
cd %REST_HOME%\demos\typetest\openapi\java_client
vbmake.bat
```

UNIX:

```
cd $REST_HOME/demos/typetest/openapi/java_client
make all
```

Looking at the Java Client code

What has been generated so far is a Java library component that neatly defines the necessary types and APIs that a developer can write a Java client against. This makes writing the Java client a very straightforward process.

```
package com.microfocus.rest4corba.typetest;
[SNIP...]
import com.microfocus.rest4corba.ApiClient;
import com.microfocus.rest4corba.ApiException;
import com.microfocus.rest4corba.Configuration;
import com.microfocus.rest4corba.DefaultApi;
[SNIP...]
import com.microfocus.rest4corba.model.LongMultiNoPathParamParamWrapper;
import com.microfocus.rest4corba.model.LongMultiParamWrapper;
[SNIP...]

public class Client {
  [SNIP...]
  private ApiClient client;
  private DefaultApi apiInstance;
  [SNIP...]

  public Client() {
    [SNIP...]
    client = Configuration.getDefaultApiClient();
    apiInstance = new DefaultApi(client);
    [SNIP...]
  }

  public void runApis(String mediaType) {
    [SNIP...]
    try {
        [SNIP...]
        System.out.println(
            apiInstance.longMultiNoPathParamRest(
                new LongMultiNoPathParamParamWrapper().valInout(10)
                   .valInout2(2).valInout3(13)
                   .valOut(0).valOut2(0),
                      contentTypeHeader, acceptHeader));

        System.out.println(apiInstance.longMultiPathParamRest(1, 1,
           1, contentTypeHeader, acceptHeader));

        System.out.println(
            apiInstance.longMultiRest(
                1, 1,
                new LongMultiParamWrapper().valIn(1).valInout(10)
                  .valOut(0).valInout2(2).valInout3(13).valOut2(0),
                    contentTypeHeader, acceptHeader));

         [SNIP...]
      } catch (ApiException e) {
        throw e;
      } catch (Exception e) {
        e.printStackTrace();
        throw e;
      }
    }

    public static void main(String[] args) {

      Client client = new Client(args);
      try {
        client.runApis("application/json");
        client.runApis("application/xml");
      } catch (ApiException e) {
        System.err.println("Exception when calling DefaultApi: " + e);
        System.err.println("Status code: " + e.getCode());
        System.err.println("Reason: " + e.getResponseBody());
        System.err.println("Response headers: " + e.getResponseHeaders());
        e.printStackTrace();
        System.exit(-1);
      } catch (Exception e) {
        System.err.println("Exception caught: " + e);
        System.exit(-1);
      }

      System.exit(0);
    }

  }
```

The above example shows how straightforward it is to make REST calls by using the Client SDK library, as follows:

- The library must be initialized by obtaining a reference to the Configuration object, and passing it into the constructor of an instantiated DefaultAPI object.
- The next step is to call the REST API on the DefaultAPI object.
- The library abstracts away the other aspects of writing a REST Client, such as dealing with the API's URI, and the HTTP method type (such as

GET, POST, PUT, or DELETE).   All this information has being encoded in the OpenAPI model, and the user of the Client SDK has no need to concern themselves with these details.

After the library has being initialized, calling any REST API typically follows these steps:

1 All the API names end with `Rest`, so the IDL operation `long_multi` is mapped to the `longMultiRest` method on the DefaultAPI object.
2 For some of the methods there will be a `<`*`type`*`>ParamWrapper` class required. This can be seen in the `longMultiRest` method, where the `longMultiParamWrapper` needs to be constructed with certain data values. These values correspond to the IDL parameter values that will be sent to the CORBA server.
3 The method will return an instance of `<`*`type`*`>ResponseWrapper`, which typically contains a return value, and any of the parameters that were defined with `inout` or `out` parameter direction modes. In the demonstration included in the product, there are no instances of `<`*`type`*`>ResponseWrapper` classes created; instead the resulting `toString()` method is invoked as the originating `defaultApi.longMultiRest` is called from a `System.out.println()` call.

For more details on writing java clients against the Client SDK, see the source code of the DefaultAPI class located at:

Windows:

```
%REST_HOME%\demos\typetest\openapi\java_client\
clientsdk_output\src\main\java\com\microfocus\rest4corba\
DefaultAPI.java
```

UNIX:

```
$REST_HOME/demos/typetest/openapi/java_client/
clientsdk_output/src/main/java/com/microfocus/rest4corba/
DefaultAPI.java
```

Running the Java Client and the generated Client SDK library

Once the Java client has been built, it can be run as follows:

Windows:

```
cd %REST_HOME%\demos\typetest\openapi\java_client
runclient.bat --hostname %COMPUTERNAME%
```

UNIX:

```
cd $REST_HOME\demos\typetest\openapi\java_client
runclient.bat --hostname $HOSTNAME
```

The snippet below shows some output from the client:

```
Running Client...
   Your current IP address : <IP address>
   Your current Hostname : <hostname>
```

```
baseUrl is http:\\<hostname>:8080
Status code: 409
Reason: null
Status code: 414
Reason: Our customer User Exception...
class LongInoutResponseWrapper {
    val: 3951034
}
class LongOutResponseWrapper {
    val: 91380352
}
class LongReturnResponseWrapper {
    ret: 1945782043
}
....
```

Deploying the OpenAPI model in Swagger-UI

This section looks at deploying the generated OpenAPI model in the Swagger-UI tool.

Swagger-UI is a freely available tool that allows users to deploy OpenAPI models in a web application. Swagger-UI is a set of javascript based frameworks that can take any OpenAPI document (in either JSON or Yaml format), and render it using modern visual representations that can be deployed in most modern browsers.

For information on Swagger-UI, see https://swagger.io/tools/swagger-ui/.

Using Swagger-UI it is possible for a user to document and visualize the REST APIs described in the OpenAPI model, and to examine all aspects of the APIs and the data-types that are sent and received through these APIs. In addition any API can be tried out against the relevant REST server instance.

The version of Swagger-UI that is packaged in CORBA Add-on for REST v.1.1 is a Micro Focus branded Swagger-UI. This section discusses how to deploy the TypeTest OpenAPI model in this Swagger-UI instance. The same instructions will hold true for deploying an OpenAPI model in a non-branded version of Swagger-UI.

Deployment Procedures

There are several ways that a Swagger-UI instance can be deployed. This section looks at two deployments:

- Using NodeJS to deploy Swagger-UI.
- Using Python’s `Http.server` module to deploy Swagger-UI.

Deploying Swagger-UI using NodeJS

As a prerequisite for deploying Swagger-UI using NodeJS, you must:

- Install and configure a recent version of NodeJS. NodeJS is freely available from https://nodejs.org.
- Set the NodeJS environment, so that you can run the `npm` and `node` commands.
- Run the following installation commands, using the `npm` tool which is part of the NodeJS installation:

- `npm install express`
- `npm install https://github.com/MicroFocus/swagger-ui/archive/v1.0.0-dist.tar.gz`

These two commands install into the NodeJS installation two packages that are required to continue with this demonstration.

Once NodeJS has been configured and installed, the following steps are required to deploy the TypeTest model in Swagger-UI.

Windows:

```
cd %REST_HOME%\demos\typetest\openapi\swagger_ui
node mf_swagger_ui.js
```

UNIX:

```
cd $REST_HOME/demos/typetest/openapi/swagger_ui
node mf_swagger_ui.js
```

This produces the following output:

```
Serving Swagger-UI at: http://<hostname>:9000
```

Use a web browser to go to the URL displayed in the output in order to see the Swagger-UI web page.

Deploying Swagger-UI using Python

The only prerequisite in this section is that Python3 is installed and configured.

Windows:

```
$ cd %REST_HOME%\demos\typetest\openapi\swagger_ui
$ %REST_HOME%\bin\deploy_swagger_ui.bat %REST_HOME%\demos\
typetest\idl\typetest.yaml
```

UNIX:

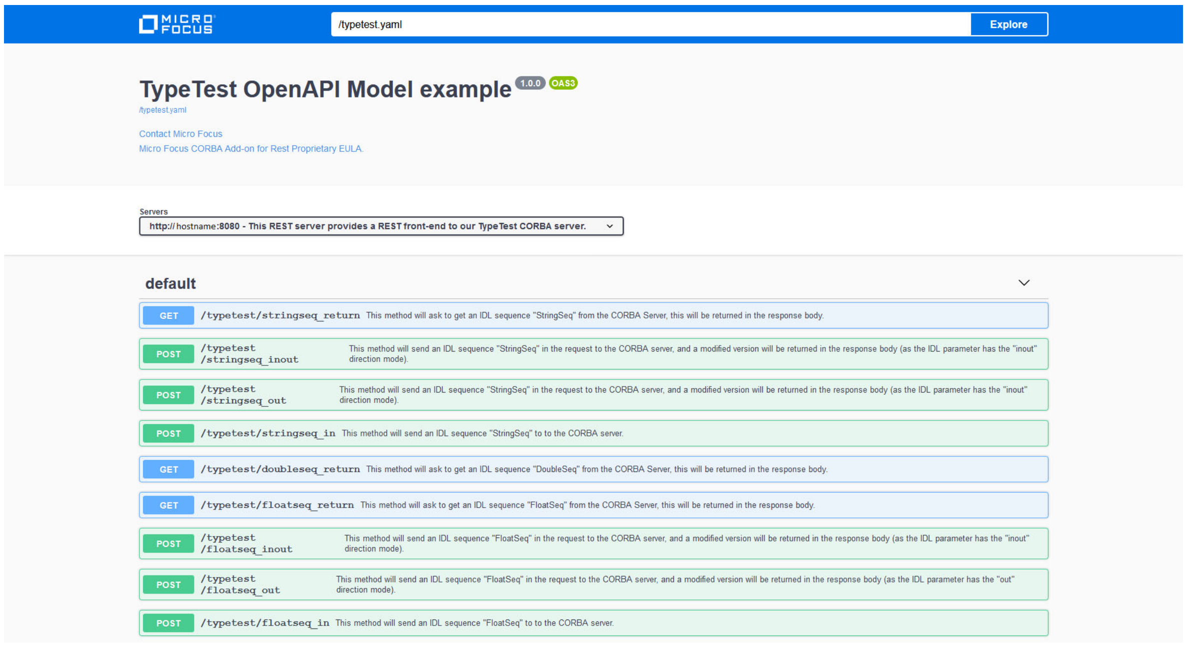
```
> cd $REST_HOME/demos/typetest/openapi/swagger_ui
> $REST_HOME/bin/deploy_swagger_ui.sh $REST_HOME/demos/
typetest/idl/typetest.yaml
```

This will generate output similar to the following:

```
Deploying Swagger-UI into D:\orb_install\REST\demos\typetest\openapi\
swagger_ui\swagger_ui_dist for standalone deployment
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\absolute-path.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\favicon-
16x16.ico
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\index.html
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\index.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\
mf_logo_white.svg
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\microfocus-
config.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\microfocus-
swagger-ui.css
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\oauth2-
redirect.html
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\package.json
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\README.md
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui-
bundle.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui-
bundle.js.map
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui-
standalone-preset.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui-
standalone-preset.js.map
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui.css
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-
ui.css.map
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-ui.js
META-INF\resources\webjars\microfocus-swagger-ui-dist\1.0.0\swagger-
ui.js.map
       18 file(s) copied.
        1 file(s) copied.
        1 file(s) moved.
Now running Python3's Http Server to serve Swagger-UI (CTRL+C) to quit
Serving HTTP on 0.0.0.0 port 9000 (http://0.0.0.0:9000/) ...
```

The Swagger-UI web page

You can now view the Swagger-UI web page by going to the URL `http://<hostname>:9000/` in a web browser.



TypeTest OpenAPI Model example

/typetest.yaml

Contact Micro Focus

Micro Focus CORBA Add-on for Rest Proprietary EULA.

Servers

http://hostname:8080 - This REST server provides a REST front-end to our TypeTest CORBA server.

default

GET `/typetest/stringseq_return` This method will ask to get an IDL sequence "StringSeq" from the CORBA Server, this will be returned in the response body.

POST `/typetest/stringseq_inout` This method will send an IDL sequence "StringSeq" in the request to the CORBA server, and a modified version will be returned in the response body (as the IDL parameter has the "inout" direction mode).

POST `/typetest/stringseq_out` This method will send an IDL sequence "StringSeq" in the request to the CORBA server, and a modified version will be returned in the response body (as the IDL parameter has the "inout" direction mode).

POST `/typetest/stringseq_in` This method will send an IDL sequence "StringSeq" to to the CORBA server.

GET `/typetest/doubleseq_return` This method will ask to get an IDL sequence "DoubleSeq" from the CORBA Server, this will be returned in the response body.

GET `/typetest/floatseq_return` This method will ask to get an IDL sequence "FloatSeq" from the CORBA Server, this will be returned in the response body.

POST `/typetest/floatseq_inout` This method will send an IDL sequence "FloatSeq" in the request to the CORBA server, and a modified version will be returned in the response body (as the IDL parameter has the "inout" direction mode).

POST `/typetest/floatseq_out` This method will send an IDL sequence "FloatSeq" in the request to the CORBA server, and a modified version will be returned in the response body (as the IDL parameter has the "out" direction mode).

POST `/typetest/floatseq_in` This method will send an IDL sequence "FloatSeq" to to the CORBA server.

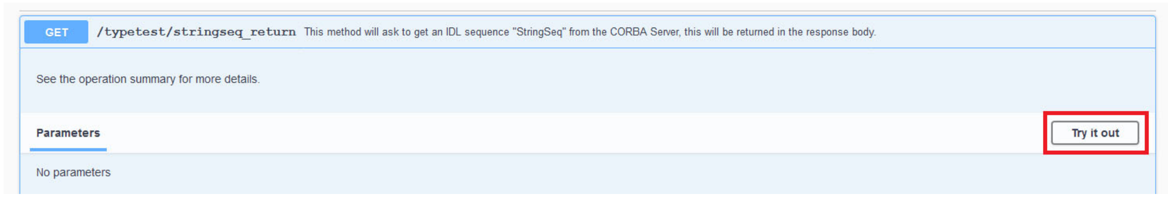
The screen illustration above shows the typical view in a web browser once the Typetest OpenAPI model has been deployed.

The OpenAPI block comments that were explored in the "OpenAPI Block Comments" section come into play here. Each of the REST APIs shown on the Swagger-UI webpage contains some of the documentation in the form of OpenAPI block comments, demonstrating how to use the IDL file to ensure that the generated REST API is fully documented.

Any of the operations on the Swagger-UI web page can be explored further by clicking on the REST API.

This section looks at exploring one of the REST API's methods, `/typetest/stringseq_ret`, and using Swagger-UI to make a test REST API call to the REST connector.

The following illustration shows how the web page displays the `/typetest/stringseq_ret` method.



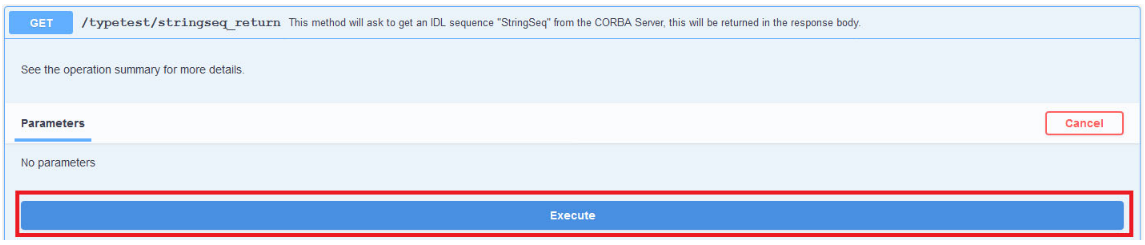
GET `/typetest/stringseq_return` This method will ask to get an IDL sequence "StringSeq" from the CORBA Server, this will be returned in the response body.

See the operation summary for more details.

Parameters

No parameters

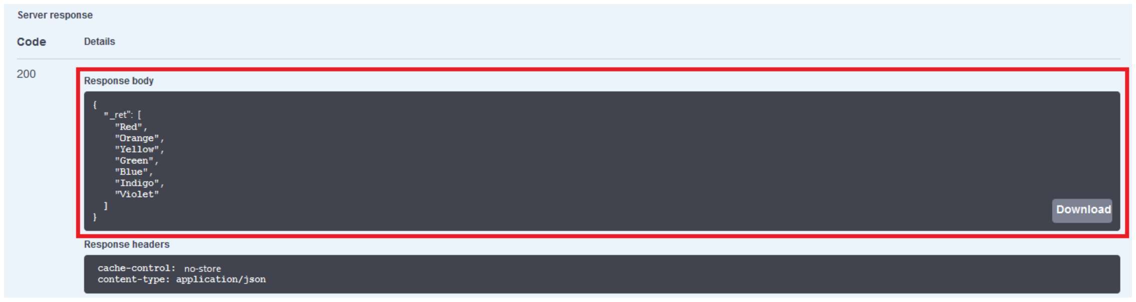
Clicking the **Try it out** button (highlighted in red). This displays another button, titled **Execute**.



GET /typetest/stringseq_return This method will ask to get an IDL sequence "StringSeq" from the CORBA Server, this will be returned in the response body.

See the operation summary for more details.

When you click the **Execute** button (again highlighted in red), another section is displayed showing the REST API response from the REST connector.



```
{
  "_ret": [
    "Red",
    "Orange",
    "Yellow",
    "Green",
    "Blue",
    "Indigo",
    "Violet"
  ]
}
```

```
cache-control: no-store
content-type: application/json
```

In the illustration above the HTTP response from the REST connector is shown highlighted in red. The response contains the JSON representation of the data returned from the IDL operation via the REST connector.

System Exceptions

This chapter describes system exception mapping in the CORBA Add-on for REST.

The CORBA Add-on for REST product provides a means to map any CORBA System Exceptions raised during the invocation or processing of a CORBA call to a HTTP response code. This gives a seamless way for REST developers to handle any error conditions, in a more HTTP-friendly manner, without needing to understand CORBA System Exceptions.

The table below shows the System Exceptions that the CORBA Add-on for REST currently maps. Further System Exception mappings may be added in a future release.

CORBA System Exception	HTTP Response Code
COMM_FAILURE and TIMEOUT	TIMEOUT (408)
OBJECT_NOT_EXIST and INV_OBJREF	GONE (410)
TRANSIENT	NOT_FOUND (404)
NO_PERMISSION	UNAUTHORISED (401)
BAD_OPERATION and BAD_PARAM	METHOD_NOT_ALLOWED (405)
MARSHAL	BAD_REQUEST (400)
INTERNAL and INITIALIZE	INTERNAL_SERVER_ERROR (500)
NO_IMPLEMENT	NOT_IMPLEMENTED (501)
IMP_LIMIT, NO_MEMORY, and NO_RESOURCES	SERVICE_UNAVAILABLE (503)
All other System Exceptions	CONFLICT (409)

Configuration

This chapter describes configuration properties for the CORBA Add-on for REST.

The CORBA Add-on for REST product typically requires a configuration file called `connector.properties`. This file is generated when you run the **idl2rest** tool to create the REST connector. The file is placed in the `generated_config` sub-folder of the current directory.

The table below shows the configuration variables that are currently supported in the product.

Property Name	Description
REST_CONNECTOR_HOSTNAME	The hostname that the Connector's HTTP server uses to listen for incoming HTTP messages
REST_CONNECTOR_PORT	The port number that the Connector's HTTP server uses to listen for incoming HTTP messages.
REST_CONNECTOR_SECURE	Whether or not the Connector uses security. The default is `false`. If security is enabled the variables `REST_KEYSTORE_*` and `REST_TRUSTSTORE_*` must be configured to point at the TLS keystore and truststores for the Connector to use.
REST_KEYSTORE_SERVER_FILE	The Java keystore file to be used when the REST Connector is deployed securely.
REST_KEYSORE_SERVER_PWD	The passphrase for the server's keystore.
REST_TRUSTSTORE_SERVER_FILE	The Java truststore file to be used when the REST Connector is deployed securely.
REST_TRUSTSTORE_SERVER_PWD	The passphrase for the server's truststore.
REST_RESOURCE_PACKAGE	The package to scan for generated REST resource classes.
REST_ADD_ON_CLASS	The fully qualified classname to look for to load a custom add-on class. This is a class that extends from the `com.microfocus.rest4corba.ext.CorbaAddOn` abstract class and that can be extended for custom operations, such as `getObjectKey()`. This configuration item would typically be used when the product is installed on top of another ORB product.
REST_CONNECTOR_ORB_ADAPTER_CLASS	The fully qualified class name to look for to load a custom ORB adapter add-on class. This is a class that extends from the `com.microfocus.rest4corba.ext.CorbaAddOn` abstract class, which can be extended for custom operations. See "Extending the ORB Adapter Class" for more information on extending the class. This configuration item would typically be used when the product is installed on top of another ORB product.

Property Name	Description
REST_CACHE_HEADER_VALUE	HTTP response caching is a feature in HTTP whereby the browser, or a HTTP intermediary between the REST Connector and a REST client, may cache the HTTP response data. By default the REST Connector sets the HTTP response header `Cache-Control` to the value `no-store` to ensure that the HTTP response is never cached.
REST_CORS_HEADER_ALLOW_ ORIGIN	This configuration variable specifies the value to assign to the HTTP response header `Access-Control-Allow-Origin`. This HTTP response header is used to indicate whether the HTTP response can be shared with the client that issued the corresponding HTTP request from the given origin. For more information on this and on CORS Headers, see the section "HTTP Cross Origin Resource Sharing support". By default the CORS HTTP response header `Access-Control-Allow-Origin` is set to the value `*`.
REST_CORS_HEADER_ALLOW_ HEADERS	This configuration variable specifies the value to assign to the HTTP response header `Access-Control-Allow-Headers`. This HTTP response header is used to respond to a pre-flight request (typically a HTTP OPTIONS request), to indicate which HTTP request headers are acceptable during the subsequent HTTP request. For more information on this and on CORS Headers, see the section "HTTP Cross Origin Resource Sharing support". By default the CORS HTTP response header `Access-Control-Allow-Headers` is set to the value `origin, content-type, accept, authorization`.
REST_CORS_HEADER_ALLOW_ METHODS	This configuration variable specifies the value to assign to the HTTP response header `Access-Control-Allow-Methods`. This HTTP response header is used to respond to a pre-flight request (typically a HTTP OPTIONS request), to indicate which HTTP method designators (such as OPTIONS, GET, PUT, POST, and DELETE) are acceptable during the subsequent HTTP request. For more information on this and on CORS Headers, see the section "HTTP Cross Origin Resource Sharing support". By default the CORS HTTP response header `Access-Control-Allow-Methods` is set to the value `POST, PUT, GET, OPTIONS, DELETE`.

Configuring Multiple Connectors

The most straightforward method of configuring multiple connectors is to maintain separate copies of the `connector.properties` file, and pass in the `--config-file` command line option to each of the Connectors.

As is detailed in the next section, it is also possible to configure the connector in a variety of ways, such as with command line options and environment variables, or even a mixture of the two approaches. That being said, Micro Focus recommends having the configuration in a single configuration file, as this gives a clearer insight into the exact configuration of the Connector.

Overriding the Configuration Variables

By default, configuration variables are configured by specifying them in the `connector.properties` file that is generated from running the **idl2rest** tool. Any of the variables in this file can be overridden by specifying an environment variable of the same name.

A configuration variable is read and set as follows:

- The value is read from the configuration file, if there is no environment variable set with the same name. If there is an environment variable set with the same name, then this value is used.
- If a command line option is set, then this takes precedence over the previous setting whether the variable existed in the configuration file or as an environment variable.

Extending the ORB Adapter Class

The CORBA Add-on for REST product provides an extension mechanism where different CORBA implementations may have a proprietary means of discovering certain information. For example this can be used if you wanted to leverage the use of a CORBA Objects object key, which is a useful piece of information to identify a unique CORBA Object.

However, starting with CORBA Add-on for REST v1.1 this proprietary lookup mechanism for an ORB’s object key is no longer required. The product will find the underlying object key using OMG-compliant APIs, a method that is ORB vendor agnostic.

The following excerpt shows the code listing for an abstract Java class that is provided by the product. By extending this abstract class, and implementing the `getObjectKey()` Java method, it is possible to have the product use this method to allow URIs to leverage the use of the object key to uniquely identify the CORBA Object.

```
package com.microfocus.rest4corba.ext;

public abstract class CorbaAddOn {

  public org.omg.CORBA.Object string_to_object(String ref)
{
    return null;
  }

  public String
getObjectKey(org.omg.CORBA.portable.ObjectImpl obj) {
      return null;
    }
```

The following is an example of implementing this class:

```
  import com.microfocus.rest4corba.ext.CorbaAddOn;

  public class RESTOrbAdapter extends CorbaAddOn {

    @Override
    public String
  getObjectKey(org.omg.CORBA.portable.ObjectImpl obj) {
       byte[] objectKeyBytes = null;
       // … Retrieve the object key via the ORBs underlying
  proprietary APIs
       objectKeyBytes = <objkey-bytes>
       return new String(objectKeyBytes);
    }

  }
```

You must then ensure that the class is compiled and present on the Connectors classpath. When the connector runs and a URI is present with the placeholder `{objkey}`, the Connector will call the `RESTOrbAdapter.getObjectKey()` method to retrieve the unique value for the CORBA Object.

Open API Block Comments Reference

This section describes the Open API Block Comments facility referred to in "Open API Support".

Each block comment follows this format:

- The comment starts with the characters `/***`. Note the 3 asterisks. This ensures that the block comments are not confused with the Javadoc style of block comments that start with a forward slash followed by two asterisks.
- The format inside the comment contains a series of tags and values. Each tag defines the type of data that follows it. A tag is identified by a prepended `@` character (for example: `@info-title`). The text that follows must be separated by a space from the end of the prefixed tag. The text is then inserted into the OpenAPI model, at the place identified by the tag.
- The block comment ends with the characters `*/`.

The tags have been named to be as self-describing as possible. For example: `@info-contact-email` would map to the `info.contact.email` entry in the model, and `@op.desc` would map to `current-path-item.description` (where `current-path-item` means the current path-item that represents a CORBA IDL operation/attribute).

The following table shows the full reference of the block comment tags that are possible:

OpenAPI Tag name	Description
`info-title`	A title to give the OpenAPI model.
`info-version`	A version number for the model.
`info-contact-name`	An author's or other name that can be contacted.
`info-contact-email`	An email address for the contact name.
`info-license-name`	A name for the software license for the model.
`info-license-url`	A URL where the license can be obtained from.
`server-url`	The URL of the server that implements or serves out this REST API.
`server-desc`	A description of the server.
`op-summary`	A brief summary (typically 1-3 lines) of the current REST API.
`op-desc`	A longer description of the current REST API.

idl2rest Options

The table below shows the command line flags that the **idl2rest** tool will accept.

Option	Usage
-D, -define *foo*[=*bar*]	Define a preprocessor macro, optionally with value.
-I, -include <*dir*>	Specify an additional directory for #include searching.
-P, -no_line_directives	Do not emit #line directives from preprocessor. Defaults to off.
-H, -list_includes	Display #included file names as they are encountered. Defaults to off.
-C, -retain_comments	Retain comments in preprocessed output. Defaults to off.
-U, -undefine *foo*	Undefine a preprocessor macro
-[no_]idl_strict	Strict OMG-standard interpretation of IDL source. Defaults to off.
-[no_]builtin (TypeCode|Principal)	Create built-in type "::TypeCode" or "::Principal". Defaults to on.
-[no_]warn_unrecognized_pragmas	Warn if a #pragma is not recognized. Defaults to on.
-[no_]back_compat_mapping	Use mapping that is compatible with VisiBroker 3.x. Defaults to off.
-[no_]preprocess	Preprocess the input file before parsing. Defaults to on.
-[no_]warn_all	Turn all warnings on/off simultaneously. Defaults to off.
-dump_tree	Dump the IDL Parse Tree (Front End)
-root_dir <*path*>	Directory in which generated files should reside.
-package <*pkg*>	Specify a root package for generated code.
-[no_]compile	Compile any Java file written automatically. Defaults to off.
-[no_]gen_config	Generate a usable Connector configuration file called `connector.properties` in the current folder. Defaults to on.
-[no_]gen_v1_wrappers	Whether to generate a REST connector that is interoperable with REST clients written against v1.0 of the product (see “Wrapped Objects” for more details). Defaults to off.
-version	Display software version numbers.

Connector Options

The table below shows the command line flags that the `rest_connector` Java class will accept.

Option	Usage
--hostname	The hostname where the Connector should listen for HTTP requests.
--port	The port where the Connector should listen for HTTP requests.
--secure	Whether the Connector should listen securely for HTTP requests on an `https` URL.
--rest-package	The Java package that should be scanned for **idl2rest**-generated classes.
--config-file	The path to a configuration file with which the connector should run with. A default file can be generated by running: `idl2rest.<bat|sh> -gen_config`

The `rest_connector` script in addition takes the following option.

Option	Usage
-verbose	Runs the connector with tracing or more logging verbosity.

Index