CA Gen Integration

Consuming REST Services – Java Edition

Christian Kersters

Broadcom Limited

Web: www.broadcom.com

Corporate Headquarters: San Jose, CA

Revision History

Revision	Date	Change Description
v1.0	2020/12/18	Initial version

References

CA Gen Integration Solutions, Christian Kersters, Broadcom, August 2017
https://community.broadcom.com/HigherLogic/System/DownloadDocumentFile.ashx?DocumentFileKey=4a401797-4dfe-4230-a031-273b908e57d3&forceDialog=0

Richardson Maturity Model
https://restfulapi.net/richardson-maturity-model/

Contents

1 Introduction

Over the last decade, REST has gained much momentum, compared to the older SOAP protocol, as a solution to exchange messages and integrate workflows among different, independent parties across the Internet.

Reasons for that enthusiasm for REST are multiple, the most important being:

- **Extensive use of the HTTP protocol**, fostering reuse of hardware and software assets across human-based and machine-based consumption (caching, authentication and authorization, …), and making it lightweight
- **Support for multiple data formats**, with most, if not all REST server frameworks supporting both XML and the less verbose JSON formats
- **Flexibility, simplicity and extensibility of APIs**, easing exchange of structured data and code reuse
- **Statelessness**, making it easy, among others, to develop test harness suites.

Due to this success, many solutions have been developed to assist with REST services publication or consumption, or to extend SOAP-based frameworks to also support REST.

Thanks to their reliance on the HTTP protocol, REST services are very easy to consume, even without such specialized framework. Many options are available, in your preferred language, to send HTTP GET, POST, … requests. Also, certainly when the structure of the messages you exchange remains simple, they can easily be created or decoded using a small set of string manipulation and domain conversion functions. When it's not the case, and the structure requires more work, complexity of the task will be significantly reduced by relying on XML or JSON libraries, many of which available from the Jakarta Java EE platform or as Open Source. *If the services you want to consume are quite independent, in their function or interface[1], this is probably the best approach*.

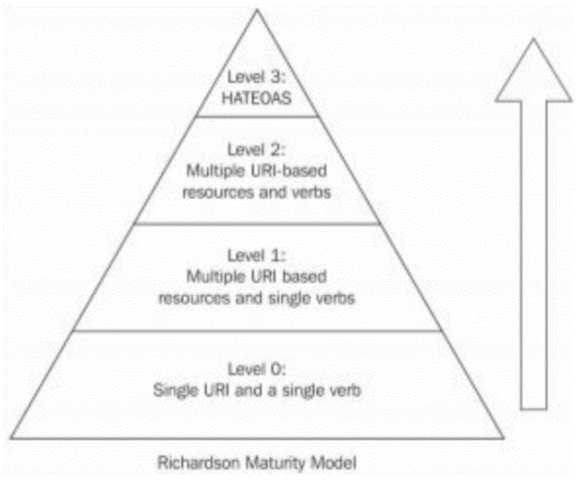
If, on the contrary, the services you want to consume provide a consistent Web API, the REST consumption frameworks provide better alternatives. In addition to give the functions to support all necessary HTTP features, **those solutions, typically, recreate the API data model, based on its formal documentation**. This data model is then populated / queried by your specific consumption logic, using generated functions and fields, and automatically serialized to / de-serialized from the selected message format (typically XML or JSON).

For the Java language, this is the case with the **Apache CXF Services framework**. Apache CXF makes consumption of REST services easy in CA Gen applications implemented in Java, even without in-depth knowledge of the language and the HTTP protocol, and it's the objective of this document to guide and help you to do this.

At Broadcom Mainframe Services for CA Gen, we've extensively used Apache CXF to consume REST services from CA Gen C external action blocks. Our best practices will be presented here.

[1] Like SOAP web services converted to REST

1.1 Richardson Maturity Model


Level 3:
HATEOAS
Level 2:
Multiple URI-based resources and verbs
Level 1:
Multiple URI based resources and single verbs
Level 0:
Single URI and a single verb
Richardson Maturity Model

Leonard Richardson analyzed a hundred different web service designs and divided them into four categories based on how much they are REST compliant.

In the rest of this document, Levels 0-1 types of REST services will be called ***Ad-Hoc Services***[2], where levels 2-3, thanks to their consistency, truly are ***Web APIs***[3].

While the technical part of this document can apply equally well to levels 1-3, the methodological view of our best practices are much more applicable when consuming Web APIs.

1.2 Specification of REST services and data in Level 2-3 APIs

1.2.1 REST service identification

In Web APIs, target REST services are identified by the following 3 components:

HTTP Verb – URL – MIME Type

1.2.1.1 HTTP verb

The standard HTTP verbs are normally used to specify the type of action the Service provides:

Verb	Meaning
GET	Read
POST	Create
PUT	Update
DELETE	[Logical] Delete

[2] Level 0 should really be considered as "*XML/JSON Services*", rather than REST, as it's only the content of the message that drives the process
[3] For information, Broadcom Mainframe Services' ***Web API Designer*** Field-Supported Solution can generate Level 1 Ad-hoc Services and Level 2 (and Level 3, using Jboss RESTEasy REST framework) Web APIs

HTTP considerations can however influence this clear setup. The most important consideration is that GET requests don’t support any message payload (nor should the DELETE ones).

(*At Broadcom Mainframe Services for CA Gen, we avoid as much as possible designs where Read requests require significant / structured input. However, when the impact of this constraint would be too high – like performances or workload – we use POSTs instead, and clearly document the case*).

1.2.1.2 MIME type

There are 2 optional MIME types associated with a request, which are specified in header parameters

- ***Content-Type***, which describes the format of the body that is sent to the service
- ***Accept***, which describes the format of body the consumer expects in the response.

For REST, wherever relevant, both contain ***xml*** or ***json***, and are most generally ***application/xml*** or ***application/json***.

As such, they don’t influence the service that is invoked by the HTTP verb and the URL.

Some REST server frameworks, however, make it possible to invoke different services, based on the MIME type. This possibility could be used for service versioning. Such custom MIME types should normally start with “vnd.” (for vendor-specific). For instance, vnd.com.broadcom.mf.gen.cse.v01+xml *as Content-Type would mean that we want to access version 1 of a service with a request payload in XML.*

1.2.1.3 URL

In Web APIs, URLs always start with the same base content:

```
      http[s]://<host>[:<port>]/<base url>
```

Next part is definition of the target ***resource***[4], with optional ***id*** and ***action***:

```
            {/<resource>[/id]}[/action]
```

If no id is specified, the action (or default HTTP verb behavior) applies to the type of resource (like listing for GET or creation for POST), and it applies to the specific resource if the id has been specified (normally defaulting to read for GET, update for PUT or delete for DELETE, as mentioned before).

Based on the identified resource (between <id> and <action>), sub-resources can be accessed, at arbitrary depth (as represented by the “{}”), as long as previous resources have been identified (accompanied by one id).

[4] A resource is similar to an object, containing fields and accessed through a number of methods (which are its REST services)

1.2.2 Exchanging data with Level 2-3 APIs

1.2.2.1 Sending data

In addition to resource identifiers specified in the URL (path parameters), consumers of APIs can also send:

- ***Query parameters*** (*?parm1=xxx&parm2=yyy*)
- ***Header fields*** (***Authorization***, custom fields)
- Request ***payload*** (except for GET/DELETE requests, as mentioned before).

1.2.2.2 Receiving data

REST services will also (normally) send information back to the consumer, as:

- ***HTTP Status code***
- Custom ***Header fields***
- Response ***Payload***.

1.3 Level 3 Web APIs and HATEOAS

HATEOAS stands for ***Hypertext As The Engine Of Application State***. This means that the REST service will send back hyperlinks to the resources it specifies in its response, to ease navigation through the API. Such information makes it very easy for consumers to fetch resource details or related information, based on an initial request.

2 API specifications

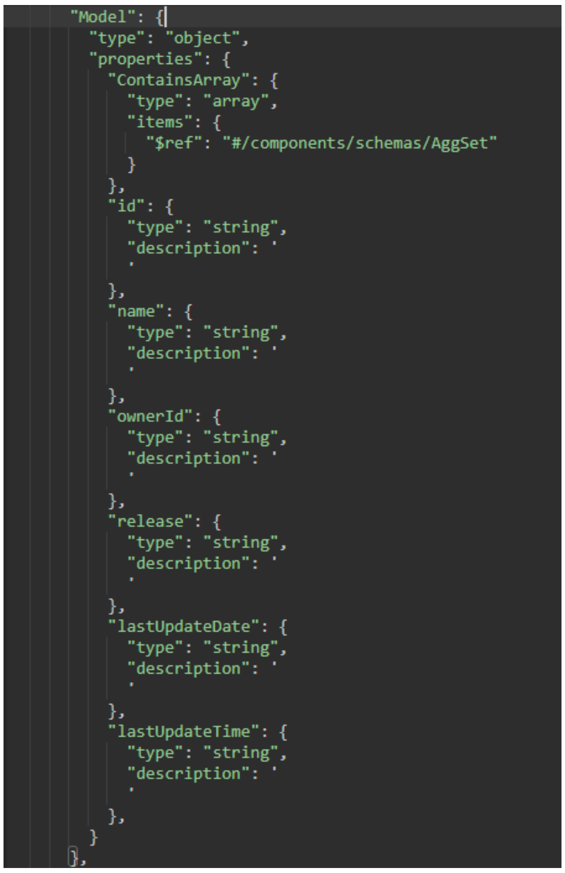
As opposed to SOAP, with its unique WSDL specification format, REST supports multiple formal representations, among others:

- ***Swagger***, or its descendant, ***OpenAPI*** (aka Swagger 3) (the most widespread), available in JSON or YAML format
- ***RAML***, YAML-based
- ***WADL***, XML-based, REST equivalent to the WSDL for SOAP, with focus on machine readability.

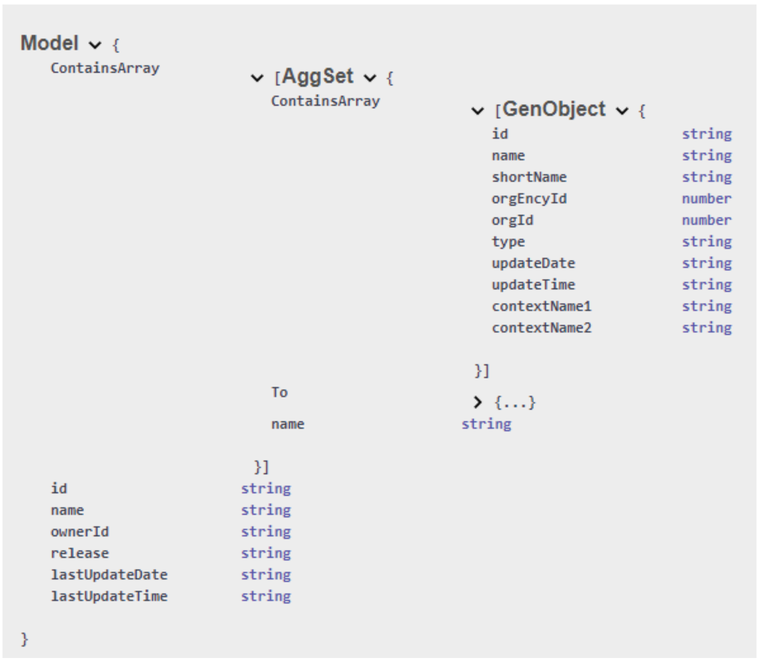
2.1 OpenAPI specification

Here are some examples of an OpenAPI specification of a REST API, in native format (JSON in this case) or human representation.

2.1.1 Data model

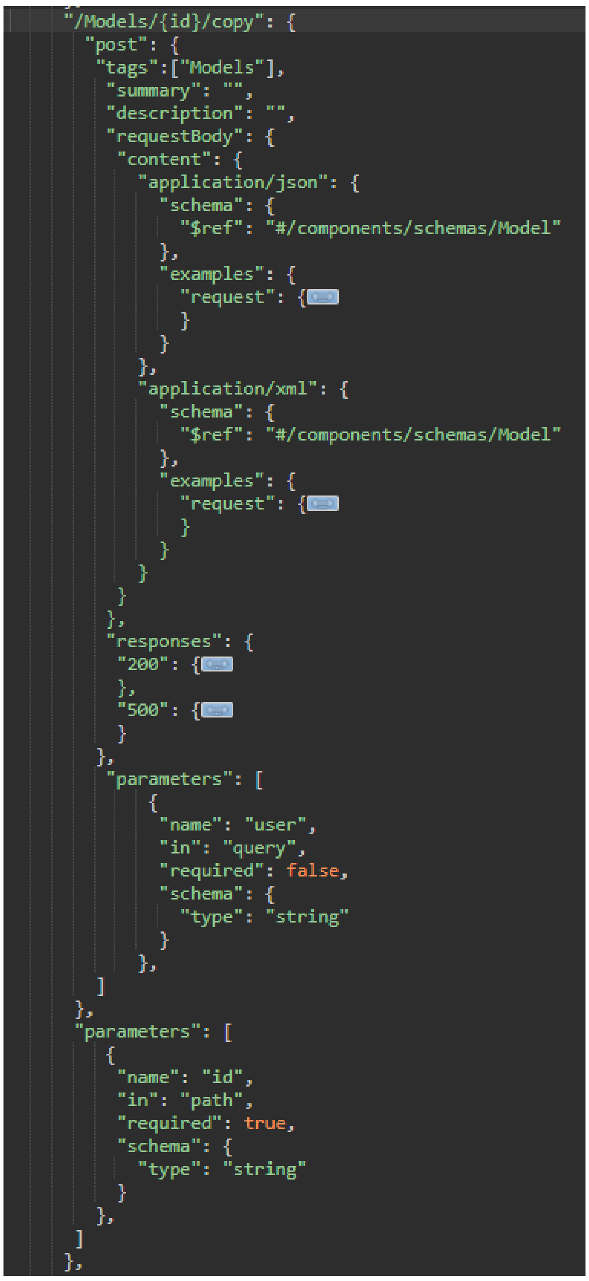


```
"Model": {
  "type": "object",
  "properties": {
    "ContainsArray": {
      "type": "array",
      "items": {
        "$ref": "#/components/schemas/AggSet"
      }
    },
    "id": {
      "type": "string",
      "description": '
      '
    },
    "name": {
      "type": "string",
      "description": '
      '
    },
    "ownerId": {
      "type": "string",
      "description": '
      '
    },
    "release": {
      "type": "string",
      "description": '
      '
    },
    "lastUpdateDate": {
      "type": "string",
      "description": '
      '
    },
    "lastUpdateTime": {
      "type": "string",
      "description": '
      '
    },
  }
},
```

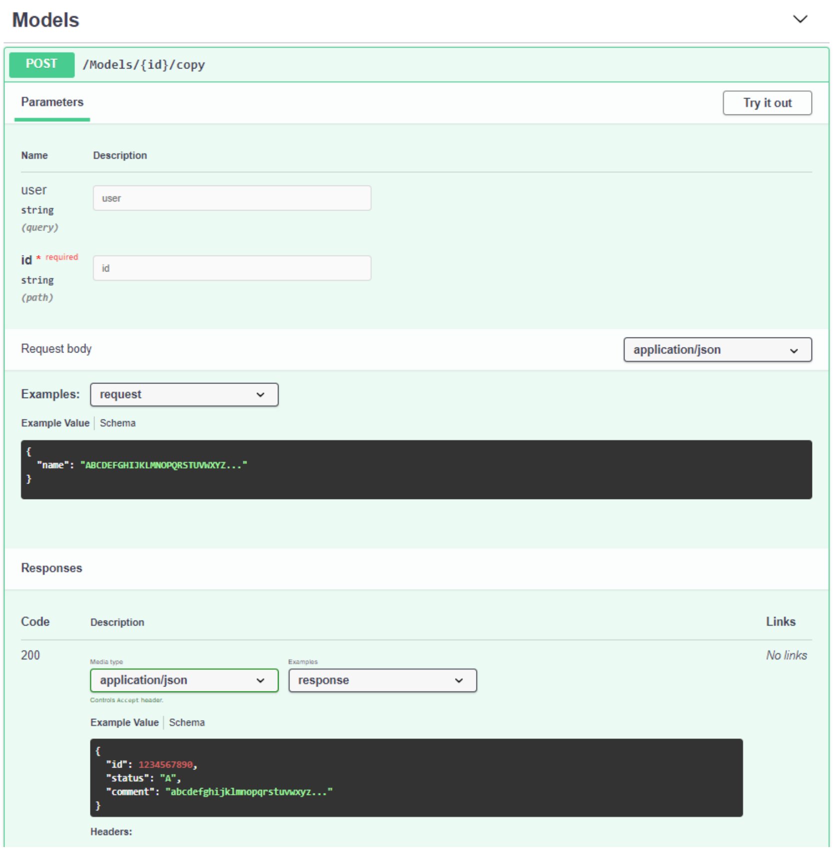


```
Model ⌄ {
   ContainsArray        ⌄ [AggSet ⌄ {
                          ContainsArray        ⌄ [GenObject ⌄ {
                                                   id              string
                                                   name            string
                                                   shortName       string
                                                   orgEncyId       number
                                                   orgId           number
                                                   type            string
                                                   updateDate      string
                                                   updateTime      string
                                                   contextName1    string
                                                   contextName2    string

                                                 }]
                          To                     › {...}
                          name                 string

                        }]
   id                   string
   name                 string
   ownerId              string
   release              string
   lastUpdateDate       string
   lastUpdateTime       string
}
```

2.1.2 Services



```
"/Models/{id}/copy": {
  "post": {
   "tags":["Models"],
    "summary": "",
    "description": "",
    "requestBody": {
     "content": {
       "application/json": {
         "schema": {
           "$ref": "#/components/schemas/Model"
         },
         "examples": {
           "request": {…
           }
         }
       },
       "application/xml": {
         "schema": {
           "$ref": "#/components/schemas/Model"
         },
         "examples": {
           "request": {…
           }
         }
       }
     }
    },
    "responses": {
     "200": {…
     },
     "500": {…
     }
    },
    "parameters": [
      {
        "name": "user",
        "in": "query",
        "required": false,
        "schema": {
          "type": "string"
        }
      },
    ]
  },
  "parameters": [
    {
     "name": "id",
     "in": "path",
     "required": true,
     "schema": {
       "type": "string"
     }
    },
  ]
},
```



Models

POST /Models/{id}/copy

Parameters

Try it out

Name	Description
user string (query)	user
id * required string (path)	id

Request body application/json

Examples: request

Example Value | Schema

```
{
  "name": "ABCDEFGHIJKLMNOPQRSTUVWXYZ..."
}
```

Responses

Code	Description	Links
200	Media type: application/json (Controls Accept header.) Examples: response	No links

Example Value | Schema

```
{
  "id": 1234567890,
  "status": "A",
  "comment": "abcdefghijklmnopqrstuvwxyz..."
}
```

Headers:

2.2 WADL

The fact that the **WADL** for a REST API has been designed, from the beginning, to be unambiguously processed by machines and its syntax, based on similar conventions as WSDLs, has made it the preferred Web API representation for a number of REST consuming frameworks (and certainly those also supporting SOAP), *among which Apache CXF*.

Fortunately, if the REST API you want to consume provides a formal specification in some other representation, you can rely on a number of different services[5] (like APIMATIC - https://www.apimatic.io/transformer/) to convert it to WADL.

There are 2 main possibilities for the design of WADL specifications:

- Put the whole specification into one file
- Separate the grammar of the data model from the rest, in the form of a reference to the xsd of the grammar in the main WADL file.

The second option will be used in this document, as it's easy to map to a data model-based approach to REST API consumption.

2.2.1 Data Model

```
<xs:element name="Model" type="ns0:model"/>
<xs:complexType name="model">
      <xs:sequence>
             <xs:element name="ContainsArray" type="ns0:aggSetArray" minOccurs="0">
             </xs:element>
     </xs:sequence>
     <xs:attribute name="id" type="xs:ID">
     </xs:attribute>
     <xs:attribute name="name" type="xs:string">
     </xs:attribute>
     <xs:attribute name="ownerId" type="xs:string">
     </xs:attribute>
     <xs:attribute name="release" type="xs:string">
     </xs:attribute>
     <xs:attribute name="lastUpdateDate" type="xs:string">
     </xs:attribute>
     <xs:attribute name="lastUpdateTime" type="xs:string">
     </xs:attribute>
</xs:complexType>
```

(Note that, in the specific extract of a WADL data model specification above, the *xs:ID* type identifies a [partial[6]] identifier for your specification type. It's not necessarily present in all data model specs).

[5] Broadcom Mainframe Services for CA Gen has also developed a converter from Swagger 2 to WADL, for batch processing
[6] As REST provides support for sub-resources, this id should be combined with the ones of parent resources, if any

2.2.2 Services

```
<wadl:resource path="Models/{id}/copy">
  <wadl:method name="POST">
    <wadl:doc><![CDATA[copy]]></wadl:doc>
    <wadl:request>
      <wadl:representation mediaType="application/json">
      </wadl:representation>
      <wadl:representation mediaType="application/xml" element="ns0:Model">
      </wadl:representation>
      <wadl:param name="user" style="query">
        <wadl:doc><![CDATA[]]></wadl:doc>
      </wadl:param>
    </wadl:request>
    <wadl:response>
      <wadl:doc><![CDATA[logging]]></wadl:doc>
      <wadl:representation mediaType="application/json"/>
      <wadl:representation mediaType="application/xml" element="ns0:Logging"/>
    </wadl:response>
  </wadl:method>
  <wadl:param name="id" style="template">
    <wadl:doc><![CDATA[]]></wadl:doc>
  </wadl:param>
</wadl:resource>
```

3 CA Gen integration

We must here distinguish between the kind of REST Services we want to consume. Are these ***ad-hoc REST services*** or do we want to consume a well-designed ***REST API***?

In the first case, easiest is probably to directly rely on views of existing entities and perform the ad-hoc mapping in an external action block.

In the second case, a much more structured approach will be welcome. As a consequence, this will be the sole focus[7] of this section.

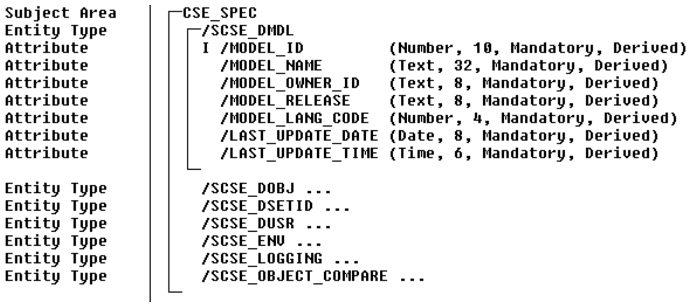
3.1 Component-Based Development

From its specification and our consumer point of view, whatever implementation is behind it, a REST API can be considered as an API to a Microservice. As described in the ***CA Gen Integration Solutions*** White Paper I wrote a few years ago, Microservices map very well to the much older, visionary, concept of Component-Based Development (CBD).

From our perspective, there are 2 major differences with our traditional CBD-based approach:

1. As there is no CA Gen-based component, we must build our specification model from scratch, rather than reusing /adapting pieces of implementation definitions
2. The usual trick CBD practicers use, which links specification definitions with implementation artifacts at runtime, must be replaced by a layer of External Action Blocks (EABs) that will perform data mapping and REST service invocations.

3.1.1 Data Model Specification



```
Subject Area      ┌CSE_SPEC
Entity Type       │ ┌/SCSE_DMDL
Attribute         │ │ I /MODEL_ID         (Number, 10, Mandatory, Derived)
Attribute         │ │   /MODEL_NAME       (Text, 32, Mandatory, Derived)
Attribute         │ │   /MODEL_OWNER_ID   (Text, 8, Mandatory, Derived)
Attribute         │ │   /MODEL_RELEASE    (Text, 8, Mandatory, Derived)
Attribute         │ │   /MODEL_LANG_CODE  (Number, 4, Mandatory, Derived)
Attribute         │ │   /LAST_UPDATE_DATE (Date, 8, Mandatory, Derived)
Attribute         │ │   /LAST_UPDATE_TIME (Time, 6, Mandatory, Derived)
                  │ └
Entity Type       │   /SCSE_DOBJ ...
Entity Type       │   /SCSE_DSETID ...
Entity Type       │   /SCSE_DUSR ...
Entity Type       │   /SCSE_ENV ...
Entity Type       │   /SCSE_LOGGING ...
Entity Type       │   /SCSE_OBJECT_COMPARE ...
                  └
```

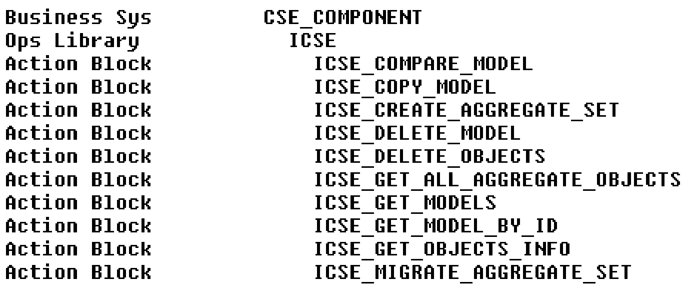
From the formal specification of the REST API, a specification data model is first constructed. This is a manual operation[8], but the mapping is quite straightforward, from any of the API formal type of representation (refer to the OpenAPI or WADL examples for a confirmation).

As usual for CBD, the owner subject area is of specification type, and the entity types are transient entity, specification or interface type.

[7] Although nothing prevents any of these guidelines to be used in other conditions as well

[8] Although some automation, at least partial, using one of the CA Gen APIs, could easily be developed

3.1.2 Functionality Specification



```
Business Sys         CSE_COMPONENT
Ops Library            ICSE
Action Block             ICSE_COMPARE_MODEL
Action Block             ICSE_COPY_MODEL
Action Block             ICSE_CREATE_AGGREGATE_SET
Action Block             ICSE_DELETE_MODEL
Action Block             ICSE_DELETE_OBJECTS
Action Block             ICSE_GET_ALL_AGGREGATE_OBJECTS
Action Block             ICSE_GET_MODELS
Action Block             ICSE_GET_MODEL_BY_ID
Action Block             ICSE_GET_OBJECTS_INFO
Action Block             ICSE_MIGRATE_AGGREGATE_SET
```

To mimic what happens with CBD, we take the following approach:

1. Group all the EABs that invoke services of the Web API in their own ***Business System***
2. As an option, create an ***Operations Library*** grouping those action blocks.

With this setup, if consumption if the Web API is needed in multiple models, it's easy and quick to migrate the Operations Library and start consuming its REST services

Advantage

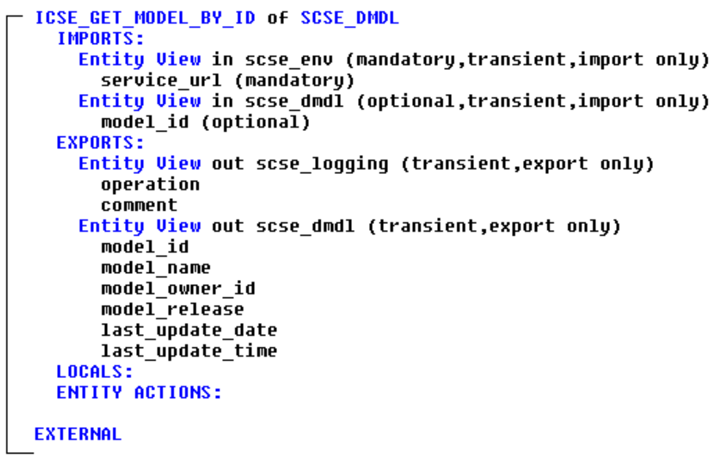
With this solution, as mentioned above, the whole set of EABs that consume the API will be managed together.

Caveat

In Java, such Operations Library would be defined for documentation only. As there is no distinction between static and dynamic libraries (jar files) in Java, and as all the effective action blocks in the Operations Library are external, the jar file defined with your Java IDE is directly used at runtime.

If this solution is selected, of course, the name of the Operations Library must be given to the Jar file containing the compiled EABs.

3.1.3 Typical definition of a REST operation



```
ICSE_GET_MODEL_BY_ID of SCSE_DMDL
  IMPORTS:
    Entity View in scse_env (mandatory,transient,import only)
      service_url (mandatory)
    Entity View in scse_dmdl (optional,transient,import only)
      model_id (optional)
  EXPORTS:
    Entity View out scse_logging (transient,export only)
      operation
      comment
    Entity View out scse_dmdl (transient,export only)
      model_id
      model_name
      model_owner_id
      model_release
      last_update_date
      last_update_time
  LOCALS:
  ENTITY ACTIONS:

EXTERNAL
```

As can be seen from the example on the left:

- The EAB, which invokes a service of a REST resource has been declared as ***operation*** of the ***specification type*** that defines the resource in the CA Gen model,
- It receives as imports:
  - The base URL for the Web API
  - The information the service needs as path or query parameter or as payload

- It returns as exports:
  - The information returned by the service as payload, header parameters, …
  - Some execution status information[9]

[9] In this example, the structure for logging-type of payload, returned by some services, has been reused

4 Using Apache CXF to consume REST Services

As said in the introduction, there are several ways of consuming REST Services and there is no doubt that Apache CXF can be used differently, using features that are provided either in the base product or through complementary libraries.

The approach taken at Broadcom Mainframe Services for CA Gen has been to create an API *data model* on the consumer (Apache CXF) side, then to use that data model to interact with CXF and the target REST Service.

Also, as our infrastructure is not sensitive to the message format, we've decided to rely on the default XML serialization, rather than use the less immediate JSON format.

4.1 Creation of the Apache CXF Java artifacts

As briefly told before, this step consists in taking a formal representation of the API (which should normally be provided by its publisher), and processing it using Apache CXF tools.

4.1.1 WADL processing

Processing of WADL files by Apache CXF wadl2java utility is done like this:

```
rmdir /S /Q src
mkdir src
rmdir /S /Q classes
mkdir classes
call M:\ck609269\JavaUtils\apache-cxf-3.2.1\bin\wadl2java -p com.ca.optsv.gen.cse.rest -d src -verbose application.wadl
```

The first command statements are used to clean up the current generation environment, to avoid any obsolete code, from a prior implementation.

Let's look in details at the wadl2java command:

```
wadl2java -p com.ca.optsv.gen.cse.rest -d src -verbose application.wadl
```

In this command, we specify, in this order:

- ***Package name*** for the generated Java classes (Web API-specific)
- ***Root directory*** for the generated Java classes
- ***Verbose*** console logging (optional)
- ***WADL*** file to be processed is ***application.wadl***[10]
- ***-javaDocs*** is an option that can also be added, to incorporate documentation present in the WADL into javadoc of the generated Java code.

This command generates a consistent set of Java classes that represent the data model specified by the WADL.

[10] The data model xsd specification (*ns0.xsd*) will automatically be included, and must be placed in the same folder as *application.wadl*

(Note that WADL2JAVA also generates interfaces for services that could be used on REST server side. We'll come back to those later).

Below an example of generated class for a resource specified in the WADL:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "model", propOrder = {
    "containsArray"
})
public class Model {

    @XmlElement(name = "ContainsArray")
    protected AggSetArray containsArray;
    @XmlAttribute(name = "id")
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    @XmlID
    @XmlSchemaType(name = "ID")
    protected String id;
    @XmlAttribute(name = "name")
    protected String name;
    @XmlAttribute(name = "ownerId")
    protected String ownerId;
    @XmlAttribute(name = "release")
    protected String release;
    @XmlAttribute(name = "lastUpdateDate")
    protected String lastUpdateDate;
    @XmlAttribute(name = "lastUpdateTime")
    protected String lastUpdateTime;

    [...]
```

From the example above we see:

- JAXB annotations are used to manage serialization / deserialization between XML message and Java object fields
- JAXB doesn't mandate an XmlRootElement in all cases, which specifies the XML name of the element corresponding to the class[11], and Apache CXF has decided not to generate the corresponding annotation, but rely on an ObjectFactory class instead.

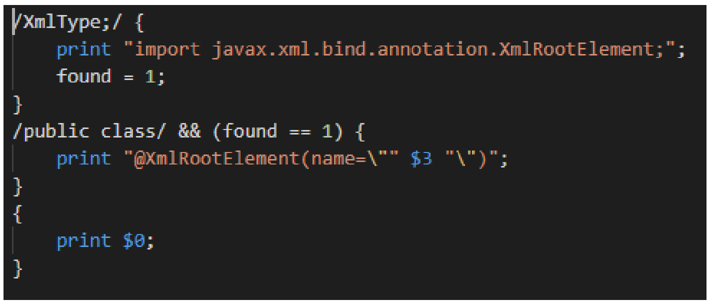
4.1.2 @XmlRootElement annotation

Rather than complexify the processing without any advantage for us, Broadcom Mainframe Services has decided to alter the generated classes and add the @XmlRootElement annotation in all classes where it makes sense.

This is done by the rest of our *generateClient.cmd* command:

[11] This allows for cases where the same class is associated with multiple XML tags

```
cd src\com\ca\optsv\gen\cse\rest
for %%f in (*.java) do call :process %%f
pause
goto :EOF

:process
gawk -f "W:\Gen x-CIDE\Development\Java\xCIDE Java Clients\Helpers\AddXmlRootElement.awk" %1 > %1.new
del %1
ren %1.new %1
```



```
/XmlType;/ {
    print "import javax.xml.bind.annotation.XmlRootElement;";
    found = 1;
}
/public class/ && (found == 1) {
    print "@XmlRootElement(name=\"" $3 "\")";
}
{
    print $0;
}
```

This is very easily done with the ***gawk***[12] script on the left.

This script says that:

- If the string “XmlType:” is found, we need to add the XmlRootElement annotation
- The name of the root element is the 3rd word of the line containing the class declaration, and we have to write the annotation before that declaration
- Except for this insertion, all file lines are written unchanged.

4.1.3 Alternative to @XmlRootElement annotation

Instead of altering the generated code like before, the generated ***ObjectFactory*** class can be used to work with the data model.

It’s then the responsibility of that class to trigger serialization / deserialization between XML and data model elements.

4.1.4 API-specific documentation

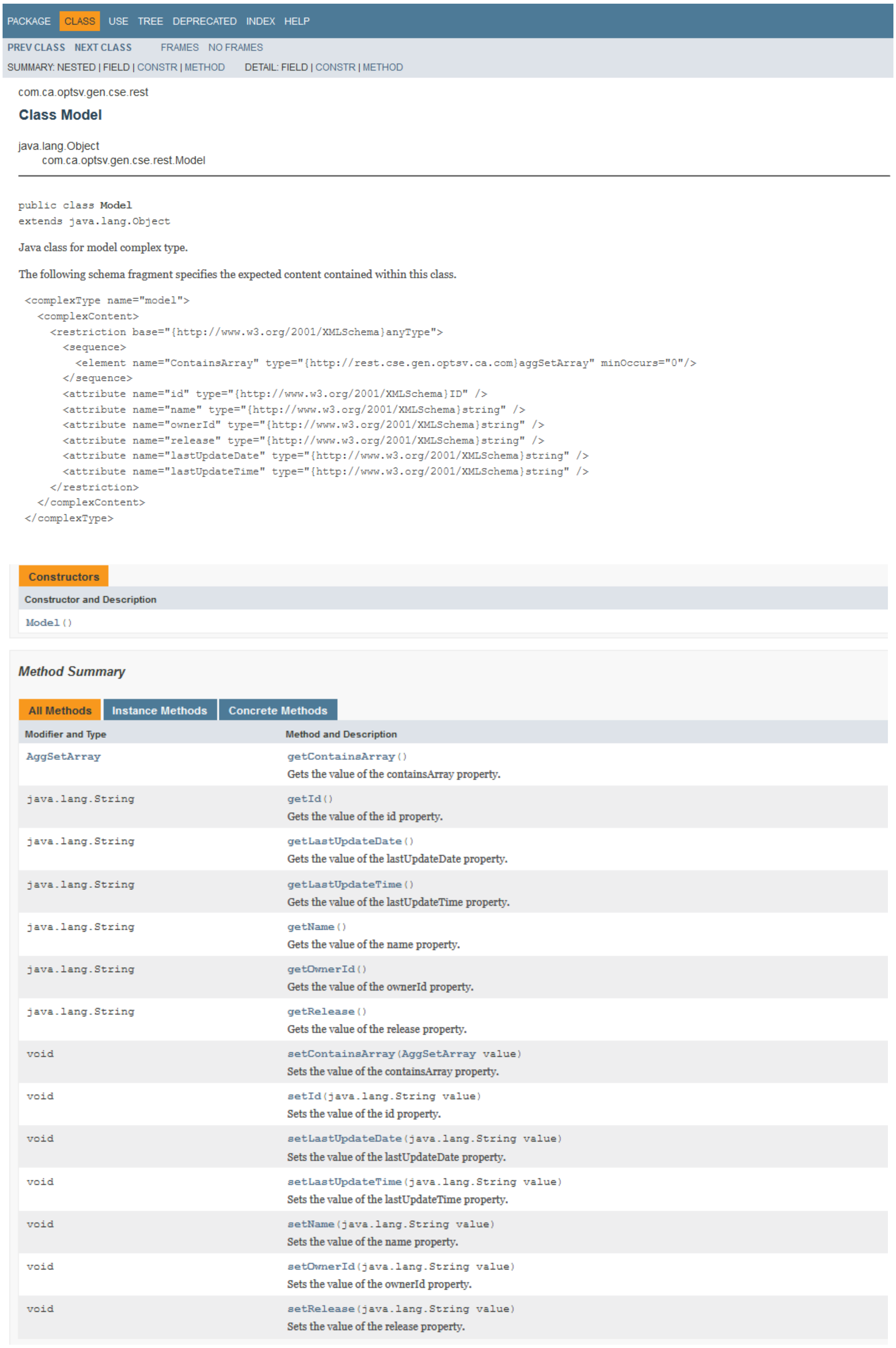
4.1.4.1 Javadoc-generated documentation

4.1.4.1.1 Data Model documentation

Apache CXF WADL2JAVA utility generates Javadoc comments to document the generated data model, in terms of

- XML element structure
- Getters and setters for Java object fields

[12] gawk is a Linux open source tool, which has been ported to many other platforms, like Unix or WIndows



PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

com.ca.optsv.gen.cse.rest

Class Model

java.lang.Object
    com.ca.optsv.gen.cse.rest.Model

```
public class Model
extends java.lang.Object
```

Java class for model complex type.

The following schema fragment specifies the expected content contained within this class.

```
 <complexType name="model">
   <complexContent>
     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
       <sequence>
         <element name="ContainsArray" type="{http://rest.cse.gen.optsv.ca.com}aggSetArray" minOccurs="0"/>
       </sequence>
       <attribute name="id" type="{http://www.w3.org/2001/XMLSchema}ID" />
       <attribute name="name" type="{http://www.w3.org/2001/XMLSchema}string" />
       <attribute name="ownerId" type="{http://www.w3.org/2001/XMLSchema}string" />
       <attribute name="release" type="{http://www.w3.org/2001/XMLSchema}string" />
       <attribute name="lastUpdateDate" type="{http://www.w3.org/2001/XMLSchema}string" />
       <attribute name="lastUpdateTime" type="{http://www.w3.org/2001/XMLSchema}string" />
     </restriction>
   </complexContent>
 </complexType>
```

Constructors

Constructor and Description
`Model()`

Method Summary

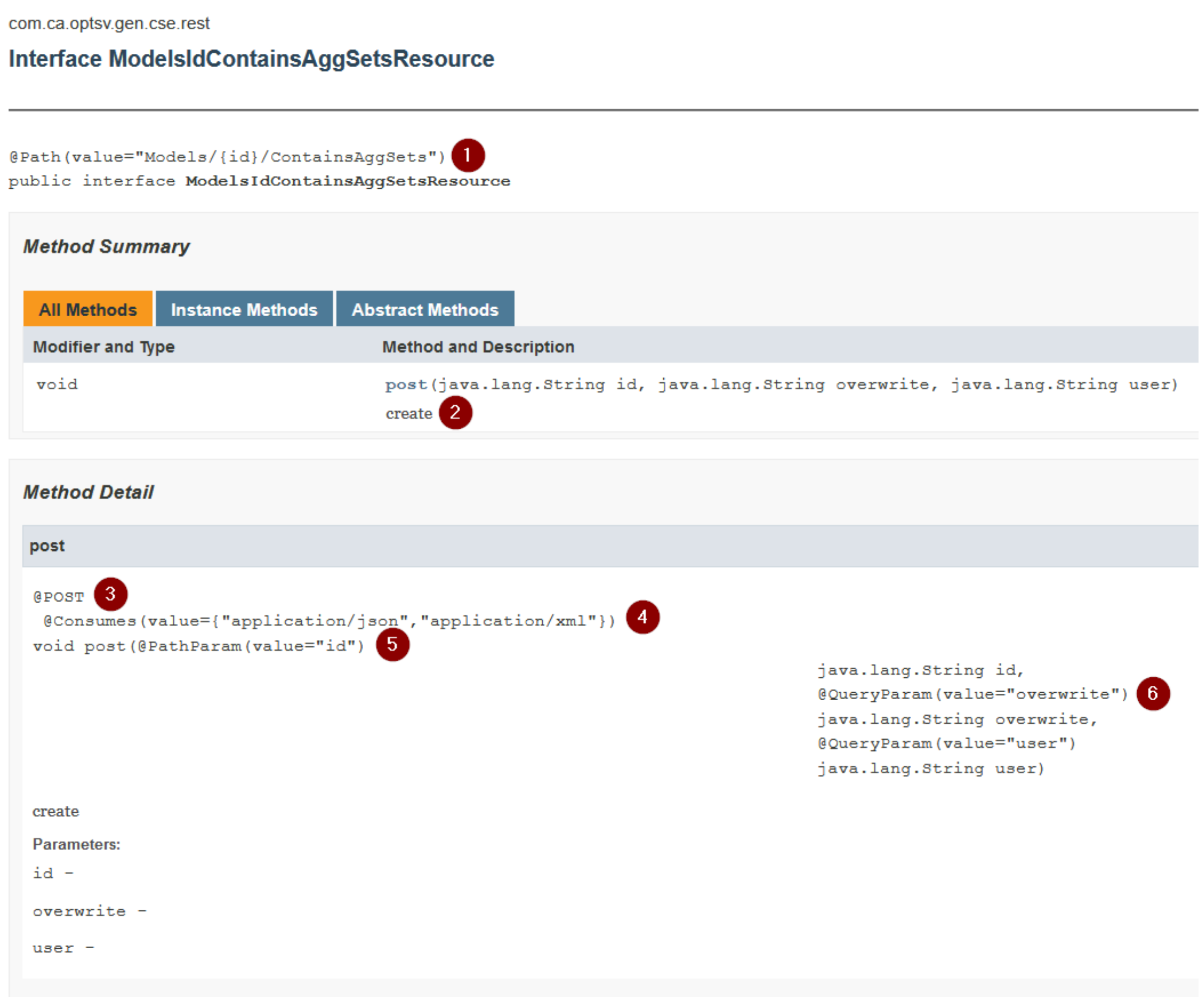
All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`AggSetArray`	`getContainsArray()` Gets the value of the containsArray property.
`java.lang.String`	`getId()` Gets the value of the id property.
`java.lang.String`	`getLastUpdateDate()` Gets the value of the lastUpdateDate property.
`java.lang.String`	`getLastUpdateTime()` Gets the value of the lastUpdateTime property.
`java.lang.String`	`getName()` Gets the value of the name property.
`java.lang.String`	`getOwnerId()` Gets the value of the ownerId property.
`java.lang.String`	`getRelease()` Gets the value of the release property.
`void`	`setContainsArray(AggSetArray value)` Sets the value of the containsArray property.
`void`	`setId(java.lang.String value)` Sets the value of the id property.
`void`	`setLastUpdateDate(java.lang.String value)` Sets the value of the lastUpdateDate property.
`void`	`setLastUpdateTime(java.lang.String value)` Sets the value of the lastUpdateTime property.
`void`	`setName(java.lang.String value)` Sets the value of the name property.
`void`	`setOwnerId(java.lang.String value)` Sets the value of the ownerId property.
`void`	`setRelease(java.lang.String value)` Sets the value of the release property.

4.1.4.1.2 Services documentation

Apache CXF WADL2JAVA utility also generates Javadoc comments to document the REST services, in the corresponding <*Service*>*Resource* interfaces, which are even completed with documentation in the WADL specification, if the ***–javaDocs*** option has been specified for the WADL2JAVA command.

Here is an example of the documentation of a service to create an aggregate set in our CSE:



com.ca.optsv.gen.cse.rest

Interface ModelsIdContainsAggSetsResource

```
@Path(value="Models/{id}/ContainsAggSets") 1
public interface ModelsIdContainsAggSetsResource
```

Method Summary

Modifier and Type	Method and Description
void	post(java.lang.String id, java.lang.String overwrite, java.lang.String user) create 2

Method Detail

post

```
@POST 3
 @Consumes(value={"application/json","application/xml"}) 4
void post(@PathParam(value="id") 5
                                                 java.lang.String id,
                                                 @QueryParam(value="overwrite") 6
                                                 java.lang.String overwrite,
                                                 @QueryParam(value="user")
                                                 java.lang.String user)
```

create

Parameters:

`id -`

`overwrite -`

`user -`

In this example, we see:

1. The path to access the service
2. The documentation of the service
3. The method we need to use to invoke it
4. The message formats supported by the service
5. Path parameters (if any)
6. Query parameters (if any)
7. The structure returned by the service (if any, here, there is no payload returned by the service).

4.2 External Action Block design

To increase reusability, we, at Broadcom Mainframe Services, don't split/reproduce common logic across multiple EABs. The way we proceed is to write "Helper" class(es), and only leave the specifics of filling in the Java data model instances with information (mainly) from our import views and fetching the results in our data model instances into our export views.

As a consequence, content of EABs is very specific and won't be further described here.

4.3 Apache CXF Helper classes

4.3.1 RESTHelper

This is our main helper class. It contains the following:

- Package, imports and class declarations:

```java
package com.ca.optsv.gen.cid.cse;

import java.util.*;

import javax.ws.rs.client.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import com.ca.optsv.gen.cse.rest.AggSet;

public class RESTHelper {
```

- Static declarations and load-time processing:

```java
    private static Client cseClient;
    private static WebTarget rootCseTarget;

    static {
        cseClient = ClientBuilder.newClient();
        cseClient.property("http.receive.timeout", 300000);
        cseClient.property("http.send.timeout", 300000);
        cseClient.property("http.read.timeout", 300000);
        PropertyResourceBundle bundle = (PropertyResourceBundle) ResourceBundle.getBundle("xcide");
        String restCseUrl = bundle.getString("CSE.URL");
        rootCseTarget = cseClient.target(restCseUrl);
    }
```

  This initializer creates instances of:
  - A CXF client, with its timeouts
  - A URL root target (with base URL taken from a property file)
- The methods used to invoke the REST services, like:

```java
    public static RESTStatus createAggregateSet(String cseUser, Long modelId, AggSet aggSet, boolean overwrite) {
        WebTarget target = rootCseTarget.path("Models").path(modelId.toString()).path("ContainsAggSets").queryParam("user", cseUser);
        if (overwrite)
            target = target.queryParam("overwrite", "Y");
        Invocation.Builder builder = target.request();
        Entity<AggSet> entity = Entity.entity(aggSet, "APPLICATION/XML");
        Response response = builder.post(entity);
        return new RESTStatus(response.getStatus(), response.getHeaderString("StatusSeverity"), response.getHeaderString("StatusMessage"));
    }
```

  The method:
  - First creates the actual URL target, by cloning the root target and appending the necessary path and query parameter components
  - Fetches the request object from the target
  - Serializes the request payload into an XML message within a generic entity object
  - Invokes the REST service
  - Returns a RESTStatus object, containing the HTTP status and, in this case, some specific header information.

4.3.1.1 Basic authenticiation

If the Web API requires basic authentication the RESTHelper class also contains a logon method, which makes use of another helper class:

```
public static void publicLogon(String userid, String password) {
    rootPublicTarget.register(new RESTAuthenticator(userid, password));
}
```

4.3.2 RESTAuthenticator

The RESTAuthenticator class contains basic authentication data (userid and password), and is invoked at runtime by Apache CXF, when invoking the service.

```
package com.ca.optsv.gen.cid.cse;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Base64;

import javax.ws.rs.client.*;
import javax.ws.rs.core.HttpHeaders;
import javax.ws.rs.ext.Provider;

@Provider
public class RESTAuthenticator implements ClientRequestFilter {

    private String user;
    private String password;

    public RESTAuthenticator(String user, String password) {
        this.user = user;
        this.password = password;
    }

    @Override
    public void filter(ClientRequestContext requestContext) throws IOException {
        requestContext.getHeaders().add(
            HttpHeaders.AUTHORIZATION, getBasicAuthentication());
    }

    private String getBasicAuthentication() throws UnsupportedEncodingException {
        String userAndPassword = this.user + ":" + this.password;
        byte[] userAndPasswordBytes = userAndPassword.getBytes("UTF-8");
        return "Basic " + Base64.getEncoder().encodeToString(userAndPasswordBytes);
    }
}
```

4.4 Build / runtime configuration

The following Java archives are needed to build or run the REST clients developed using Apache CXF (reference: Apache CXF 3.2.1):

- cxf-core-3.2.1.jar
- cxf-rt-frontend-jaxrs-3.2.1.jar
- cxf-rt-rs-client-3.2.1.jar
- cxf-rt-rs-extension-providers-3.2.1.jar
- cxf-rt-transports-http-3.2.1.jar
- javax.annotation-api-1.3.jar
- javax.ws.rs-api-2.1.jar
- stax2-api-3.1.4.jar
- woodstox-core-asl-4.4.1.jar

5 Conclusion

In this document, we've shown (or at least tried to show) that, even without any automation, consuming REST services, even in large, is quite feasible in Java.

There is of course much more to the topic than this. We, at Broadcom Mainframe Services, have many more examples, with cases that couldn't be detailed here, and are here to help you.

Appendix A. Examples

CXF Helper class

```
package com.ca.optsv.gen.cid.cse;

import java.util.*;

import javax.ws.rs.client.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import com.ca.optsv.gen.cse.rest.AggSet;

public class RESTHelper {

       private static Client cseClient;
       private static WebTarget rootCseTarget;

       static {
              cseClient = ClientBuilder.newClient();
              cseClient.property("http.receive.timeout", 300000);
              cseClient.property("http.send.timeout", 300000);
              cseClient.property("http.read.timeout", 300000);
              PropertyResourceBundle bundle =
                 (PropertyResourceBundle) ResourceBundle.getBundle("xcide");
              String restCseUrl = bundle.getString("CSE.URL");
              rootCseTarget = cseClient.target(restCseUrl);
       }

       public static RESTStatus createAggregateSet
         (String cseUser, Long modelId, AggSet aggSet, boolean overwrite) {
              WebTarget target = rootCseTarget.path("Models")
                 .path(modelId.toString())
                 .path("ContainsAggSets").queryParam("user", cseUser);
              if (overwrite)
                     target = target.queryParam("overwrite", "Y");
              Invocation.Builder builder = target.request();
              Entity<AggSet> entity = Entity.entity(aggSet, "APPLICATION/XML");
              Response response = builder.post(entity);
              return new RESTStatus(response.getStatus(),
                 response.getHeaderString("StatusSeverity"),
                 response.getHeaderString("StatusMessage"));
       }
```

REST Service invocation

```
package com.ca.optsv.gen.cid.cse;


import com.ca.optsv.gen.cse.rest.AggSet;
import com.ca.optsv.gen.cse.rest.GenObject;
import com.ca.optsv.gen.cse.rest.GenObjectArray;


public class MigrationBaselineUpdate {

    // [...]

      private static void createSynchronizationSet(String logFileName)
       throws EncyException, ApiException, IOException, InterruptedException {
             AggSet syncSet = new AggSet();
             syncSet.setName(("$" +
                srcModel.getName() + "                              ")
            .substring(0, 30) + " B");
             GenObjectArray array = new GenObjectArray();
             syncSet.setContainsArray(array);
             List<GenObject> syncObjects = array.getGenObject();
             for (String sync: synchronizationArray) {
                    GenObject syncObject = new GenObject();
                    syncObjects.add(syncObject);
                    syncObject.setId(sync.split(";")[0]);
             }
             RESTStatus status = RESTHelper.createAggregateSet
                (cseUser, model.getId(), syncSet, true);
             if (status.getHttpStatus() >= 300) {
                    throw new RuntimeException
                ("Failed to send aggregate set creation, status: "
                + status.toString());
             }
      }
}
```

(Note that, in this example, the AggSet, GenObjectArray and GenObject classes have been generated by the Apache CXF WADLTOJAVA utility, based on the API documentation).