REST Coder Documentation

Release 0.1

Hiranya Jayathilaka, Stratos Dimopoulos

January 05, 2016

Contents

Contents:

CHAPTER 1

Introduction

REST Coder is a collection of tools for auto-generating documentation and client stubs for RESTful web APIs. This release of REST Coder comes with four primary tools.

- Client stub generators
  - Python client stub generator
  - JavaScript/AJAX client stub generator
- API doc generators
  - Sphinx API doc generator
  - HTML API doc generator

Each of the above tools take an API description (specification) in JSON format as input. Client stubs generated by REST Coder tools can be embedded in a wide range of desktop, mobile and web applications to consume the target web APIs with or without human intervention. The documentation generated by these tools can be published on the web as reference material for the developers and users. The REST Coder stub generators are primarily designed for application and mashup developers who regularly need to write code that consumes one or more web APIs. The REST Coder documentation generators allow generating high-quality API docs from a given API specification, which allows API providers to publish comprehensive API docs for their APIs, and speed up the release cycles by avoid spending time on manually writing and editing API docs.

1.1 Prerequisites

The Python client stub generator and Sphinx API doc generator require following software to be setup.

- Python 2.7
- Sphinx documentation engine

The AJAX client stub generator and the HTML API doc generator require following software to be setup.

- JDK 1.6 or higher
- NodeJS and Express (Installation Instructions included on the documentation)

1.2 Installation

Use a Git client to check out the necessary source and binary artifacts of RESTCoder.

```
git clone https://github.com/hiranya911/rest-coder.git
```

No additional installation/setup procedures are necessary.

If you also wish to setup the sample `Starbucks` service to run some tests using RESTCoder, you need to first download and install Apache Tomcat 6.0 or higher. Copy the `starbucks-1.0-SNAPSHOT.war` file in the `java-lib` directory of RESTCoder into the `webapps` directory of Tomcat, and start the Tomcat server. The mock service will be available on the URL http://localhost:8080/starbucks-1.0-SNAPSHOT

1.3 Next Steps

Refer the documentation of individual REST Coder tools to learn more about how to use them.

CHAPTER 2

API Description Language

All REST Coder tools take an API description (specification) as input. These API descriptions should be specified in a JSON based language, as described below.

The JSON based API description language captures all the resources, operations and data types related to a web API. It also captures a wide range of non-functional properties of APIs such as licensing information, ownership information and SLA details. These API descriptions can be compiled manually or generated automatically by analyzing the source code of the web API/service implementations.

The high-level grammar/structure of this API description language is described in the next section.

2.1 Language Grammar

2.1.1 API

```
{
  "name" : "string",
  "description" : "string",
  "version": {
      "identifier" : "string",
      "scheme" : "BaseAppend|Header|None",
      "compatibility" : [ "string", "string", "string" ]
  },
  "base" : [ "url", "url" ],
  "state" : "active|deprecated|retired",

  "resources" : [...],
  "dataTypes" : [...],

  "security" : {
      "ssl" : "Always|Never|Optional",
      "auth" : "Basic|OAuth",
      "credentials" : "url"
  },

  "license" : "string",
  "ownership" : [...],
  "categories" : [ "string", "string", "string" ],
  "tags" : [ "string", "string", "string" ],
  "community" : "url"
  "sla" : [...]
}
```

2.1.2 Resource

```
{
  "name" : "string",
  "path" : "uri-template",
  "inputBindings" : [...],
  "operations" : [...]
}
```

2.1.3 Input Binding

```
{
  "id" : "string",
  "mode" : "url|query|header",
  "name" : "string",
  "type" : "TypeRef|TypeDef",
}
```

2.1.4 Operation

```
{
  "name" : "string",
  "method" : "GET|POST|PUT|DELETE|OPTIONS|HEAD",
  "description" : "string",
  "input" : {
      "contentType" : [ "mime-type", "mime-type" ],
      "type" : "TypeDef|TypeRef",
      "params" : [...]
  },
  "output" : {
      "status" : HTTP status code,
      "contentType" : [ "mime-type", "mime-type" ],
      "model" : "TypeDef|TypeRef",
      "headers" : [...]
  },
  "errors" : [...]
}
```

2.1.5 Parameter

```
{
  "binding" : "string",
  "description" : "string"
  "optional" : true|false
}
```

Or:

```
{
  "mode" : "url|query|header",
  "name" : "string",
  "type" : "TypeRef|TypeDef",
  "description" : "string"
  "optional" : true|false
}
```

2.1.6 Header

```
{
  "name" : "string",
  "type" : TypeRef|TypeDef,
  "ref" : "string",
  "description" : "string"
}
```

2.1.7 Error

```
{
  "status" : HTTP status code,
  "cause" : "string"
}
```

2.1.8 Type Definition (TypeDef)

```
{
  "name" : "string",
  "description" : "string",
  "fields" : [...]
}
```

2.1.9 Field

```
{
  "name" : "string",
  "description" : "string",
  "type" : "TypeRef | TypeDef",
  "optional" : true | false,
  "ref" : "string",
  "multi" : true | false
}
```

2.1.10 Type Reference (TypeRef)

A type reference is a string literal that points to a primitive type, container type or a user defined type.

```
PrimitiveTypeName | ContainerTypeName | UserDefinedTypeName
```

This API description language supports following primitive types.

- int
- long
- short
- double
- string
- boolean
- byte
- binary
- href

A container type is a type reference wrapped in one of the following containers.

- list
- set

2.1.11 SLA

```
{
  "name" : "string",
  "availability" : percentage,
  "rateLimit" : int,
  "timeUnit" : "second|minute|hour|day",
  "costModel" : {
      "unitPrice" : double,
      "currency" : "string",
      "requestsPerUnit" : int
  }
}
```

2.1.12 Owner

```
{
  "name" : "string",
  "email" : "string",
  "ownerType" : "string"
}
```

2.2 Example API Description

This section further explains the syntax and semantics of the API description language using the specification of a hypothetical API named `Starbucks` as an example.

```
{
  "name":"Starbucks",
  "resources":[
      {
          "name":"Order",
          "path":"/{orderId}",
          "operations":[
              {
                  "name":"getOrder",
                  "method":"GET",
                  "description":"Retrieve the order identified by the specified identifier",
                  "input":{
                      "params":[
                          {
                              "optional":false,
                              "binding":"orderIdBinding"
                          }
                      ]
                  },
                  "output":{
                      "type":"Order",
                      "contentType":["application/json"],
                      "status":200
                  },
                  "errors":[
                      {
                          "cause":"Specified order does not exist",
                          "status":404
                      },
                      {
                          "cause":"An unexpected runtime exception",
                          "status":500
                      }
                  ]
              },
              {
                  "name":"deleteOrder",
                  "method":"DELETE",
                  "description":"Remove the order identified by the specified ID from the system",
                  "input":{
                      "params":[
                          {
                              "optional":false,
                              "binding":"orderIdBinding"
                          }
                      ]
                  },
                  "output":{
                      "type":"Order",
                      "contentType":["application/json"],
                      "status":200
                  },
                  "errors":[
                      {
                          "cause":"Specified order does not exist",
                          "status":404
                      },
                      {
                          "cause":"An unexpected runtime exception",
                          "status":500
                      }
                  ]
              }
          ],
          "inputBindings":[
              {
                  "id":"orderIdBinding",
                  "name":"orderId",
                  "type":"string",
                  "mode":"url"
              }
          ]
      },
      {
          "name":"AllOrders",
          "path":"/",
          "operations":[
              {
                  "name":"submitOrder",
                  "method":"POST",
                  "description":"Place a new drink order.",
                  "input":{
                      "type":"OrderRequest",
                      "contentType":["application/json", "application/xml"]
                  },
                  "output":{
                      "type":"Order",
                      "contentType":["application/json"],
                      "headers":[
                          {
                              "name":"Location",
                              "type":"href",
                              "ref":"Order",
                              "description":"A URL pointer to the Order resource created by this oper
                          }
                      ],
                      "status":201
                  },
                  "errors":[
                      {
                          "cause":"An unexpected runtime exception",
                          "status":500
                      }
                  ]
              },
              {
                  "name":"getAllOrders",
                  "method":"GET",
                  "description":"Retrieve all the orders currently pending in the system",
                  "output":{
                      "type":"list(Order)",
                      "contentType":["application/json"],
                      "status":200
                  },
                  "errors":[
                      {
                          "cause":"An unexpected runtime exception",
                          "status":500
                      }
                  ]
              }
          ]
      }
  ],
  "description":"Place and manage drink orders online.",
  "categories":["marketing", "retail"],
  "tags":["beverages", "recreation", "marketing", "sales"],
  "base":[
      "http://localhost:8080/starbucks-1.0-SNAPSHOT/starbucks",
      "https://localhost:8243/starbucks-1.0-SNAPSHOT/starbucks"
  ],
  "dataTypes":[
      {
          "name":"Order",
          "fields":[
              {
                  "name":"orderId",
                  "type":"string",
                  "description":"Unique system generated string identifier of the drink.",
                  "optional":false,
                  "unique":true
              },
              {
                  "name":"drink",
                  "type":"string",
                  "description":"Name of the drink",
                  "optional":false
              },
              {
                  "name":"additions",
                  "type":"list(string)",
                  "description":"List of additions (flavors) to be included in the drink",
                  "optional":true
              },
              {
                  "name":"cost",
                  "type":"double",
                  "description":"Cost of the drink in USD",
                  "optional":false
              },
              {
                  "name":"next",
                  "type":"href",
                  "ref":"Order",
                  "description":"A URL pointing to the next resource in the workflow"
              }
          ],
          "description":"Describes an order submitted to the system."
      },
      {
          "name":"OrderRequest",
          "fields":[
              {
                  "name":"drink",
                  "type":"string",
                  "description":"Name of the drink to order",
                  "optional":false
              },
              {
                  "name":"additions",
                    "type":"list(string)",
                    "description":"A list of additions to be included in the drink",
                    "optional":true
                }
            ],
            "description":"Describes an order that can be submitted to the system by a client applicat
        }
    ]
}
```

This specification describes an API with two resources.

- Order
- AllOrders

The `AllOrders` resource can be used to submit orders (`submitOrder` operation) and retrieve a list of all pending orders (`getAllOrders` operation). The `input` section of the `submitOrder` operation indicates that the operation takes a JSON or XML payload and that payload should describe an `OrderRequest` object. The type `OrderRequest` is fully defined in the `dataTypes` section of the API specification. The `output` section of the `submitOrder` operation indicates that upon successful completion of the request, the API returns a `HTTP 201 Created` response with a JSON payload. This output JSON payload encodes an `Order` object, whose type is also defined in the `dataTypes` section. The `output` configuration of the `submitOrder` operation further specifies that the response from the API contains a HTTP `Location` header.

Now lets take a close look at a data type definition.

```
{
  "name":"OrderRequest",
  "fields":[
    {
      "name":"drink",
      "type":"string",
      "description":"Name of the drink to order",
      "optional":false
    },
    {
      "name":"additions",
      "type":"list(string)",
      "description":"A list of additions to be included in the drink",
      "optional":true
    }
  ],
  "description":"Describes an order that can be submitted to the system by a client application."
}
```

Above TypeDef element defines a complex type named `OrderRequest`, which is the type of the input payload to the `submitOrder` operation. According to this type definition, an object of type `OrderRequest` contains two data fields. The `drink` field is a simple string field and is required. The `additions` field is a list of string values and is optional. Following JSON string specifies a payload that adheres to the above type definition.

```
{
  "drink" : "Frapacinno",
  "additions" : [ "caramel", "whip cream" ]
}
```

When serialized into XML the same object may look something like this.

```
<OrderRequest>
  <drink>Frapacinno>
  <additions>caramel</additions>
  <additions>whip cream</additions>
</OrderRequest>
```

The API description language also allows defines anonymous types inside operations, bindings and other data type definitions.

```
{
  "name":"Foo",
  "fields":[
    {
      "name":"foo",
      "type": {
        "fields" : [
          {
            "name":"bar",
            "type":int
          }
        ]
      },
    }
  ]
}
```

The input bindings are used to define operation input parameters that are extracted from non-payload elements of the HTTP request. For instance an operation may extract certain input data items from the HTTP header or URL of the request. As a more concrete example, take a look at the `getOrder` operation of the above `Starbucks` API specification. This operation extracts the `orderId` value from the request URL and therefore the `orderId` parameter has been defined as a reference to an externally defined input binding. Also note that several operations make use of the same URL based input parameters and therefore, defining this piece of information as an external input binding, makes it possible to share that definition across multiple operations.

The `Starbucks` API specification defines two base URLs for the API. When invoking a particular operation of the API, the URL path of the corresponding resource must be appended to these base URLs. For example if the `getOrders` operation is needed to be invoked, the URL fragment `/{orderId}`, should be appended to one of the base URLs to construct the full URL of the request. Further note that `/{orderId}` itself is a URI template with the variable `orderId`. This value should be filled in by the client during invocation time.

2.3 Validity of an API Description

An API description is valid if it satisfies the following conditions.

- API has a name (has a `name` attribute)
- API has at least one base URL (has a `base` attribute pointing to a non-empty array)
- API has at least one resource (has a `resources` attribute pointing to a non-empty array)
- Each resource has at least one operation (each `resource` element has an `operations` attribute pointing to a non-empty array)
- Each operation has a HTTP method (each `operation` element has a `method` attribute)
- There are no references to undefined types
- There are no references to undefined input bindings

Note that the above conditions allow for many information (fields) to be left out from an API specification. For instance all the description fields, error fields and header fields can be left out. Also all the non-functional fields such as `license`, `community` and `tags` can be left out from an API specification.

CHAPTER 3

HTML/ JQuery API Doc and Code Generator

The HTML/ JQuery API doc and code generator auto-generates HTML code augmented with JQuery code to allow users to view API documentation in a clear HTML format and directly interact with the API invoking the various operations that it supports. The only input required is the JSON based API description as described earlier. The tool includes 3 different parts:

- An html UI generator coded with JAVA
- A JQuery client who is able to get the posted user information from the UI, recognize the operation it belongs to and send it to the proxy server
- A proxy server builded with NodeJS that forwards the above request to the server that hosts the API

The html UI generator will output an index.html file that contains all the documentation. All the elements and place-holders are structured and named in such away that each HTTP request can be uniquely identified by the JQuery client. The JQuery client reads this information and makes an asynchronous call to the NodeJS server, who in turn also makes an asynchronous HTTP request to the API server. When the necessary information is returned the NodeJS server pushes the information back to the JQuery client that finally injects it on the appropriate fields of the user interface.

3.1 System Architecture

In the above architecture we see that having as our only input the API description in JSON format, we generate HTML/ JQuery code all included in a single (index.html) file. NodeJS acts both as a server for our application and as a reverse proxy that forwards the user requests to the actual API server. We should mention here that because of the same origin security policy we cannot directly invoke our API methods using ajax through our user interface, but we should first send the request to another app hosted on the NodeJS server that acts as a reverse proxy. The proxy will form an HTTP request and make an asynchronous call to the API and when it gets the answer it will push it back to the user interface and JQuery will finally inject the response to the HTML code.

3.2 Using the Doc Generator

To run the code you need to be on the java-lib directory and type the following command that runs the code and takes as an input argument the starbucks.json API description.

```
~/rest-coder/java-lib$ ./jsgen.sh ../java-src/input/starbucks.json
```

and the expected output after you run this command should be:

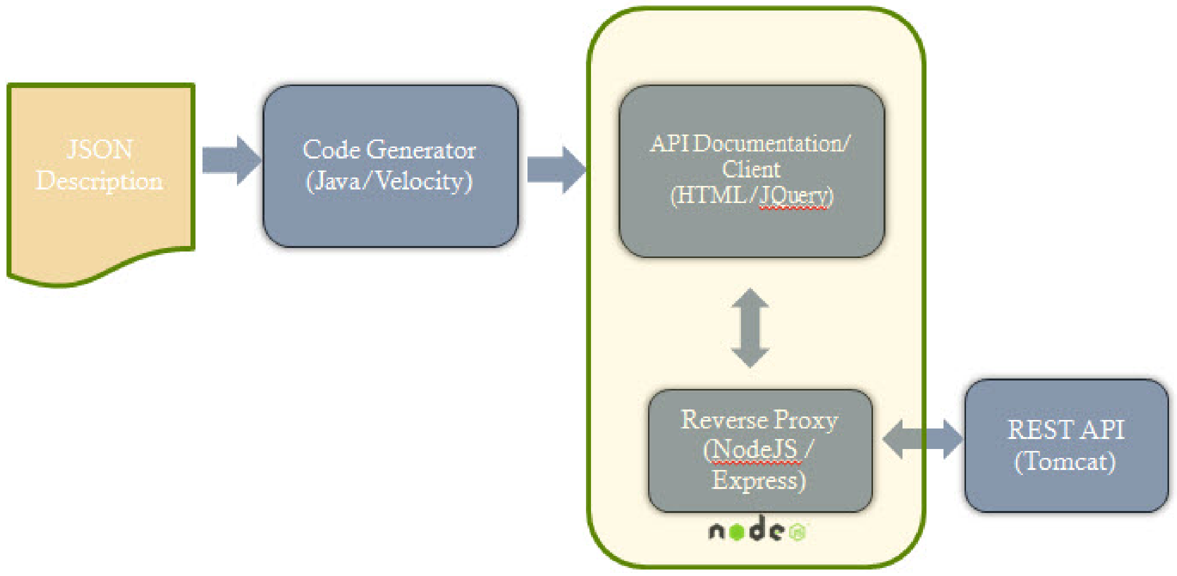
JSON Description
Code Generator (Java/Velocity)
API Documentation/ Client (HTML/JQuery)
Reverse Proxy (NodeJS / Express)
node
REST API (Tomcat)

Fig. 3.1: HTML/ JQuery doc and code generator architecture

```
Code successfully generated!
```

Of course if you prefer you can setup the code in Eclipse or the IDE of your preference and run this code by specifying the necessary argument through the project properties.

3.3 Generated Code

Running the above script will place the generated index.html file directly to the directory from witch NodeJS reverse-proxy server reads (ie: api-proxy/public/ directory).

This directory contains all the necessary filesd (CSS, Javascript etc) that our NodeJS server needs to serve the user interface. NodeJS serves our html code by reading from those directories. So it is important to maintain the same structure.

3.4 Setting up NodeJS/ Express

To use the generated code you have to first install NodeJS which is the technology we used to build our reverse proxy server, required to overcome the same origin security policy constraint that doesn't allow us to directly send our requests from the user interface to the API server.

In order to install NodeJS and the requirements for this particular application you have to do the following:

```
Install nodejs: http://howtonode.org/how-to-install-nodejs

Install npm: http://howtonode.org/introduction-to-npm

npm install request

npm install jquery
npm install jsdom

npm install connect

npm install express

npm install consolidate
```

Then you are ready to start the app. Go to the api-proxy/ directory and type:

```
node app.js
```

If everything was installed correctly you should get the following message on your console

```
Express server listening on port 3000
```

3.5 Using the tool

After completing the step above all you have to do is to open a browser and use http://localhost:3000/index.html as the address.

You should be able to see the resources of the starbucks API, and be able to expand them in order to see the different operations they support. By submitting the required input you should be able to get back the response sent by the server, acompanied with the corresponding success or failure code and the response headers.

Sphinx API Doc Generator

Sphinx is an open source documentation generation tool. It was originally created for the new Python documentation, and it is the framework of choice when it comes to documenting Python based projects and APIs. However the flexibility and versatility of Sphinx makes it suitable for documenting a wide range of other projects and software APIs including RESTful web APIs.

To create documentation using Sphinx, one must first create the necessary documentation files in the reStructuredText format. This is a simple textual format which allows the author to specify text formatting and layout information using text annotations and markup. In many ways, reStructuredText is similar to the markup languages used by documentation wikis and LaTeX. Once the documentation is created in the reStructuredFormat, these files are fed to the Sphinx documentation compiler which is capable of generating intelligent and professional looking project documentation in a number of output formats including HTML and LaTeX. The documentation of REST Coder, which you are reading right now has also been created using reStructuredText and compiled into HTML using the Sphinx compiler.

The REST Coder's Sphinx API Doc Generator operates in two steps.

- Given a REST API description (in JSON) as the input, auto generate API documentation in the reStructuredText format.
- Execute Sphinx compiler on the reStructuredText files created in the previous step to generate the final HTML based output.

The next few sections describe how to use the Sphinx API Doc Generator and what to expect from it as the output.

4.1 Using the Doc Generator

REST Coder's Sphinx API Doc Generator requires Python 2.7 and the Sphinx documentation engine to be installed. If these tools are already available, simply head over to the `bin` directory of the REST Coder installation and execute the script named `docgen.py` as follows.

```
./docgen.py -f /path/to/api/description.json -o /path/to/output/directory
```

This will load the specified API description file and create the Sphinx documentation at the specified output directory. If the output directory does not exist, it will be created by the tool.

It is also possible to load the input API description from a HTTP/S URL.

```
./docgen.py -u http://example.com/description.json -o /path/to/output/directory
```

By default the HTTP `OPTIONS` method will be used to pull the API description from the input URL. To use a different method use the `-m` flag.

To see the full list of command line options supported by the Sphinx Doc Generator, run `docgen.py` with the `-h` flag.

```
./docgen.py -h
```

4.2 Output Format

The Sphinx API Doc Generator creates a collection of reStructuredText (RST) files in the output directory and compiles them into HTML by default. Generated HTML files will also be stored in the same output directory and therefore after a successful run the output directory will contain both RST files and HTML files. You may manually execute the `sphinx-builder` command-line utility on the generated RST files to export the API documentation into a different format (e.g. LaTeX). If you want REST Coder to directly export the API documentation into a different (non-HTML) format, use the `-e` flag on the `docgen.py` utility.

4.3 What to Expect in the Generated Output?

The output created by the Sphinx API Doc Generator consists of an index page (index.rst/index.html) and a collection of other pages. The index page provides a description of the API and lists all its resources, operations and data models. These list entries link to other pages which provide in-depth details regarding the resources, operations and data types used by the API. Finally the index page lists all the base URLs (endpoints) of the web API and its licensing information.

The Doc Generator creates one page per resource defined in the API. These resource pages have multiple sections, one per operation. Each of these sections describes the API operation, its input and output types, HTTP status codes and also provides a sample Curl command to invoke the operation. Information regarding request/response media types, possible error conditions and sample JSON responses are also given where appropriate.

This tool also generates one page per complex data type defined in the API. These pages describe the data type and lists all the fields in each complex type. Resource pages link to these data model pages where appropriate. A data model page may link to other data model pages where necessary.

4.4 Project Metadata

Sphinx embeds some project metadata entries in each of the generated pages. These include project name, version and copyright information. The REST Coder's Sphinx API Doc Generator uses API metadata values extracted from the input API description to fill these entries in.

Project Metadata Entry	API Metadata Entry
Project Name	API Name
Project Version	API Version
Copyright Owners	API Owner

4.5 Sphinx Template

Sphinx uses a special template file to load various project metadata, global layout settings and other runtime options. The REST Coder's Sphinx Doc Generator is shipped with a default template file which control the look and feel of all the generated API docs. This file is named `sphinx_template.py` and it can be found in the `python-lib` directory of the RESTCoder distribution. You may modify this file manually to assert more control over the behavior and layout of the API docs generated by the Sphinx API Doc Generator tool.

CHAPTER 5

Python Client Stub Generator

The RESTCoder's Python client stub generator can be used to quickly and easily generate Python modules that can consume remote web APIs. The generated code handles data marshaling, unmarshaling, HTTP invocation and also in many cases error handling, thereby relieving the mashup and application developers from having to manually implement all that logic. If the input API description is properly documented, the Python code generated by the stub generator would have proper docstrings describing each of the auto-generated methods. By auto-generating code that masks all the complexities of communicating with a remote API, this tool greatly simplifies the process of developing mashups, desktop applications, command-line tools and webapps that rely on remote web services.

The following sections describe how to use the code generator and what to expect as its output.

5.1 Using the Python Code Generator

The Python client stub generator requires Python 2.7 to be installed. If Python 2.7 is already installed, simply head over to the `bin` directory of the RESTCoder installation and execute the script named `codegen.py` as follows.

```
./codegen.py -f /path/to/api/description.json -o mymodule.py
```

This will generate a Python module named `mymodule.py` which can be used as a client stub (proxy) to consume the remote API described in `description.json`.

It is also possible to load the input API description from a HTTP/S URL.

```
./codegen.py -u http://example.com/description.json -o mymodule.py
```

By default the HTTP `OPTIONS` method will be used to pull the API description from the input URL. To use a different method use the `-m` flag.

To see the full list of command line options supported by the Python code generator, run `codegen.py` with the `-h` flag.

```
./codegen.py -h
```

5.2 Generated Code

Python code generated by the client stub generator can be executed on any Python 2.7 runtime. The generated code is also highly backwards compatible with Python 2.6. The auto-generated code relies on following built-in Python modules.

- sys

- urlparse
- httplib
- json
- urllib

The generated code may also rely on the `api.py` module shipped with the RESTCoder distribution. This module can be found in the `python-lib` directory of RESTCoder.

The Python code generator, generates a separate Python class for each resource defined in the API description. The class is usually has the same name as the resource with the suffix `Client` appended to it. Operations of a resource are converted into methods of the corresponding generated class. Therefore if a particular input API description has a resource named `OrderManager` which has the operations `getOrder` and `submitOrder`, the code generator would generate a Python class named `OrderManagerClient` which has the functions `getOrder()` and `submitOrder()`. The input parameters of the two operations would be turned into Python method arguments and the output of the operations would be turned into return types.

The code generator would also generate separate classes for each of the data types defined in the input API description. Instances of these classes will be used as input arguments and return objects where appropriate. A separate set of static methods would be generated which handles serialization and deserialization of Python objects.

5.3 Media Types

As of now the Python code generator can only generate code for JSON based APIs (i.e. APIs that consume and produce JSON). If the input API description uses other media types, the code generator would create some place holder serialization/deserialization functions which simply raises the *NotImplementedError* exception.

```
def deserialize_VideoFeed_atomxml(obj):
  raise NotImplementedError
```

It is trivial to add support for other media types in the code generator tool. Simply implement a serializer extending the `AbstractSerializer` class of the `serializers.py` module. In this class, specify how to recursively convert a Python object into a byte string in the target media type and how to covert a byte string into a Python object. Refer the default `JSONSerializer` class in the same module for an example.

In situations where the target API supports multiple media types, you can force the code generator to stick to a single preferred media type when generating code by specifying the `-d` option.

```
./codegen.py -f /path/to/api/description.json -o mymodule.py -d json
```

5.4 Using the Generated Code

Simply import the generated module and use the client classes in the module to communicate with the remote API resources. The docstrings of the class methods would list all the input arguments accepted by each method.

```
def deleteVideo(self, videoId):
  """
  Args:
    videoId string

  Returns:
    An instance of the VideoFeedEntry class
  """
  query = ''
  conn = self.get_connection()
  ...
```

The generated methods will take care of marshaling input arguments, HTTP connection establishment and teardown and also unmarshaling response data. An example code snippet that uses a generated module is given below.

```
import youtube

if __name__ == '__main__':
  client = VideoSearchFeedClient()
  entries = client.getVideoFeed('api', alt='json', q='Google Glass').feed.entry
  print 'Found ' + str(len(entries)) + ' videos...'
  for item in entries:
    print item.title.t, '(Uploaded by:', item.author[0].name.t, ') - ', item.link[0].href
```

5.5 Error Handling

If the generated code encounters an error while invoking the target API, it would throw a `RemoteException`, which is a custom exception type defined in the generated module.

5.6 Using a Custom URL

By default, the generated code would communicate with the target API by making a HTTP connection to the base URL specified in the API description. But in some cases it would be required to communicate with the API using a custom URL (some gateway or proxy URL). To specify a custom URL, specify the `endpoint` argument in the constructor of the corresponding resource client.

```
import youtube

if __name__ == '__main__':
   client = VideoSearchFeedClient(endpoint='http://my.custom.url')
   ...
```

5.7 Debug Mode

The auto-generated code supports a special debug mode. When executed in this mode, the client code prints all the requests and responses exchanged with the backend API. To enable the debug mode, simply pass `True` to the `debug` argument of the constructor of the corresponding resource client.

```
import youtube

if __name__ == '__main__':
   client = VideoSearchFeedClient(debug=True)
   ...
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search