Automated Creation of Navigable REST Services Based on REST Chart

Li Li, Tony Tang, and Wu Chou
Shannon IT Lab, Huawei, Bridgewater, New Jersey, U.S.A.
Email: {li.nj.li, tony.tang, wu.chou}@huawei.com

***Abstract*—As REST architectural style gains popularity in various areas, there is an acute need for a REST toolkit that can automate the process of generating service implementations from the service descriptions. Despite that we can generate SOAP service implementations from WSDL files, there is a key distinction between REST services and WS-* services: the former is navigable by hypertext whereas the later is not. Conventional REST toolkits tie the REST API navigability with service actions, such that the navigability can only be achieved at the expense of additional data models and programs. To address this problem, this paper proposes Navigation-First Design to make a REST API navigable before implementing any service actions. A Java REST toolkit has been developed to realize the benefits of this approach through automated generation of JAX-RS compliant Java resource and message classes based on the Hierarchical REST Chart, a Petri-Net based service design framework for REST API. The toolkit can transform REST Chart XML files into a navigable REST API prototype, build and deploy it without the developer writing one line of code. The preliminary experiments show that the approach is feasible and promising.**

***Index Terms*—REST API, REST Chart, JAX-RS, Petri-Net, navigation-first design, code generation**

I. Introduction

In recent years, the REST architectural style [1], [2] is widely applied in various areas, including real-time communications [3], [4], Cloud computing [5], and software-defined networking (SDN) [6]. It is an efficient and flexible approach to access and integrate large-scale complex and distributed systems. However, as the popularity of REST services grows, there is an acute need for a service modeling framework and toolkit with a standard machine-readable service description language for REST APIs - analogous to WSDL [7], [8] and SOAP [9] toolkits that automate service development process for WS-* style web services. Such a modeling framework and toolkit is critical to the rapid design, development, and deployment of REST services. Moreover, it can enable the following important features for REST:

- Contract-First design: a REST API can be designed and conveyed accurately to its stake holders, in which the design can be verified automatically against REST constraints, such that problems or disagreements can be identified and resolved before expensive service implementations.
- Meta-programming: programs for client and server can be generated automatically or semi-automatically from the description language that characterizes the REST API. This not only speeds up the development process, but also reduces code size and inconsistencies between the design and implementation of the REST API.
- Meta-interoperability: service description language can be used to detect and cope with functional changes in a REST API without testing its implementation. This cability can significantly reduce the cost of integrating a large complex system by catching the problems at the design time.
- Automated testing: coverage and workload test cases can be derived from the service description of the REST API which can be agnostic and independent of its implementation.

Since 2009, several service description languages, including REST Chart [10], WADL [11], RAML [12], Swagger [13], RSDL [14], API-Blueprint [15], SA-REST [16], ReLL [17], RADL [18], and RDF-REST [19] are developed for REST API. However, they are still under development and none of them is yet being standardized.

Some REST toolkits have also been developed for these service description languages. These toolkits can take a concrete REST service description and generate skeleton code in some programming language, for example Java, which partially implements the described REST services. The generated skeleton code can be modified by developers to complete the service implementation, to be compiled and then deployed to a HTTP server.

Although this workflow for REST service is very similar to the contract-first development process for WS-* services aided by some SOAP toolkits, the two workflows have a key distinction: a REST API is driven by hypertext whereas a WS-* service is not. A REST API is implemented by distributed resources and each of which maps the incoming hypertext resource representation to outgoing resource hypertext

Manuscript received December 29, 2014; revised May 20, 2015.

doi: 10.12720/joams.4.5.385-392

representation according to the action performed on the resource state, such as to create a computer network, retrieve a conference, update a virtual machine, or delete a participant. According to the 7 REST constraints [10], the ability for a client to navigate the resources from an entry point is a requirement for a REST API and this requirement is independent of the resource representations. To implement this navigability in a REST API typically requires implementation of the actions performed on resource states, as different actions can produce different outgoing representations. However, such an approach, despite being quite complete, is much more costly because in addition to the REST service description, it requires development of backend data models and programs to manipulate the resource states in a typically 3-tiered REST service architecture. Furthermore, such approach forces us to commit to a data model and programs that may need to be updated or even abandoned, as the REST API under development often changes and updates frequently.

To address these problems, we propose a new approach to REST API design and development called Navigation-First Design, whose main idea is to make a REST API navigable before it performs any actions. This approach has several benefits:

- Automated code generation tools can be used to rapidly create a lightweight REST API prototype from a REST service description without any programming work for the developers.
- The working REST API prototype allows the stake holders (developers and users for example) of the REST API to interactively study and test the resource connections, resource representations, and service allocations through resource navigation.
- The code of the prototype can be reused to speed up the development process of the REST API.

To realize these benefits, this paper describes a REST toolkit that transforms a REST Chart to Java source code that supports navigation without action. The toolkit also provides ant script to automatically compile and deploy the generated Java code, such that with a few clicks in an IDE, a developer can produce a navigable REST API prototype without writing one line of code.

The rest of this paper is organized as follows. Section II reviews the related work. Section III reviews the basic REST Chart structure for modeling REST APIs. Section IV gives an overview of the toolkit. Section V focuses on the REST service generation workflow. Section VI discusses the prototype implementation, and we summarize our contributions with section VII.

II. Related Work

WADL [11] is an early effort to describe REST services, followed by RAML [12], Swagger [13], RSDL [14], API-Blueprint [15], SA-REST [16], ReLL [17], REST Chart [10], RADL [18], and RDF-REST [19]. All of them are encoded in some machine-readable languages, such as XML, and most of them are standalone documents, except a few of them, such as SA-REST, are intended to be embedded within a host language, such as HTML.

RAML is a YAML language that organizes a REST API as tree whose nodes are URI templates or references. The root of the tree identifies the entry point to the REST API and the children identify the resources reachable from the parents. Each URI may be associated with some access methods that define the input and output representations. While RAML offers a minimalist structure and several interesting design primitives, such as inline documentation, resource traits and types, it could lead to inadvertent violation of the REST constraints [10] by exposing a list of fixed resource locations. Also, RAML does not seem to have a way to tie a hyperlink in hypertext representations with a URI template in the REST API URI tree. Without these ties, it would be difficult for a REST client to know the exact method in order to access a particular hyperlink during the navigation.

Swagger can describe a REST API in either YAML or JSON, and its descriptive structure is very similar to RAML, except with a different set of primitives. For this reason, it has the same problems as RAML.

RSDL is a XML language that organizes a REST API around a list of <resource> elements. Each <resource> element may contain a <location> element that defines its URI, a <link> element that links it to other resources, and a <method> element that defines the request and response. This resource centered design could inadvertently violate the REST constraints [10] by exposing a fixed set of resource locations to the clients. Moreover, there are no ties between the <link> elements, the <method> elements, and the responses, by which a hyperlink in a response hypertext can point to its access method.

RADL is a XML language that also organizes a REST API around <resource> elements. Each <resource> element may contain a <uri> element that defines its location and a <interface> that defines the access methods. The request and response of a method are defined by <document> elements, which may contain <link> elements that point to other <document> elements. Like RSDL, this resource centered design could lead to fixed resource locations. In this design, even if a client knows the interface of a resource, it may not know the exact method in the interface to access a hyperlink of a document, should there be two or more methods in the interface.

Despite many improvements over the years, most recent REST service description languages, except REST Chart and SA-REST, still include some form of resource constructs that could lead to fixed resources locations, relations and interfaces, which is a violation of Roy Fielding's REST constraints R3-R5 [10]. Furthermore, none of the REST toolkits we are aware of supports Navigation-First Design, as the conventional approaches regard navigation as the result of actions.

III. REST Chart Overview

REST Chart is proposed in [10] to design and describe REST API without violating the REST principles [1],

[10]. One of the key REST principles states that any client of a REST API should be driven by nothing but hypertext. This principle requires that a REST API should guide a client through hypertext without relying on any out-of-band information that could restrict the REST API's freedom to reorganize its resources or change their representations. It suggests that a REST API should spend almost all its descriptive effort in defining the media types (name, structure, and processing rules of data formats) that the resources accept and produce, without exposing any fixed resources names, locations, types or hierarchies.

Following these guidelines, REST Chart models a REST API as a Colored Petri-Net [20], [21], where each place in the Petri-Net denotes a media type and each token denotes a resource representation. A place only admits tokens of the same media type, but the transitions are "color blind" and it can be fired by tokens of any media type. Under this model, REST Chart uses the transitions to connect the media types with hyperlinks, and these connections define the processing rules of the hyperlinks. The places and transitions collectively answer the critical questions for a hypertext-driven REST client: 1) where are the hyperlinks in a hypertext; 2) what service does a hyperlink provide; 3) how to interact with a hyperlink; and 4) what hypertext will the interaction produce.

Fig. 1 illustrates a basic REST Chart that describes a typical hypertext-driven request-response interaction. The REST Chart consists of one transition that connects two input places and one output place that are labeled by media types. The "login" places is called "server place" as its token is generated by the REST server, and the "credential" place is called "client place" as its token is generated by the REST Client. This REST Chart indicates that a client can follows a hyperlink L from the "login" place to the "account" place, which is another server place, when the client creates a token in the "credential" place. The transition models the interaction with a remote resource identified by L. However, unlike the other service description languages, REST Chart has no <resource> element or fixed URI, as it does not model a REST API as a set of resources.

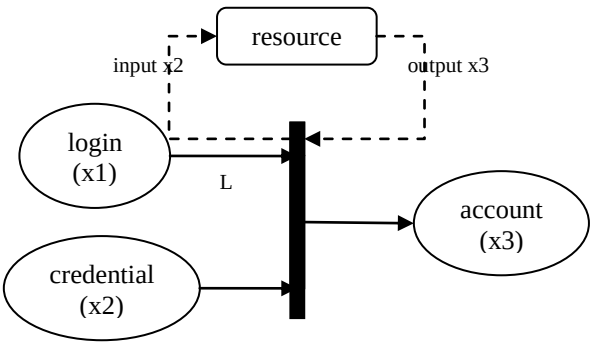

Figure 1. Example of a basic REST Chart.

REST Chart uses a XML dialect [10] to encode a Colored Petri-Net such that it is machine readable and extensible, and moreover, it can be validated by XML Schemas. The REST Chart XML for the REST Chart diagram in Fig. 1 is shown in Listing 1. The places are defined by the nested <representation> elements that admit two media types, XML defined by XML Schema [22] and JSON defined by JSON Schema [23]. The hyperlink L is defined by a <link> element that contains a <rel> element, a URI [24] that identifies the service and a <href> element, a URI Template [25] that identifies the possible locations of the resource. The transitions are defined by the <transition> elements which bind the <link> element to the HTTP protocol [26]-[28].

A REST Chart typically consists of many places and transitions, and a place in one REST Chart contain another REST Chart to form Hierarchical REST Chart. Fig. 2 and Fig. 3 show a Hierarchical REST Chart that break a large REST API into two modules that can evolve independently.

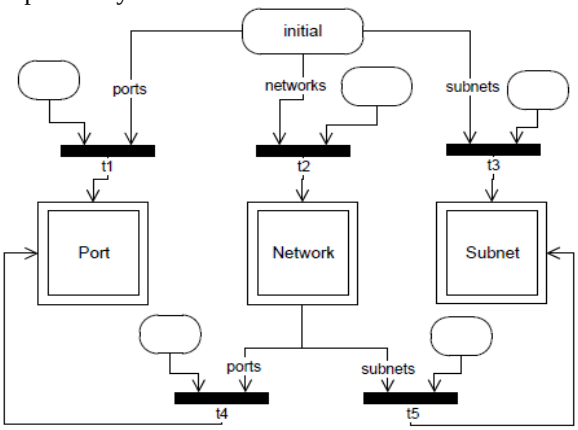

Figure 2. Top-level REST Chart for SDN NBI

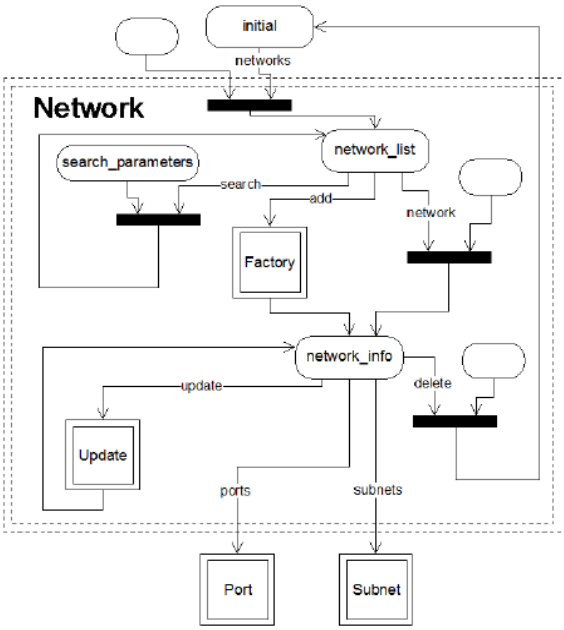

Figure 3. Nested REST Chart for SDN NBI

IV. REST CHART TOOLKIT

The overall workflow of the REST Chart Toolkit (RC Toolkit) is outlined in Fig. 4 which transforms a REST Chart into a navigable REST API prototype in Java. The

workflow consists of three stages: 1) service generation; 2) service build; and 3) service deployment.

In **stage 1**, the Java Source Generation module accepts a REST Chart XML file and the relevant XML Schema files, and produces three types of artifacts:

- JAX-RS [29] compliant Java Resource Classes that correspond to the resources of the REST API.
- Java Message Classes that correspond to the XML Schemas used by the REST API.
- XML Navigation Messages that contain hyperlinks between the resources.

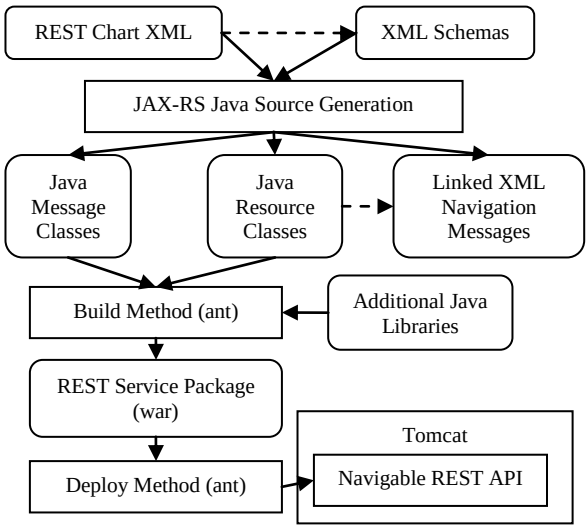

Figure 4. Overall workflow of RC Toolkit

In **stage 2**, the Build Method module compiles the generated Java classes and additional Java libraries (jar files) into a Web Application Archive (war) package that can be easily stored, transmitted, and deployed to a Web application container, like Apache Tomcat.

In **state 3**, the Deploy Method module deploys the war file into a Tomcat server, such that the REST services can be navigated using any Web browser or test tool.

To start the navigation, a user visits the entry URI of the deployed REST services to obtain a response message in either XML or JSON which contains hyperlinks that can be followed. The user chooses the appropriate HTTP 1.1 method according to the REST Chart, including GET, POST, PUT and DELETE, to interact with hyperlinks and navigate to any resource in the REST API.

This 3 stage process are automated to the extent that a developer can click a few buttons and configure a few parameters in Eclipse IDE to create a navigable REST API from a REST Chart without writing one line of Java code. The following sections focus on stage 1, the service creation stage.

V. REST Service Generation

This stage consists of three related workflows as depicted in Fig. 5. The first workflow generates the Java Message Classes from the XML Schemas. The second workflow generates the XML navigation messages, and the third workflow generates Java Resource Classes.

A. *Java Message Class Workflow*

The first workflow uses a JAXB [30] tool xjc [31] to transform XML schemas to Java Message Classes that can be used to deserialize XML requests into Java objects and serialize Java objects to XML responses. This workflow is relatively independent of REST Chart.

B. *Navigation Message Workflow*

In the second workflow, the REST Chart XML is parsed into a DOM tree, from which an internal REST Chart data structure is built. The XInstance [32] tool is used to produce sample XML documents from the XML schemas. These documents are called link-free navigation messages as they contain random hyperlinks that are irrelevant to the REST Chart. The Link Replacement module replaces the random hyperlinks in those XML messages by the valid hyperlinks in the REST Chart to produce Linked Navigation Messages for each possible response of the REST API. The result of this procedure is illustrated in Fig. 6.

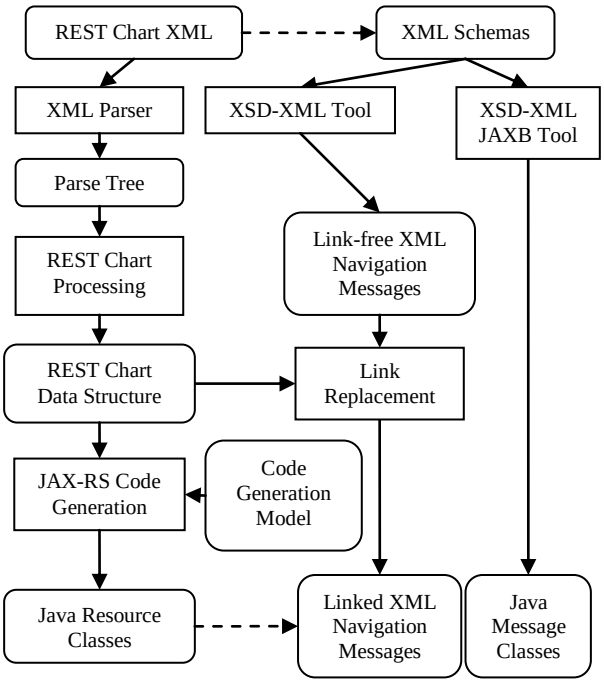

Figure 5. Service generation workflow

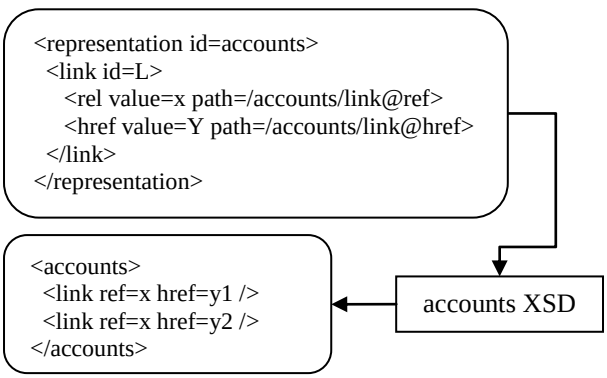

Figure 6. Link replacement in navigation message

In Fig. 6, a REST Chart <representation> element declares a hyperlink L with elements L.<rel>=x and

L.<href>=Y. The locations of these values in XML messages are specified by XPath [33]. When a random XML <accounts> message is generated from the XSD, the workflow locates the <link> elements in the message based on the XPath. For each located <link> element, its attributes (rel, href) are replaced by (x, expansion(Y)).

The expansion is necessary because Y is a URI template that contains variables and cannot be navigated by any REST client. There are two types of variables in a URI template: 1) authority variables (e.g. {a}) that expand to domain names or IP addresses, and 2) path variables (e.g. {u}) that expand to identifiers. As the REST services are not yet deployed at this stage, the workflow only expands the path variables, and the expansions of authority variables are carried out at runtime by the generated Java methods.

For example, for a URI template Y=http://{a}/users/{u}/login, the workflow will expand the path variable {u} in Y to random identifiers u1 and u2 that identify two distinct resources:

y1=http://{a}/users/u1/login
y2=http://{a}/users/u2/login

The reason that random identifiers will work is because when y1 and y2 are dispatched to a generated Java Resource Class, the class will not use these variables to perform any actions.

C. Java Resource Class Workflow

The third workflow and the second workflow share the internal REST Chart data structure. But the third workflow generates Java Resource Classes based on a Code Generation Model that defines the mappings from REST Chart elements to Java Resource Classes as depicted in Fig. 7. The Code Generation Model maps each <transition> element in a REST Chart to a Java Resource Class, and the elements associated with a <transition> are used to determine the variables in the Java Resource Class Template as shown by the dashed arrows.

The Java package name is derived from the target_namespace of the REST Chart. Each REST Chart <link> element is used to derive the {Path}, {ClassName} and {Method} variables of the Java Resource Class. Two <link> elements with the same href but different rel will become two Java methods in the same Java Resource Class. The <control> element is used to derive the {HTTPMethod}. The media type and XSD of the <input> element are used to derive the {MediaTypeX} and {MessageClass} variables respectively. The media type of the <output> element is used to derive the {MediaTypeY} variable, while the XSD is used to construct the Response object.

As the XSD files are transformed to Java Message Classes by the first workflow in subsection A, this second workflow knows which Java Message Class is used for which Java method based on the <transition> element that ties all the relevant elements together. It also knows if a Java Message Class is a parameter or the return type to which the Java method is based on, if the XSD is associated with the <input> or the <output> element of the <transition>.

The workflow iterates over the <transition> elements and creates corresponding Java Resources Classes following the above mappings. Its time complexity is *O(N)* where *N* is the number of transitions in the REST Chart.

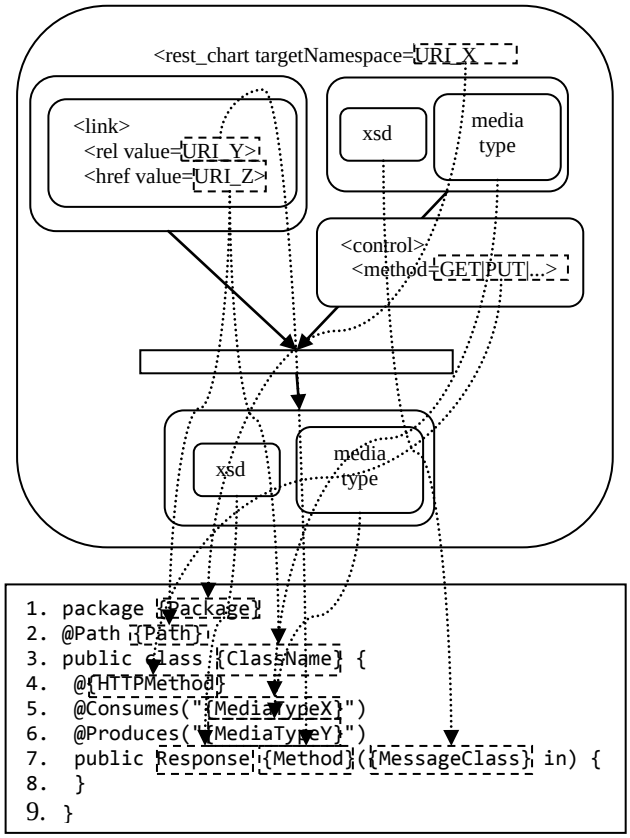

Figure 7. Map REST chart to java resource class

To elaborate this workflow, we illustrate how the following REST Chart XML (Listing 1) for Fig. 1 is used to derive a Java Resource Class (Listing 2), where the JAX-RS annotations are prefixed by @ and the code fragments between {} are omitted for clarity.

```
1.  <rest_chart xmlns=... target_namespace=...>
2.   <representation id=login>
3.    <link id=L>
4.     <rel value=http://www.bank.com/login />
5.     <href value=http://{a}/users/{u}/login />
6.    </link>
7.    <representation id=login_xml
8.     media_type=application/xml>
9.     <schema location=login.xsd />
10.   </representation>
11.   <representation id=login_json
12.    media_type=application/json>
13.    <schema location=login.jsd />
14.   </representation>
15.  </representation>
16.  <representation id=credential>
17.   <representation id=credential_xml
18.    media_type=application/xml>
19.    <schema location=credential.xsd />
20.   </representation>
21.   <representation id=credential_json
22.    media_type=application/json>
23.    <schema location=credential.jsd />
24.   </representation>
```

```
25.  </representation>
26.  <representation id=account>
27.   <representation id=account_xml
28.    media_type=application/xml>
29.    <schema location=account.xsd />
30.   <representation>
31.   <representation id=account_json
32.    media_type=application/json>
33.    <schema location=account.jsd />
34.   </representation>
35.  </representation>
36.  <transition>
37.   <input>
38.    <representation ref=login link=L />
39.   </input>
40.   <input>
41.    <control method=POST />
42.    <representation ref=credential />
43.   </input>
44.   <output>
45.    <control status=201 />
46.    <representation ref=account />
47.   </output>
48.  </transition>
49. </rest_chart>
```

Listing 1: REST Chart XML

```
1. package {Package}
2. {imports}
3. @Path("/users/{u}/login")
4. public class LoginResource {
5.   @Context
6.   {context_variables}
7.   {auxiliary_methods}
8.   @POST
9.   @Consumes("application/xml")
10.  @Produces("application/xml")
11.  public Response loginXML({MessageClass} in) {
12.     {load_navigation_xml} }
13.  @POST
14.  @Consumes("application/json")
15.  @Produces("application/json")
16.  public Response loginJSON({MessageClass} in) {
17.     {load_navigation_json} }
18. }
```

Listing 2: Java Resource Class derived from Listing 1

TABLE I. MAPPINGS FROM REST CHART TO JAVA

	Chart	comments
1	1	{Package}←target_namespace
2	-	Java libraries
3	5	@Path←//link[@id=L]/href
4	5	Login←//link[@id=L]/href
5	-	Java Context annotation
6	-	Obtain request context
7	-	Private auxiliary methods
8	41	@POST←//input/control@method for XML response
9	42,18	XML credential request
10	46,28	XML account response
11	19,29	{MessageClass}←credential.xsd Response←account.xsd
12	-	Load XML navigation message
13	41	@POST←//input/control@method for JSON response
14	42,21,22	JSON credential request
15	46,31,32	JSON account response
16	19,29	{MessageClass}←credential.xsd Response←account.xsd
17	-	Transform XML navigation message to JSON

Table I explains the relation between each line of Java Resource Class and the REST Chart XML. The line numbers of REST Chart are ordered by the direction the workflow searches the REST Chart.

D. Navigation Message Workflow

Each method in a generated Java Resource Class follow a common workflow as illustrated in Fig. 8 to return navigation messages in appropriate media types based on content-negotiation. For different media types, different procedure is generated, such as {load_navigation_xml} for XML and {load_navigation_json} for JSON, as shown in Listing 2.

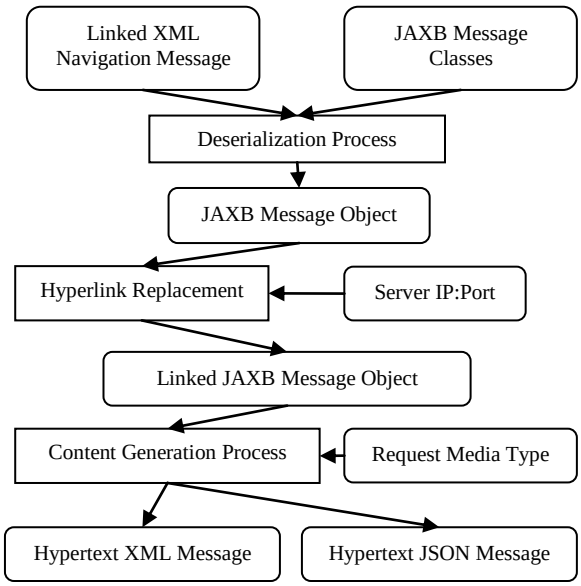

Figure 8. Common navigation response workflow

In this workflow, each time a request is dispatched to the Java method, it loads the corresponding Linked XML Navigation Message, which is generated by the workflow in section B and saved on disk, into the memory, and transforms it to a (JAXB) Java Message Object using the (JAXB) Java Message Class generated by the workflow in section A. The authority variable in a URI template is then replaced by the server IP address and port to produce Linked Message Object. For example: http://{a}/users/u1/login could become http://localhost:8080/users/u1/login. Depending on the requested media types, the method will transform a linked message object to either XML or JSON Java Response to ensure the consistency between two different representations.

Although it is an overhead that the method loads the navigation message upon each request, it has the flexibility to allow the navigation messages to be changed between requests without restarting the REST services.

VI. PROTOTYPE AND EXPERIMENT

The proposed RC Toolkit has been implemented in Java and tested on several REST Charts. A user can use either the Eclipse IDE or command-line to configure and run an ant script that contains targets to generate JAX-RS Java source code, to build them and deploy them to a Tomcat server.

Once the REST API is deployed successfully, a user can use Postman [34], a REST client extension to the Chrome browser, to navigate the REST API starting from an entry URI. The following screenshot (Fig. 9) shows a navigation response from a SDN Northbound REST API generated and deployed to a Tomcat server by RC Toolkit. The navigation XML messages contain random data but fully functional hyperlinks. The hyperlinks are highlighted by Postman and they can be followed by clicking. To switch between XML and JSON, a user just clicks the buttons on top of the screen.

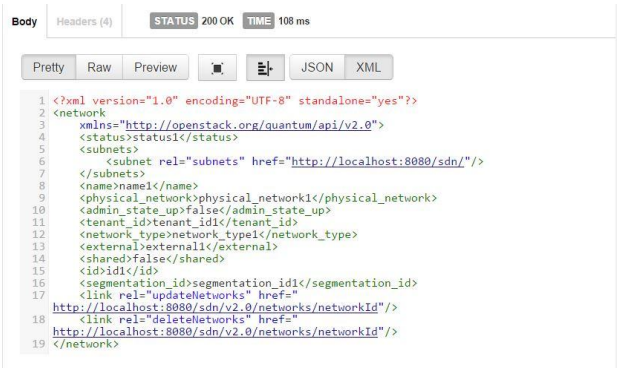

Figure 9. Screenshot of testing a generated REST API

A user can also send HTTP POST or PUT request to a URI by providing some random XML or JSON body. The REST API will return navigational response messages with functional hyperlinks as well.

Two REST APIs were tested: the SDN REST API defined by a Hierarchical REST Chart and 4 nested REST Charts, and the Coffee REST API defined by one REST Chart. For each REST Chart, the total numbers of places, transitions, XML schemas, Java Message Classes, Java Resource Classes, and Navigation Messages are summarized in Table II. The performance of the REST Service Generation stage on these two REST Charts is based on the average of over 6 runs on a 32-bit Windows 7 notebook computer (Intel Core i5 dual core 2.67Ghz and 4GB RAM), and it is summarized in the last two rows of Table II.

TABLE II. PERFORMANCE OF JAVA CODE GENERATION

Measurements	SDN	Coffee
Place	19	9
Transition	22	7
XSD	15	11
Navigation Message	15	11
Java Message Class	64	49
Java Resource Class	16	6
avg (second)	4	3.3
std	1.2	0.8

The experiments show that the workflow scales well when the number of transitions increases. The SDN REST Chart had 3 (22/7) times of number of transitions and 2 (19/9) times of number of places as the Coffee REST Chart, but its workflow time is only 4/3.3 = 1.2 times of Coffee, with relatively stable performance.

We also tested the performance of the generated SDN REST API deployed to Tomcat (JVM 64 bit, 1.7.0_67 Tomcat 7.0.55) running at a Linux CentOS 6.5 server (Intel Xeon 5645-six core [2.4GHz]) x2 24 logical core, 12 physical core, 132GB RAM) using JMeter [35] running at 64-bit Windows 7 Professional machine (Intel core i7-2600 CPU@3.4GHz 4 physical core, 8 logical core, 16 GB RAM). The server and client machines are connected by LAN. JMeter was used to simulate the concurrent REST clients accessing a Tomcat server, while each JMeter thread repeats 17 predefined requests 100 times. 4 performance tests were conducted without any optimization to the network, Tomcat, or Java:

- Test 1: 10 Tomcat instances and 1 JMeter thread for each Tomcat.
- Test 2: 10 Tomcat instances and 5 JMeter threads for each Tomcat.
- Test 3: 20 Tomcat instances and 1 JMeter thread for each Tomcat.
- Test 4: 20 Tomcat instances and 5 JMeter threads for each Tomcat.

Tests 1 and 2 were each repeated 5 times, while tests 3 and 4 were each repeated 3 times. The average response time and throughput (#request/s) reported by JMeter are summarized in the table below.

TABLE III. PERFORMANCE OF GENERATED REST API

Tests	Total requests	Average response time (ms)	Average throughput (responses/s)
1	17,000	21.2	433.9
2	85,000	57.6	805.3
3	34,000	33.6	422.6
4	170,000	109.6	763.8

The average response time was less than 60ms when the workloads were moderate in the first three tests. The average response time increased to 100ms when the workload was high at the 4th test. The system throughputs ranged from 21 to 43 responses/second per Tomcat. The tests show that the automatically generated REST API is efficient and it can support concurrent testing by a team of users.

VII. CONCLUSIONS

The contributions of this paper are summarized below:

- We proposed Navigation-First Design approach to reduce dependencies of REST API on backend data models and programs for rapid prototyping.
- We developed a REST toolkit based on REST Chart that combines REST service generation, build, and deployment into a seamless workflow.
- We developed a method that generates JAX-RS compliant Java Resource and Message Classes that allows a REST API to be navigated without performing any service actions.
- The generated Java code automatically supports many media types defined in REST Chart based on content-negotiation.

For future work, we plan to enrich the Java binding of REST Chart and enhance the process of navigation messages generation. We also plan to integrate code generation process with code editing process, such that

programs created by developers and tools can be integrated seamlessly to enable a rapid development cycle.

REFERENCES

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] L. Richardson and S. Ruby, *RESTful Web Services*, 1st ed. Sebastopol, CA: O'Relly, 2007.

[3] Twilio REST API. [Online]. Available: http://www.twilio.com/docs/api

[4] GSMA OneAPI. [Online]. Available: http://www.gsma.com/oneapi/voice-call-control-restful-api/

[5] Amazon Simple Storage Service REST API. (March 2006). [Online]. Available: http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html

[6] Floodlight REST API. [Online]. Available: http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API

[7] E. Christensen, et al. (eds). (March 2001). Web Services Description Language (WSDL) 1.1, W3C Note. [Online]. Available: http://www.w3.org/TR/wsdl

[8] R. Chinnici, et al. (eds). (June 2007). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Recommendation. [Online]. Available: http://www.w3.org/TR/wsdl20/

[9] M. Gudgin, et al. (eds). (April 2007). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation. [Online]. Available: http://www.w3.org/TR/soap12-part1/

[10] L. Li and W. Chou, "Design and describe REST API without violating REST: A petri net based approach," in *Proc. ICWS 2011*, Washington DC, USA, July 4-9, 2011, pp. 508-515.

[11] M. Hadley. (August 2009). Web Application Description Language. W3C member Submission. [Online]. Available: http://www.w3.org/Submission/wadl/

[12] RAML Version 0.8. [Online]. Available: http://raml.org/spec.html

[13] Swagger 2.0. [Online]. Available: https://github.com/swagger-api/swagger-spec

[14] J. Robie, R. Cavicchio, R. Sinnema, and E. Wilde. (2013). RESTful Service Description Language (RSDL). Describing RESTful Services Without Tight Coupling, Balisage: The Markup Conferenc. [Online]. Available: http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html

[15] API Blueprint Format 1A revision 7. [Online]. Available: https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md

[16] K. Gomadam, A. Ranabahu, and A. Sheth. (April 2010). SA-REST: Semantic Annotation of Web Resources. [Online]. Available: http://www.w3.org/Submission/SA-REST/

[17] R. Alarcon and E. Wilde, "Linking data from RESTful services, LDOW 2010," April 27, 2010, Raleigh, North Carolina.

[18] J. Robie. (2014). RESTful API Description Language (RADL). [Online]. Available: https://github.com/restful-api-description-language/RADL

[19] Pierre-Antoine Champin, RDF-REST: A Unifying Framework for Web APIs and Linked Data. Services and Applications over Linked APIs and Data (SALAD), workshop at ESWC, May 2013, Montpellier (FR), France, pp.10-19.

[20] T. Murata, "Petri nets: Properties, analysis and applications," in *Proc. the IEEE*, vol. 77, no. 4, pp. 541-580, April 1989.

[21] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. New York, NY: Springer, 2008, ch. 4.

[22] H. S. Thompson, et al (ed). (October 2004). XML Schema Part 1: Structures Second Edition, W3C Recommendation. [Online]. Available: http://www.w3.org/TR/xmlschema-1/

[23] JSON Schema and Hyper-Schema. [Online]. Available: http://json-schema.org/documentation.html

[24] T. Berners-Lee, et al. (January 2005). Uniform Resource Identifier (URI): Generic Syntax, Request for Comments: 3986. [Online]. Available: https://tools.ietf.org/html/rfc3986

[25] J. Gregorio, et al. (March 2012). URI Template, Request for Comments: 6570. [Online]. Available: https://tools.ietf.org/html/rfc6570

[26] R. Fielding, et al (eds). (June 2014). Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, IETF RFC7230. [Online]. Available: http://tools.ietf.org/html/rfc7230

[27] R. Fielding, et al (eds). (June 2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, IETF RFC7231. http://tools.ietf.org/html/rfc723

[28] M. Belshe, et al (eds). (October 2014). Hypertext Transfer Protocol version 2. [Online]. Available: https://tools.ietf.org/html/draft-ietf-httpbis-http2-15

[29] S. Pericas-Geertsen and M. Potociar, JAX-RS: Java™ API for RESTful Web Services, Version 2.0 Final Release, May 22, 2013.

[30] K. Kawaguchi, S. Vajjhala, and J. Fialli, The Java™ Architecture for XML Binding (JAXB) 2.2, Final Release, December 10, 2009.

[31] Java TM Architecture for XML Binding Binding Compiler (xjc). Implementation Version: 2.2.4. [Online]. Available: https://jaxb.java.net/2.2.4/docs/xjc.html

[32] XSInstance. Generating Sample XML for given XMLSchema. [Online]. Available: http://code.google.com/p/jlibs/wiki/XSInstance

[33] A. Berglund, et al (ed). (December 2010). XML Path Language (XPath) 2.0 (Second Edition), W3C Recommendation. [Online]. Available: http://www.w3.org/TR/xpath20/

[34] Postman. [Online]. Available: https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en#detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en

[35] Apache JMeter. [Online]. Available: http://jmeter.apache.org/

Dr. Li Li received his Ph.D. in computer sciences from University of Alabama at Birmingham, USA in 1995, and M.S. in computational linguistics from Huazhong University of Sciences and Technology, China in 1987. He joined Huawei Shannon IT Lab in 2012 and his current research interest includes web services, cloud computing and software-defined networking. He has published over 60 conference and journal papers and 1 book on Artificial Intelligence. He currently holds 6 US patents. Dr. Li is a member of IEEE and ACM, and he was the editor of 2 ISO/ECMA CSTA standards and made significant contributions to W3C WS-RA standard suite.

Mr. Tony Tang received his M.S. in electrical and computer engineering from Northeastern University, USA in 2008. He has over 5 years of working experience in the IT industry and he currently works as a contractor for Huawei.

Dr. Wu Chou graduated from Stanford University in 1990 with four advanced degrees in science and engineering. He is VP, Chief IT Scientist, and Head of Huawei Shannon (IT) Lab, USA. He joined AT&T Bell Labs after obtaining his Ph.D. degree in electrical engineering, and he continued his professional career from AT&T Bell Labs to Lucent Bell Labs and Avaya Labs before joining Huawei. He published over 150 journal and conference papers, holds 35 US and international patents with many additional patent applications pending.
Dr. Chou is an IEEE Fellow. He served as editor and area expert for multiple international standards at W3C, ECMA, ISO, ETSI, etc. He was an editor of IEEE Transactions on Services Computing (TSC), IEEE TSC Special Issue on Cloud Computing, IEEE Transactions on Audio and Language Processing, and Journal of Web Services Research.