Automatic Multi Language
Program Library Generation for REST APIs

Thomas Steiner
Master Thesis
August 2007

Advisor
Prof. Dr. Jacques Calmet
Institute for Algorithms and Cognitive Systems
University of Karlsruhe (TH)

Supervisor
Patrick Chanezon
API Evangelist
Google Inc.
Statement of originality

Hereby I declare this thesis to be the work of my own, written independently. All sources and references are correctly cited with complete references to their sources.

Thomas Steiner
Karlsruhe, August 3rd, 2007
Dedication

Dedicated to my wife Laura whose support and patience made this project possible, to my first, still unborn child that kicks its legs in joy when I sing for it, and finally to my sempai and colleague Patrick who helped me become a passable engineer, and who taught me one of the most fundamental lessons in life: doing a great job is important, but family is even more important.

Acknowledgments

I would like to thank the following persons (appearance in no particular order):

Prof. Dr. Jacques Calmet, University of Karlsruhe
Patrick Chanezon, Google Inc.
Marc Hadley, Sun Microsystems
John Musser, ProgrammableWeb.com
Eduard Pelegri-Llopart, Sun Microsystems
Dion Almaer, Ajaxian.com, Google Inc.
Stefan Tilkov, InfoQ.com
Bill Graham, CNET.com
Abstract

In the recent years Representational State Transfer (REST) APIs have gained astonishing popularity among both Web Service providers and Web Service consumers. REST is considered simple, lightweight, and by many is also thought of as the better alternative to SOAP-based solutions that used to dominate the Web Service market before the rise of REST. In the blogosphere there is still a passionate discussion going on whether the SOAP vs. REST war is over or not. We will not presume to present a response to this question, however, given the matter of fact that most of the APIs released these days are REST-based, and given our previous experiences with SOAP, beyond doubt we are biased towards REST. One of the promises of SOAP is the availability of a broad spectrum of SOAP-related tools and libraries. These facilities create the impression that from an existing SOAP Web Service description the necessary client proxy code is just one click away. Unfortunately in practice the devil is in the details. SOAP is not SOAP per se, there is SOAP 1.x, there is SOAP 2.0, and there are more than two, of course incompatible ways for fundamental things like parameter encoding styles. Short, what at the first glance sounds like a good idea, sometimes later turns out to be a nightmare of library incompatibilities and differing implementations of API providers.

Driven by these frustrations some people have come up with new suggestions for simplified REST description languages. In this thesis we will first motivate the reasons behind these suggestions, and afterwards have a detailed look at each proposition of such REST description language. This analysis will be followed by the careful selection of one of the available Web Service description languages. Our language of choice will be Sun engineer Marc Hadley's Web Application Description Language (WADL). We will then present a design document that describes the requirements of a WADL editor and a WADL compiler Ajax application. This application shall be able to analyze sample requests for REST APIs, and based on this analysis suggest a WADL description, that can then in turn serve as an input for a WADL compiler. The goal of the compiler shall be to create source code in various programming languages for existing WADL descriptions. The design document will be followed by a detailed description of the solutions that found its way into the implementation of the previously outlined application. Some major challenges like code generation and type estimation will be treated in more detail. In addition to that an interesting extension for the initial approach (sample request analysis) based on the "dog food" principle will be presented. This extension performs an additional response analysis in order to refine the quality of the auto-generated WADL. The thesis will be completed by a breakdown of the achieved results, a summary, and finally will end with some prospects for further improvement of the application.

Tags: rest, restful, api, web service, wadl, rest describe, rest compile, rest describe & compile, compiler, editor, service description

Resources: this thesis is accompanied by an application called "REST Describe & Compile" which is available online:

- [http://tomayac.de/rest-describe/latest/RestDescribe.html](http://tomayac.de/rest-describe/latest/RestDescribe.html) (live demo)

In addition to that there are some introductory screencasts available on YouTube:

- [http://youtube.com/user/tomayac](http://youtube.com/user/tomayac) (screencasts)
5.3.3 Implementation of the dog food principle ............................................. 40

6 Conclusion and prospect .............................................................................. 44

6.1 Summary ...................................................................................................... 44
6.2 Conclusion ................................................................................................... 44
  6.2.1 Code generation ..................................................................................... 44
  6.2.2 Type estimation ..................................................................................... 44
  6.2.3 WADL .................................................................................................... 45
6.3 Prospect and future areas of research ...................................................... 45
  6.3.1 Does REST need WADL? ..................................................................... 45
  6.3.2 Improve the quality of the generated code ......................................... 46
  6.3.3 Improve type estimation ....................................................................... 46
  6.3.4 Improve the GUI ................................................................................... 46

Annex A ............................................................................................................. 47

  Time schedule .................................................................................................. 47

Bibliography ...................................................................................................... 49

  Foreword ......................................................................................................... 49
  Printed publications ......................................................................................... 49
  Electronic sources ........................................................................................... 49
Automatic Multi Language Program Library Generation for REST APIs

By Thomas Steiner (http://blog.tomayac.de, tomac@google.com)

1 Introduction and definitions

1.1 What Web Services and cell phones have in common

1.1.1 Introductory thoughts

Modern Web Services and cell phones share a lot of common properties: they have become a part of our daily lives, you can do awesome things with them, and of course they are supposed to make our communication patterns easier. And they have both kind of failed. How many times have you seen these people talking in the train by phone, telling someone else that they are actually sitting in there, and that their train will be 5 minutes late. Oh my God, five minutes late! Many of today's Web Services are pretty much like that. Just like the people that pick you up at the station do not assume that your car got stolen, then there was an earthquake, then a terrible flood, locusts...1, Web Services as well should just make sense of the situation themselves, without the need to telling them about every minute the train is off schedule.

1.1.2 From cell phones to Web Services

However, a lot of Web Services represent this kind of "five minute late" guys. They are verbose, talkative, and contain a lot of boilerplate. Most of the developers out there just do not care whether the ID of something is an Int32, or an Int64 value. This kind of strong typing in a Web Service intended for a broad audience to use makes it fragile and inflexible, where instead it should be forgiving and easy-going. The common SOAP/WSDL approach for Web Services forces exactly the way it should not be.

"But SOAP/WSDL has the advantage of automatic code generation", you might oppose. Again, this is like calendar synchronization between your phone and your favorite PIM software. In principle it works, at least it should, there exist some flaws you know how to work around, and others you just got used to live with. There are many SOAP toolkits2 in the wild, some with full SOAP 1.x support, others with partial support, and each of them - like every non-trivial software product - with its own bugs. An in theory correct WSDL file might just work perfectly with one toolkit, and need a lot of manual fine-tuning with another. Like calendar syncing... I think you got the point.

---

1 Freely adapted from the Blues Brothers movie, http://www.imdb.com/title/tt0080455/quotes

What Web Services and cell phones have in common


1.2 A short definition of SOAP/WSDL and REST

1.2.1 SOAP

SOAP is a protocol for exchanging XML-based messages over computer networks. Usually this happens via HTTP(S), even if SMTP is a valid application layer protocol as well. Originally SOAP was an abbreviation for Simple Object Access Protocol, however, with the introduction of version 1.2 of the protocol, the abbreviation was dropped as it was considered misleading. No one should be talking about SOAP without having read Pete Lacey's "The S stands for Simple" blog entry. This is probably one of the most entertaining and fun ways to learn a lot about SOAP.

1.2.1.1 A little SOAP history

Initially Don Box, Mohsen Al-Ghosein, Dave Winer, and Bob Atkinson designed the protocol back in 1998. At the time Atkinson and Al-Ghosein worked with Microsoft in Redmond, later IBM as well supported the team. Today the SOAP specification is maintained by the independent XML Protocol Working Group of the W3C (World Wide Web Consortium). The protocol development team chose XML for the broad acceptance of XML with companies, and for the wide spectrum of available tools around this language.

1.2.1.2 Advantages and disadvantages of SOAP

Because it is just modeled on top of HTTP, SOAP works well with firewalls. This is a major advantage of SOAP over other technologies like DCOM. The main weakness of SOAP is its lengthy XML format. However, XML allows for direct wire-level inspection of the data, as XML is human-understandable. This makes it quite slow in comparison to binary formats both for bandwidth usage, but also for parsing the messages. Especially for smaller messages there is a considerable overhead of the actual data, and the protocol boilerplate.

1.2.1.3 Structure of a SOAP message

SOAP messages consist of an Envelope container that holds a necessary Body element and an optional Header element. SOAP has its own namespace, usually referred to as soapenv. In the following a sample SOAP request for an imaginary credit card number validating service (adapted from the German Wikipedia entry on SOAP). This SOAP message contains just a SOAP Envelope and Body, together with three application-specific elements in the Body:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
   <soapenv:Body>
     <ns1:validate xmlns:ns1="urn:CardValidator"
       soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
     <number xsi:type="xsd:string">1234 5678 9876 5432</number>
     <valid xsi:type="xsd:string">12/08</valid>
   </soapenv:Body>
</soapenv:Envelope>
```
1.2.2 WSDL

You cannot talk about SOAP without talking about WSDL, the Web Services Description Language. It serves to specify the data types, parameter lists, operation names, transport bindings, and the endpoint URI involved in a Web Service. WSDL is an XML-based service description language. The version 2.0 of the specification was released in June 2007, while the currently most used version 1.1 was released in 2001.

1.2.2.1 What is WSDL good for?

If a client wants to use a Web Service, it can access the service's WSDL in order to discover the available methods and the used data types. WSDL uses the XML schema types, and allows for user-extended types to be embedded directly into the WSDL file. Each method is accessible by means of a port. Ports are used to define individual endpoints by specifying single addresses for a binding. Bindings define the concrete protocol (usually HTTP) to be used for the messages involved in a SOAP communication. Messages and ports are defined in separate ways. This approach allows the reuse of existing abstract definitions with concrete instances.

1.2.2.2 Criticism

The WSDL specification is considered very complex, hard to read, and difficult to understand. In addition to that it has to be noted that common practice of using the language, and the way usage is defined by the specification, do not completely comply. This is also due to the different parameter encoding standards: RPC/Encoded, Document/Literal, and the Wrapped-Document/Literal style introduced by Microsoft. All in all this makes SOAP/WSDL less interoperable and less intuitive to use as it should be, for all end-users, Web Service providers, but also for SOAP toolkit developers. Usually WSDL files are auto-generated. This order is another main source of criticism, as it normally does not allow for contract-first development, but rather forces the developer to do the straight opposite.

1.2.3 REST

The term REST stands for Representational State Transfer. It was coined in the year 2000 by Roy Thomas Fielding in his doctoral dissertation about the principle design of the modern web architecture. The most famous quote\(^4\) is

"Representational State Transfer is intended to evoke an image of how a well-designed web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

1.2.3.1 REST as non-SOAP

In addition to this quite abstract definition the term is also used in a loose sense to describe any simple interface that transmits domain-specific data over HTTP, where it is to be noted that this transfer happens without any additional messaging layer such as SOAP and usually without session tracking. The main idea of REST is to apply verbs like GET, POST, PUT, and DELETE to nouns (i.e. subjects) that represent resources. These verbs happen to be HTTP methods, but are not limited to those operations. The original concept assumes communication to be stateless. Each HTTP message should contain all the necessary information to understand
the message. However, in practice there is often added a state by means of cookies or session variables.

Due to the fact of every resource having its very own URI, resource usage statistics often reduces to analyzing server log files. In addition to that efficient caching of requests is possible. Representations of REST systems are usually built on top of XML or (X)HTML. These languages can contain resource descriptions and hyperlinks. It is thus possible to navigate from one resource to another without an additional infrastructure layer. Each resource represents a state. Upon resource change the client transfers to a new resource representation, and is thus in a new state. Hence the name Representational State Transfer.
2 Definition of the problem space

2.1 Introducing the REST challenge

2.1.1 Introductory example

Besides all the negative points of SOAP that have been outlined before (verbose, complex, inconsistency in usage), there are very strong positive points as well. OK, SOAP/WSDL might not be the best choice for every application, but for many at least it is a not so bad one. And even if code generation does not always work perfectly, it usually saves a lot of work. There is just this feeling that there should be something simpler, more straightforward, and more intuitive. And then REST enters the stage. It is not that REST by definition is easier than SOAP/WSDL. In contrary for machines it is not easier at all. To understand this, let us have a look at Yahoo's REST API for its News Search\textsuperscript{5}. This API allows the user to search the Internet for news stories. You place requests by means of query parameters to the API's endpoint:

\[
\text{http://search.yahooapis.com/NewsSearchService/V1/newsSearch?}\ \text{appid=YahooDemo}\ &\ \text{query=madonna}\ &\ \text{results=2}\ &\ \text{language=en}
\]

In this example just by looking at the URL we can easily see that there are four parameters:

- one for the \textit{appid},
- one for the actual \textit{query},
- one for the number of \textit{results},
- and finally one for the result \textit{language}.

Further, the service endpoint is \texttt{NewsSearchService/V1/newsSearch}, consisting of:

- the service name (\texttt{NewsSearchService}),
- an API version number (\texttt{V1}),
- and the actual API request (\texttt{newsSearch}).

2.1.2 Hackable URLs

We as human beings have the ability to actually understand such an API just by having a closer look at it. The authors very much believe in the principle of "hackable" URLs\textsuperscript{6}. A hackable URL is a URL that by its very structure allows you to retrieve the desired content. We want to get back to the news search example. If we, say, want 20 results instead of just 2, and we would rather prefer any language than just restricting the results to English, we can simply try "hacking" the URL, assuming that setting the \texttt{results} variable to 20 will guarantee the first property, and completely omitting the \texttt{language} variable the second, and end up with a new request URL:

\[
\text{http://search.yahooapis.com/NewsSearchService/V1/newsSearch?}\ \text{appid=YahooDemo}\ &\ \text{query=madonna}\ &\ \text{results=20}
\]
2.1.3 Need for a simple description language

Why cannot a machine do this? Well, because machines are pretty poor hackers. There is need for a description language for this kind of RESTful APIs. If we think of a SOAP/WSDL approach for the news search Web Service, we would end up with a long WSDL file including the description of the types, the requests, the responses, the bindings, the port types, and the service endpoint. But do we really need all this? What are we actually interested in?

- What parameters does the specific Web Service method offer?
  - Are there any optional, fixed, or required parameters?
  - Do parameters have a fallback default value if they are omitted?
  - What type does each parameter have?

- What is the specific Web Service method's endpoint?

- Is the HTTP method GET, PUT, DELETE, or PUT?

- What errors can occur?

At present, such services are described using a mixture of XML schema and textual description. We want a simpler description language than WSDL, but we want to keep all of its benefits, especially the ability to automatically generate code. Sun Microsystems’s Norman Walsh writes: "We know the hard things are possible, we just have to make the easy things easy."

2.2 Different variants of REST Web Services

2.2.1 Classification of REST Web Services

In the following we want to propose a classification for the different variants of REST Web Services that can be found in practice. Please note: this classification uses the "REST as non-SOAP" definition of REST. In addition to that we will differentiate between static and dynamic resources: static resources will always be the same for every user of the API, whereas dynamic resources will depend on the user. For example the results of a Web search API would be independent of the user, whereas a banking API would necessarily return different results for every user (read "be personalized").

- Static resources:
  (The resource is the same for each user)

  1. Static "true REST" in Roy T. Fielding's sense:
     This type of API is characterized by a unique address for each resource. Typically each endpoint (printed in bold) corresponds to an operation name. All HTTP methods are allowed, however, in practice most APIs use GET and POST. Prominent examples are Yahoo's Search APIs.

     \{GET | POST | PUT | DELETE\}
     http://example.com/path/to/operation1?
     param1=value1&
     param2=value2
Different variants of REST Web Services

2. **Static "parameterized operation"**: This type of API is characterized by a unique address for all resources. Each resource is addressed by a special **operation parameter** (printed in bold) that holds the operation name. All HTTP methods are allowed, however, in practice most APIs realize all function calls via GET. A prominent example is Amazon's E-Commerce Service API.

   ```
   {GET | POST | PUT | DELETE}
   http://example.com/path/to/operation2?
   param1=value1&
   param2=value2
   ```

3. **Static "XML over HTTP"**: This type of API is characterized by a unique address for all resources. There is no operation parameter as this type of API basically just uses **HTTP to tunnel remote procedure calls**. The operation name is thus contained in the XML payload. Most of these APIs use POST, even if all methods besides GET (XML cannot be transferred as a query parameter) are allowed in theory. A prominent example is Google's Checkout API.

   ```
   {GET | POST | PUT | DELETE}
   http://example.com/some/path?
   operation=operation1?
   param1=value1&
   param2=value2
   ```

   ```
   {GET | POST | PUT | DELETE}
   http://example.com/some/path?
   operation=operation2?
   param1=value1&
   param2=value2
   ```

Dynamic resources:
(The resource is different for every user)

1. **Dynamic "true REST" in Roy T. Fielding's sense**: This type of API is characterized by a unique address for each resource for each user. Typically each **endpoint** (printed in underlined bold) corresponds to an operation name. All HTTP methods are allowed, however, in practice most APIs use GET or POST. A prominent example is Wesabe.com's banking API.

   ```
   {GET | POST | PUT | DELETE}
   http://example.com/path_param1/\{path_value1\}/operation?
   ```
query_param1=query_value1
query_param2=query_value2

[The other cases 2 and 3 (dynamic "parameterized operation" and dynamic "XML over HTTP") work in an analog way]

2.2.2 Mapping of the different Web Service variants to programming code

As we will show below, the cases 1 and 2 can be directly mapped to source code classes. However, due to the request data being given in form of an XML payload, case 3 causes problems, as the hierarchical structure of XML cannot easily be mapped to the linear structure of a class constructor. The other cases are relatively straightforward, we simply map each path parameter to a class attribute, and afterwards do the same for each query parameter. See the following constructors as generic examples:

1. The "true REST" request to be mapped:

{GET | POST | PUT | DELETE}
http://example.com/
path_param1/{path_value1}/.../path_paramN/{path_valueN}/
operation1?
  query_param1=query_value1&
  ...
  query_paramM=query_valueM

The corresponding class constructor:

class Operation1(
  path_param1,
  ..., path_paramN,
  query_param1,
  ..., query_paramM
)

2. The "parameterized operation" request to be mapped:

{GET | POST | PUT | DELETE}
http://example.com/
path_param1/{path_value1}/.../path_paramN/{path_valueN}/
operations?
  operation=operation1&
  query_param1=query_value1&
  ...
  query_paramM=query_valueM

The corresponding class constructor:

class Operation1(
  path_param1,
  ..., path_paramN,
  query_param1,
  ..., query_paramM
)
2.2.3 Drawbacks with "XML over HTTP"

As can be seen from the generic examples the generated code for the two first cases ("true REST" and "parameterized operation") is the same. It is thus possible to completely abstract the generated classes from the underlying APIs, which is a great advantage from a consistency point of view. However, case 3 ("XML over HTTP") being not easily mappable to a linear class constructor structure, there is further research necessary on how to solve this kind of problems. JAXB in Java is a solution that might be adaptable for other languages as well. See the prospects in chapter 6 for more details. In addition to that potential XML namespaces and XML schemas in the XML payload need to be supported. Thus code generation gets more complicated and less straightforward. We definitely will address these challenges in future releases.
3  REST Web Services description languages

3.1  Proposals on description languages for REST APIs

The question of describing (REST) Web Services in a machine-readable way other than WSDL has been raised before. However, often the motivation behind was more to get rid of WSDL rather than actually solving the REST description issues. Many suggestions are more or less ad hoc inventions designed to solve particular problems. It is to be noted that with WSDL 2.0 it is possible to describe REST services, but here we want to focus on some examples of non-WSDL approaches.

3.2  Listing of alternative Web Service description languages

In the following listing of Web Service description languages different from SOAP, we have included samples which are either provided by the authors themselves, or which have been generated by us to the best of our knowledge according to the provided specifications. In order to simplify the sample structure, those parts of the sample describing a request have been highlighted in underlined green, and those parts describing a response have been highlighted in italic red. In some cases the samples have been shortened or reformatted for better readability. However, these changes do not tamper the samples' expressiveness. Due to the ravages of time some of the described services do not exist anymore, or have been changed (see the release date for some of the proposals).

3.2.1  WRDL (Web Resource Description Language)

Description: Paul Prescod, co-author of the debatable "XML Handbook", offers an approach that treats the available REST API functions as a collection of resources. Each resource is represented by its very own URI and represents a state in the process of handling the requests. Each method has one or more valid representations. When XML is sent or received, the WRLD engine should check - just like an XML schema validator - whether the data corresponds to the valid representations or not. While XML schema does not describe the interaction between different resources, the author claims that WRDL is capable of doing so by describing the service's runtime behavior. Therefore, the WRDL engine should enrich each hyperlink with a resource type. Hyperlinks may come back in response to requests, based on the specification found in the schema. Thus each action has a set of defined follow-up actions. POST and freely definable other HTTP methods can have an optional @creates attribute that allows checking the result against a schema for validation. The author provides a fundamental (piecewise in pseudo-code) Python implementation, however states that the code is in a very rough and premature state (which is true).

Author-provided sample: Babelfish Translation service

```xml
<?xml version="1.0"?>
<!DOCTYPE types SYSTEM "wrdl.dtd">
<types>
  <resourceType representations="html" name="babelfish">
    <POST>
      <input representations="babelFormData">
        <query name="doit" apiName="translate" default="done"/>
        <query name="tt" default="urltext" apiName="actiontype"/>
        <query name="urltext" use="required" apiName="urltext"/>
        <query name="lp" use="required" apiName="languages"/>
```
Automatic Multi Language Program Library Generation for REST APIs

Listing of alternative Web Service description languages

3.2.2 NSDL (Norm's Service Description Language)\textsuperscript{12}

Description: During the development of NSDL, Sun Microsystems's employee Norman Walsh was driven by two main goals: first, make Web Services just as transparently usable as normal local code libraries, and second, allow service providers to describe their services in a totally interoperable way. Walsh limits his approach to describing services accessible by means of HTTP POST or GET. His proposal is pretty straightforward: each method is abstracted in a service element containing attributes that describe the HTTP method and the service URI. Each service element has one request, and one response element. Requests contain parameter elements with \texttt{@type}, \texttt{@optional}, and \texttt{@default} attributes. For POST requests there is an additional \texttt{body} element that, in contrast to GET parameters that are simply URL-encoded, allows complete XML requests to be sent. Response elements contain result elements with a \texttt{@select} attribute that, assuming an XML response, via XPath allows the desired subset of the whole response XML to be selected. In addition to that, NSDL supports fault elements that also via XPath allow reacting on erroneous requests. The author provides a complete implementation of the language in form of three Perl modules, indeed allowing Web Services with an appropriate NSDL description to be used in a high-level way from within any Perl script.

Author-provided sample: Amazon Item Search service

\begin{verbatim}
<descriptions xmlns="http://nwalsh.com/xmlns/nsdl#"/>
<service xmlns:a="http:// [...]
   name="booksbykeyword"
   action="get"
   uri="http:// [...]
         SearchIndex=Books&"
>
  <request>
    <parameter name="SubscriptionId" type="xsd:string"/>
    <parameter name="Keywords" type="xsd:string"/>
  </request>

  <response>
    <result name="count" select="/a: [...] /a:Items/a:TotalResults" />
    <result name="time" select="/a: [...] /a:OperationRequest/a:Request [...]"
    <result name="titles" select="/a: [...] /a:Items/a:Item [...] /a:Title"
  </response>
</service>
<descriptions>
\end{verbatim}
3.2.3 SMEX-D (Simple Message Exchange Descriptor)

Description: Tim Bray, director of Web technologies at Sun and co-inventor of XML has proposed SMEX-D, a description language developed with the goal in mind to provide an implementation in the simplest way that could possibly work. SMEX-D focuses both on SOAP- and REST-style message exchanges. The basic idea is that there is necessarily a request, followed by an optional response element. Requests can take on three forms: name-value pairs, SOAP, and XML different from SOAP. Responses can either be SOAP, XML different from SOAP, or nothing at all. Name-value pairs describe URI parameter requests that can be typed by means of types defined in XML schema section 3.2, where the default type is simply xsd:string. SOAP messages have a header and a body element each of which must contain at least one language element. The language element describes the elements that may appear in the SOAP header and body. This is realized either via putting both the SOAP header and body independently in a certain namespace, and/or via assigning an XML schema to the header and body. This also applies to non-SOAP XML (NSX) messages, with the difference that there are no header and body elements.

Author-provided sample: Amazon Item Search service

```xml
<smex-d xmlns="http://smex-d.net/ns/"
       href="http://webservices.amazon.com/onca/xml">

  <request form="pairs">
    <pair name="Service" />
    <pair name="Subscription" />
    <pair name="Operation">
      <enum>
        <v>ItemSearch</v>
      </enum>
    </pair>
    <pair name="AssociateTag" />
    <pair name="ResponseGroup">
      <enum>
        <v>Groups</v>
        <v>Accessories</v>
        ...
        <v>VariationSummary</v>
      </enum>
    </pair>
    <pair name="Style" type="anyURI" />
    <pair name="ContentType" />
    ...
    <pair name="MaximumPrice" type="decimal" />
    <pair name="MerchantId" />
    ...
  </request>

  <response form="nsx">
    <language namespace="http://...amazon.com/...">
      <schema flavor="xsd"
             href="http://...amazon.com/...">
      ...
    </language>
  </response>
</smex-d>
```

Spec last updated: May 2005
3.2.4 Resedel (REstful SErvices DEscription Language)\textsuperscript{14}

**Description:** John Cowan has a blog called Recycled Knowledge and this was exactly the motivation that drove him when he created Resedel. It is a mixture of NSDL and SMEX-D. Cowan states having stolen most of Norman Walsh’s RPC-style encoding, and the rest from Tim Bray. As of today there is only a Relax NG schema available, without any description at all, besides the comments. The approach maps the four basic functions of persistent storage CRUD (CREATE, READ, UPDATE, DELETE) directly to the HTTP methods POST, GET, PUT, DELETE. This adds another abstraction layer, however, given the different assumptions of the CRUD model and REST concerning UPDATE, not without being controversial.

**Sample (created in accordance to the specification):** Yahoo News Search service

```xml
<?xml version="1.0" encoding="UTF-8"?>
<resedel version="0.2" xmlns="http://www.ccil.org/~cowan/resedel/ns">
  <type id="Yahoo Search" flavor="xsd"
  <service id="News Search"
           uri="http://search. [...] .com/ [...] newsSearch
                 operation="read">
    <request soap="false">
      <parameter name="appid" />
      <parameter name="query " />
      <parameter name="type" default="all" />
      <parameter name="results" typeref="xsd:integer" default="10" />
      <parameter name="start" typeref="xsd:integer" default="1" />
      <parameter name="sort" default="rank" />
      <parameter name="language" />
      <parameter name="site" default="" />
      <parameter name="output" default="xml" />
      <parameter name="callback" />
    </request>
    <response>
      <language uri="urn:yahoo:yn">
        <schema flavor="xsd" root="ResultSet"
                href="NewsSearchResponse.xsd"/>
      </language>
      <fault name="Bad Req" status="400" />
      <fault name="Forbidden" status="403" />
      <fault name="Service Unavailable" status="503" />
    </response>
  </service>
</resedel>
```

**Spec last updated:** May 2005

3.2.5 RSWS (Really Simple Web Service Descriptions)\textsuperscript{15}

**Description:** Richard Salz, named contributor to the HTTP/1.0 and HTTP/1.1 specs, proposes a three-part approach consisting of

1. a schema definition which defines how messages look like
2. an interface definition which describes what methods are provided
3. a location definition which tells the processor where to find the service
These three elements are put in a container called description that has a @name attribute that actually is a URI. According to the author, RSWD should support all available schema description languages. The interface describes the available methods in operation containers that hold input and output elements. Basically these two elements contain simply a reference to the appropriate section of the schema in order to define the valid parameters. For this initial version Salz abstains from defining error elements. According to Salz this is because of the necessity to keeping RSWD independent from the different SOAP versions.

Author-provided sample: Generic imaginary description

```xml
<rsws:description name="http://example.com/rsws/">
  <rsws:schema id="mytypes">
    <![CDATA[
      default namespace = "http://example.com"
      element foo { attribute bar { string } }
    ]]>}
  </rsws:schema>
  <rsws:interface id="sample">
    <rsws:operation>
      <rsws:input ref="tns:fooIn"/>
      <rsws:output ref="tns:fooOut"/>
    </rsws:operation>
    <rsws:fault soap11Faultcode="env:server"
      soap12Code="env:server"
      soap12Subcode="tns:ErrorValues"/>
    ...
  </rsws:operation>
  <rsws:location>
    <rsws:provides href="#mgmt"/>
  </rsws:location>
</rsws:description>
```

Spec last updated: October 2003

3.2.6 WDL (Web Description Language)¹⁶

Description: Dave Orchard is member of several W3C committees. In his approach the author suggests to basically just list all the resources of a Web Service where each resource is assigned to a @location via an attribute. Each resource can have operation elements. In addition to that, in order to simplify the syntax with the most common GET operation, WDL supports a getoperation element. These operation (and of course getoperation) elements in turn serve as containers for input, output and fault elements (however faults are not defined in the provided schema). He deliberately does not list all the parameters a service can potentially have, claiming that this would be a step backwards to the WSDL 1.1 definition weakness. Orchard wants the approach to also include HTTP headers, configuration, and status codes in order to stick to the existing significations of HTTP communication.

Author-provided sample: Yahoo Search service
Listing of alternative Web Service description languages

3.2.7 WADL (Web Application Description Language)

**Description:** Dr. Marc J. Hadley is a senior staff engineer in the Office of the CTO with Sun Microsystems, and represents Sun on the W3C XML Protocol and W3C WS-Addressing working groups. There he is co-editor of the SOAP 1.2 and WS-Addressing 1.0 specifications. Hadley's approach intents to list the resources in hierarchical form, where the resources are grouped together by a `resources` element that has a `@base` attribute indicating the Web Service's base address. In consequence each `resource` element has a `@path` attribute which defines the resource's path relative to the base address. Each `resource` has `method` elements that in turn are defined by `request` and `response` elements. WADL allows schemas to be included in the description, the author calls this grammar. Both XML schema and RelaxNG are supported. Requests and responses can be defined by means of a `representation` element allowing for schema-conform representation. Via an `@element` attribute a particular section of the schema can be selected for refinement. Responses can also have `fault` elements. These can have a `@status` attribute and in addition to that be represented by a schema. WADL has some interesting extensions not included in other approaches, the most interesting one probably being the so-called template parameters. This feature allows resources to have a dynamic path variable that only at runtime is substituted by the actual value.

**Author-provided sample:** Yahoo News Search service
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="[...] sun.com/wadl/ [...] wadl.xsd"
    xmlns:tns="urn:yahoo:yn"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:yn="urn:yahoo:yn"
    xmlns:ya="urn:yahoo:api"
    xmlns="http://research.sun.com/wadl/2006/10">
    <grammars>
        <include href="NewsSearchResponse.xsd"/>
        <include href="Error.xsd"/>
    </grammars>
    <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
        <resource path="newsSearch">
            <method name="GET" id="search">
                <request>
                    <param name="appid" type="xsd:string" style="query"
                        required="true"/>
                    <param name="query" type="xsd:string" style="query"
                        required="true"/>
                </request>
                <response>
                    <representation mediaType="application/xml"
                        element="yn:ResultSet"/>
                    <fault status="400" mediaType="application/xml"
                        element="ya:Error"/>
                </response>
            </method>
        </resource>
    </resources>
</application>

Spec last updated: November 2006

In order to gain a better understanding of the structure of WADL, in the following we try to mix the best out of various representation styles, namely regular expressions, XML schema, and actual XML. This representation served us as a quick reference during the implementation of REST Describe & Compile as it shows each element's child elements and attributes in an informative, however informal way. The document is thus shorter than it would be possible with an XML schema document:
WSDL pseudo XML/Regular Expression/Schema Document

```xml
<application>
  <doc xml:lang="" title="xsd:string" />
  <grammar>
    <doc xml:lang="" title="xsd:string" />
    <include href="xsd:anyURI" />
  </grammar>
  <resources base="xsd:anyURI"/>
    <doc xml:lang="" title="xsd:string" />
    <resource id="xsd:ID" path="xsd:string" type="[xsd:anyURI]" queryType="application/x-www-form-urlencoded" />
  </resources>
  <method id="xsd:ID" name="{GET|POST|PUT|DELETE|HEAD}">
    <doc xml:lang="" title="xsd:string" />
    <request>
      <doc xml:lang="" title="xsd:string" />
      <representation href="xsd:anyURI">
        // the referenced representation must have an attribute @id
        // equal to the @href value
        <representation mediaType="" element="" profile="" status="" />
      </representation>
      <param path="" />
    </request>
    <response>
      <doc xml:lang="" title="xsd:string" />
      <representation href="xsd:anyURI">
        // the referenced representation must have an attribute @id
        // equal to the @href value
        <representation mediaType="" element="" profile="" status="" />
      </representation>
      <fault mediaType="" element="" profile="" status="" />
    </response>
  </method>
</application>
```
3.3 Observations during the examination of the proposals

When we have a closer look at the above proposals there can be remarked a peculiar phenomenon: since 2005 there is certain inactivity on the field of description languages. public-web-http-desc@w3.org is the World Wide Web Consortium's official mailing list dedicated to discussion of web description languages based on URI/IRI and HTTP, and aligned with the Web and REST Architecture. The activity for the year 2006 with 72 single postings has more than halved regarding the 147 postings in 2005. Besides WADL, all proposals date back to the year 2005 or earlier. The term "REST" was brought up by Roy T. Fielding in 2000, however, only since 2005 the concept of RESTful Web Services has really started to explode. What made developers and web technology architects suddenly turn away from the idea of describing REST Web Services in a general way? Why did most of the initial drafts never get updated, picked up by others, and, not to mention, get implemented?

At present there is no satisfactory answer. Maybe it is just because REST is so simple in many cases. Most of the APIs in the wild that call itself RESTful actually are not really RESTful in Fielding's sense of the term (especially regarding application state and functionality that need to be divided into uniquely addressable resources), but rather "RESTful" in the sense of not being SOAP. Some Web Services use HTTP to tunnel function calls. This technique is called XML over HTTP. Web services that follow this architecture typically just have one endpoint and can be used by POSTing raw XML data. A prominent example is Google's Checkout API. If an API is limited to HTTP GET operations, not even one single line of code is necessary in order to use the API: these pure read-only APIs can be explored directly from within any web browser, here common examples are eBay's REST API, Yahoo's Search REST APIs introduced earlier, and equally Amazon's REST Web Services.

3.4 Choice of a description language

Given the above listing the repertory of available REST Web Service description languages is rather limited. Most propositions never got implemented at all, and probably never will. Besides WADL, all description languages are described in a rather informal way in form of a blog entry, or in form of an article illustrated by generic examples. Nearly all of the specifications have been written once, without having been updated later. In the following we provide a quick table overview of the various description languages published so far:
## Description Language

<table>
<thead>
<tr>
<th>Description Language</th>
<th>Mod. Date</th>
<th>Code avail.</th>
<th>Schema avail.</th>
<th>Pub. form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WRDL</strong> <em>(Web Resource Description Language)</em>: <a href="http://www.prescod.net/rest/wrdl/wrdl.html">http://www.prescod.net/rest/wrdl/wrdl.html</a>*</td>
<td>Nov 2002</td>
<td>Yes (Python, pseudo code)</td>
<td>None</td>
<td>Article</td>
</tr>
<tr>
<td><strong>NSDL</strong> <em>(Norm's Service Description Language)</em>: <a href="http://norman.walsh.name/2005/03/12/nsdl">http://norman.walsh.name/2005/03/12/nsdl</a>*</td>
<td>Sep 2005</td>
<td>Yes (Perl)</td>
<td>RelaxNG</td>
<td>Blog entry</td>
</tr>
<tr>
<td><strong>SMEX</strong> <em>(Simple Message Exchange Descriptor)</em>: <a href="http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX-D">http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX</a>*</td>
<td>May 2005</td>
<td>No</td>
<td>None</td>
<td>Blog entry</td>
</tr>
<tr>
<td><strong>Resedel</strong> <em>(RESTful SERVICES DESCRIPTION LANGUAGE)</em>: <a href="http://recycledknowledge.blogspot.com/2005/05/resedel.html">http://recycledknowledge.blogspot.com/2005/05/resedel.html</a>*</td>
<td>May 2005</td>
<td>No</td>
<td>RelaxNG</td>
<td>RelaxNG</td>
</tr>
<tr>
<td><strong>WDL</strong> <em>(Web Description Language)</em>: <a href="http://www.pacificspirit.com/Authoring/WDL">http://www.pacificspirit.com/Authoring/WDL</a>*</td>
<td>May 2005</td>
<td>No</td>
<td>None</td>
<td>Article</td>
</tr>
<tr>
<td><strong>WADL</strong> <em>(Web Application Description Language)</em>: <a href="https://wadl.dev.java.net">https://wadl.dev.java.net</a>*</td>
<td>Nov 2006</td>
<td>Yes (Java)</td>
<td>XML schema</td>
<td>Semi-formal specification, XLM and RelaxNG schemas</td>
</tr>
</tbody>
</table>

Already from this table, but also considering the information given in the sections above, it is pretty obvious that almost all ways lead to WADL. Besides the question whether WADL as a description language is needed at all, or not, at present there is not big of a choice. So, after having checked all the available alternatives, we decided to go for WADL.
4 Planning and implementation

4.1 Design document for the implementation of a WADL editor and a WADL compiler

Authors: Implementation: Thomas Steiner <tomac@google.com>
Mentoring: Patrick Chanezon <chanezon@google.com>

Status: Working beta 0.7.2 (as of August 2007).
See http://tomayac.de/rest-describe/latest/RestDescribe.html for details.

Objective: The project's main goal is to create a compiler that allows for automatic client code generation for REST APIs in various programming languages. The project's second goal is to implement a rich web application that allows for more or less interactive Web Service description creation. The description language to be used is WADL, the Web Application Description Language. The application is to be implemented using the Google Web Toolkit.

Background: In order to understand what this project is about, a good understanding of RESTful design and WADL is necessary. As to our knowledge, no other project has been started yet with the ambitious goal of providing auto-generated code for REST APIs in various programming languages. The author of WADL, Marc Hadley, provides a code-generating tool for Java called WADL2Java (in analogy to WSDL2Java).

Overview: REST APIs are in broad use throughout the Web Services development world. However with RESTful Web Services, in contrast to SOAP/WSDL, capabilities for automatic code generation are still very limited, or not available at all. The project consists of two sub-projects:

- The project's main goal is to create a WADL Compiler that allows for automatic client code generation for REST APIs in various programming languages. This should be based on a meta description of a RESTful Web Service. After a detailed market survey of the available description languages, the meta description language of choice is Marc Hadley's approach named WADL. We call this part of the application "REST Compile". The following diagram tries to illustrate the idea:
• In our opinion RESTful Web Services are so popular because of their simplicity for human beings. However, writing a (WADL) description for already existing Web Services can be a rather annoying manual job. The project's second goal is thus to implement a rich web application in form of a **WADL Editor** which allows for more or less interactive WADL creation based on "copied and pasted" sample requests which is usually available in textual form in the human-readable docs of the Web Service. This part of the application is referred to as "**REST Describe**". Below the correspondent diagram:

![Flowchart](image)

**4.2 Implementation details**

**4.2.1 Program version**

At the time of writing the application had reached version 0.7.2 beta. The following description covers only this release. It is very likely that future releases will invalidate parts of the current description.

**4.2.2 Development environment: Eclipse\(^{21}\) and Google Web Toolkit\(^{22}\)**

**Eclipse**: Eclipse is one of the most powerful IDEs for Java (and other languages). Google Web Toolkit has a command line tool for creating an Eclipse project that can then be imported into Eclipse. Eclipse is the default development environment for Google Web Toolkit.

**Google Web Toolkit**: The Google Web Toolkit (GWT) is an open source product that allows developers to create Ajax applications in the Java programming language (at the time of writing Java 1.4). It integrates easily in IDEs like Eclipse. The toolkit ships with a so-called **hosted browser** that can be used to run created applications smoothly from within the IDE. The main feature, however, is a **Java to JavaScript compiler** that transforms programs written in Java into web applications written in JavaScript, that will run in any web browser. One of the design goals of Google Web Toolkit is to be able to profit from the excellent tools available in the Java world (JUnit, IDEs, ...) also for Web application development.
4.2.2.1 Advantages and features of the GWT

Included in the toolkit are common widgets like trees, radio buttons, panels, and the like. In addition to that the toolkit implements event listeners like WindowResizeListener that are an abstract layer on top of the browser's event handling. Common issues with Web development are browser differences and incompatibilities. These make life a lot harder than it should be. In consequence the toolkit abstracts all these worries away, and the generated applications usually behave the same on all the different browser platforms. The toolkit deals with browser incompatibilities by creating separate files for each browser family (like Internet Explorer, Safari, Firefox, ...). The advantage of this approach is that the footprint for each individual browser is kept small, as users only need to download the particular code for their browser, and not workaround code for different browsers.

4.2.2.2 Java and JavaScript

The 1.3.3 release of Google Web Toolkit is based on Java 1.4. Basically all Java code that uses the most common package classes java.lang and java.util can be compiled into idempotent JavaScript code, with some important limitations. To make this idea clear, let us have a look at an example: via XMLHttpRequests JavaScript applications can access resources as long as they do not hurt the Same Origin Policy. However, even if any arbitrary Java program may access external resources from third parties without any difficulty, this will not work in JavaScript (even if the code would compile without any error).

A closer look at the Same Origin Policy: if you are on the domain

   http://my.company.com/dir1/a.html,

another valid resource would be

   http://my.company.com/dir2/b.html,

however, an invalid external resource would already be

   http://your.company.com/dir1/a.html

because of the different host (your instead of my),

   https://my.company.com/dir1/a.html

would be invalid because of the different protocol (https instead of http), and finally

   http://different.enterprise.com/dir1/a.html

would be invalid as well for obvious reasons. In short, for a call to succeed, the domain name, the port, and the protocol must be the same.

Object orientation (OO) in JavaScript is prototype-based, whereas Java has a class-based OO model. To many programmers the class-based approach is more intuitive, so with the Google Web Toolkit you can program class-based JavaScript from within Java. In addition to that, Java's static type checking catches many faults at compile-time, that from a pure JavaScript approach would not be noted until run-time, if at all.

4.2.2.3 Justification of the selected tools

We have some experience with pure JavaScript application development, however, for this project the host company requirement was to use the in-house Google Web Toolkit. The widgets mentioned above are of great value for the project, especially the tree widget for the tree view of the generated WADL file (explanation later). The fact that the Google Web Toolkit generated code is compatible to most of today's browsers is a big plus for each JavaScript developer. While we probably would have been able to deal with some of the incompatibilities, having a toolkit do this for us is of great convenience, and allows us to focus on the implementation details of our application rather than on browser quirks. So even if the choice of Google Web Toolkit was not ours, we never felt bad about it. Google provides very good introductions and among many others a "hello world" application that helps understanding the general concepts of the toolkit.
4.3 Application architecture

4.3.1 External architecture

- From a user's point of view, the application is structured in two modules: **REST Describe** and **REST Compile**. In the following an overview of each module's responsibilities, followed by a screenshot of the particular module:

4.3.1.1 REST Describe module

**Provide WADL creation** of a Web Service based on sample requests. The idea is to profit from the information intrinsic to REST-style Web Services, i.e.

- The service structure corresponds to the URI structure
- Often the endpoint names correspond to the available Web Service operations
- Sample requests contain a lot of data that can be analyzed with an heuristic approach
- The HTTP method which is of particular interest for REST is part of the sample requests
- The response of a Web Service can be analyzed in order to gain more knowledge about the service, especially for fault handling and XML schemas in use

Support **WADL upload and modification** of existing WADLs. The more people will use WADL, the more will exist the need to modify and parse existing WADLs. REST Describe has a WADL parsing feature which checks the given WADL for correctness, and upon success creates a tree representation for interactive editing.

Support **WADL download** of the created WADL files. This allows API providers to offer a WADL description of their APIs to their clients in order to make API development more comfortable and faster.
**4.3.1.2 REST Compile module**

Support code generation in various programming languages based on a WADL file. At the time of writing these were

- PHP 5 (since version 0.2)
- Ruby (since version 0.2.1)
- Python (since version 0.2.2)
- Java (since version 0.3)

The first three languages are dynamically typed and interpreted languages, whereas Java is statically typed. In some respects the strict typing that Java enforces is only an outside-world feature, as internally in the end everything is a string that has to be transmitted via the HTTP protocol.
4.3.2 Internal architecture

In the following listing we present a short package-oriented overview of each class's role. This is necessary in order to understand the UML sequence diagrams that will be presented afterwards. There are five packages that bundle related classes like GUI functionality classes, or classes that are responsible for code generation. The left column contains the package name, the middle column contains the package's class names, and the right column gives a short explanation of what each class is good for.

```
com.google.code.apis.rest.client

RestDescribe.java

The entry point to the application. This is a singleton whose only job is to initialize the application GUI.

GUI

GuiFactory.java

This package contains all the GUI elements. The main screen consists of several horizontal and vertical panels that are docked to a dock panel inside of the GUI factory.

ParameterPanel.java
WadlPanel.java
MainMenuPanel.java
RequestUriPanel.java
RestCompilePanel.java
```

(Screenshot of the REST Compile module)
AboutDialog.java
HelpDialog.java
BatchUriDialog.java
CustomTypesDialog.java
DiscoveredItemsDialog.java
FullscreenDialog.java
SettingsDialog.java
TestRequestDialog.java
WadlPreviewDialog.java
WadlSaveDialog.java
WadlUploadDialog.java
ParameterTree.java
RequestUriTree.java
Indicator.java
HttpRequest.java
HttpResponseHandler.java
Notifications.java
Properties
Notifications_ca.properties
Notifications_de.properties
Strings.java
Strings.properties
Strings_ca.properties
Strings_de.properties

WadlTreeRoot.java
ApplicationItem.java
FaultItem.java
GenericClosingItem.java
GrammarsItem.java
IncludeItem.java
MethodItem.java
ParamItem.java
RepresentationItem.java
RequestItem.java
ResourceItem.java
ResourceTypeItem.java
ResourcesItem.java
ResponseItem.java

This package also contains all the dialog windows.

Of great importance are the two tree files which manage the parameter tree on the left side of the application (ParameterTree.java) and the request URI tree in the upper part of the main application window (RequestUriTree.java).

Indicator is a small notification class as seen on Gmail or other ajaxified sites. It signalizes script activity. HttpRequest.java and HttpResponseHandler.java represent typical script activities via Ajax.

In addition to that there are some internationalization (I18N) files for (currently) English, German, and Catalan language support. This is realized thanks to Google Web Toolkit's I18N module which is based on Java properties files.

In the Tree package there are all the classes that serve as WADL tree item representations in the center of the application window. The tree root is the WadlTreeRoot.java file, all other tree nodes may be added according to the WADL specification. The idea is to have a tree that behaves like an XML file, including code folding and interactive editing. The ParamItem.java and the MethodItem.java files are linked to the parameter tree and the request URI tree.
<table>
<thead>
<tr>
<th>Util</th>
<th>InvalidUriException.java</th>
<th>SyntaxHighlighter.java</th>
<th>Tools.java</th>
<th>TypeEstimator.java</th>
<th>Uri.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>WADL</td>
<td>Analyzer.java</td>
<td>WadlParser.java</td>
<td>WadlXml.java</td>
<td>ApplicationNode.java</td>
<td>DocNode.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FaultNode.java</td>
<td>FaultRepSuperNode.java</td>
<td>GrammarNode.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MethodNode.java</td>
<td>NamespaceAttribute.java</td>
<td>ParamNode.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RequestNode.java</td>
<td>RepresentationNode.java</td>
<td>RequestNode.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ResourceManager</td>
<td>ResourceTypeNode.java</td>
<td>ResourcesNode.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ResponseNode.java</td>
<td></td>
</tr>
</tbody>
</table>

This is a container package for utilities like URI management, syntax highlighting, debugging, and type estimation.

The WADL package contains everything that is related to WADL.

The analyzer checks sample request URIs, the parser parses uploaded WADLs, and the WadlXml converts a WADL tree into an XML file.

A WADL tree consists of many different nodes, each of them represented by its own class. Some nodes share subnodes for several analysis runs, e.g. Application nodes remember their subnodes, as each request URI in a batch analysis a priori belongs to the same application node.

The architecture of the code generators follows a classical factory pattern: a code generator factory initializes the particular generator classes.

All generators share a common HTTP error class that allows for fault handling.

The parameter class is an abstraction of a Web Service parameter and provides visibility information and coding conventions.

The generators communicate with the factory by means of a request messager.

For the static parts code generation happens via Java properties files. The
4.3.3 UML sequence diagrams

4.3.3.1 Class interaction

In the following we want to give an overview of how the classes interact. There are several steps during execution that can be combined to complete program cycles.

4.3.3.2 GUI setup

This is the first and basic step during program execution. The sole responsibility of this unit is to initialize the working environment. Strings and Notifications are internationalization classes that contain all the program messages in various languages. At run-time the correct class is initialized depending on the selected language. See the sequence diagram below for details:
4.3.3.3 Request URI analysis

One of the most common tasks is request URI analysis. The idea is to create a WADL XML tree based on a single request URI. Each request URI consists of an HTTP method and the actual URI string. The RequestUriTree class manages this information. This class first tries to analyze the request response in order to discover the namespaces and grammars that are used by the Web Service, and then initializes the Analyzer. The following sub-steps are all dependant on the request structure. In consequence the following diagram describes a standard case for a request URI with one parameter, and one level of resource hierarchy. First, the ResourcesNode gets created. It contains the URI authority as the ResourcesNode's base attribute. Second, a ResourceNode with its path attribute in correspondence to the URI path is initiated. Afterwards, a MethodNode reflects the HTTP method used for the request in the name attribute. The following RequestNode is just a container for the ParamNodes. These ParamNodes from an analysis point of view are the most interesting ones as this is where type analysis happens. Via a multi-step structure of regular expressions the correct type is tried to be estimated. Type estimation will be explained in a later section. The final DocNode preserves the original sample value for the parameter and contains information about the quality of the estimation (one out of "sure", "supposed", unsure).
4.3.3.4 Batch URI analysis

Batch URI analysis is basically just a modification of the previous case with the difference that there are more than one RequestUriTree instances at the beginning. These instances are created from within the GUI package's BatchUriDialog class. The processing of these URIs is in line with the previous diagram.
4.3.3.5 XML tree creation

Once a request has been parsed or an already existing WADL file has been uploaded, this step is responsible for creating a graphical representation of the tree. The XmlTree class instantiates an ApplicationItem instance, and from that point on the tree gets created item by item, where each item can have subitems according to the WADL schema. The diagram below shows a very incomplete view of what can happen. Tree creation is not a strictly sequential process which makes it hard to represent the correct workflows in a sequence diagram style. References may be circular and elements may have the same kind of child nodes. There is no generic way to represent this in UML form, so the diagram is something in between a concrete sequence and an attempt to show the process in its generality. However, the expressiveness is of rather limited quality. The WADL XML schema or RelaxNG schema is far better suited for gaining a better understanding of the tree, as the implementation strictly follows the structure of the schema.
4.3.3.6 WADL upload

Via the WadlUploadDialog class a WADL file gets forwarded to the WadlParser class. This class then instantiates the ApplicationNode, and then dependent on whether the particular node exists in the WADL or not, the particular nodes get instantiated and are added to the abstract syntax tree representation with the application node as a root element. The WADL structure is checked during runtime for allowed child nodes and valid attributes. As soon as the first error is detected, the analysis is interrupted with an appropriate error message. In the end a reference to an ApplicationNode is passed to the XmlTree class, that then creates a graphical tree-like representation of the Web Service.

4.3.3.7 Code generation

Probably the most important task in REST Describe & Compile is the compilation of WADL files to programming languages. The compilation process, however, is rather simple. The central component is the CodeGenerator factory. Depending on the selected language, the appropriate generator class is loaded. The generator factory and the concrete generators communicate by means of a so-called RequestMessager class after the syntax tree (which is the internal representation of the graphical WADL XML tree) has been traversed. The RequestMessager contains the parameters, the request name, the HTTP method, and the absolute address (i.e. the URI without the parameters). Faults are passed to the generators as separate HTTPError objects. In case of statically typed languages such as Java and C#, a TypeMapper class maps the XML schema types to the language's native types. Each target language has a templates class that contains templates for the static code generation fractions. The diagram below shows the Java generation sequence.
4.3.4 Additional remark on the sequence diagrams

The sequences outlined above can be combined in whatever sense-making combination, for example GUI Setup followed by WADL Upload followed by XML Tree Creation and finally Code Generation. Another sequence would be GUI Setup followed by Request URI Analysis.
5 Implementation challenges

5.1 Type estimation in REST Describe

Type estimation is an important element of request analysis. In the following the approach of
REST Describe & Compile is described. It is based on a fall-through hierarchy of regular
expressions. The estimation process takes both the parameter value and the parameter
name into account. This allows for pretty good estimation with the process judging its own
quality. Custom types are a bit of a problem (for example a type sort with the allowed values
\{date, relevance\}). These parameters look like strings, but - besides a potential XML schema
analysis - based on pure example value analysis there is no way to determine whether only
certain values (custom type), or all values (general string) are allowed. At this step of the whole
analysis process the application runs completely on the client side. Because of the Same Origin
Policy, schema retrieval would require a server side component, therefore at this step the default
type estimation for this kind of parameters is string, however, just with estimation quality
"supposed" (and not "sure").

The diagram below contains the complete fall-through hierarchy, where the green, yellow, and
red boxes symbolize the type estimation at the particular point, green meaning "sure", yellow
"supposed", and red "unsure". The left side of the diamonds means "condition fulfilled", the
right side means "condition not fulfilled".
5.1.1 Observations during type estimation

In practice the following observations can be made:

- **Only parameters without any value fail to be analyzed**, their type is "N/A" and the estimation quality in consequence "unsure".

- **Most parameters default to string**, this is reasonable and given the limitations with custom type estimation not a big surprise.

- **Language and country code estimation works pretty well**. This is due to the two-level checks which includes parameter name and value analysis.

- **Integer and float estimation works very well**, these are easy to determine by a regular expression.
• **The same argumentation applies for URIs** as the prefixes "http(s):/" or "www" are clear indications.

• **Two-level analysis**, e.g. taking both the name and the value into account **allows for fine-grained estimations**. For example `start=1` would be estimated to be integer ("supposed"), whereas `isEnabled=1` would be estimated to be boolean ("sure") for the "is" prefix and the "ed" suffix in the parameter name, which are clear indicators for boolean values.

### 5.1.2 The role of types

The first additional remark concerns types with Web Services in general. When you take an online search Web Service with an imaginary parameter `results=10`, type estimation will detect that the 10 is an integer. But what does this mean? In the end, when the request goes on the wire, you are always sending a string. Whatever types your Web Service client code enforces on the user interface (this means on the constructors of the request classes), there must be a **string representation of the data** for internal processing.

Second remark; imagine a language parameter that allows a search to be restricted to a subset of languages. In a programming language you would create a list or an array of the allowed languages, but how should an array or a list be mapped to a string? In practice there are two ways to represent this in the REST world:

• The first variant is to have a **parameter appear several times**. This results in a request like:
  ```
  ...&language=en&language=fr&language=de...
  ```

• The second, however far less applied, variant is to introduce some kind of **separation character**, for example comma (","), or space (" "). After URL-encoding the special character (in the case of "," the URL-encoded form is "%2C"), this results in a request like:
  ```
  ...&language=en%2Cfr%2Cde...
  ```

### 5.2 Code generation with Java properties files

One of the more interesting features of REST Describe & Compile is the way code generation is implemented. The application is created with Google Web Toolkit. The toolkit's I18N Message feature is based on Java properties files and provides us with a certain kind of template functionality. How does this work? Let us start with an explanation of the feature the way it was thought (i.e. for I18N):

You start with a template (that is a Java properties file) in each language:

```java
runningOutOfDiskSpace = Caution, you only have {0} left. # EN 
runningOutOfDiskSpace = Achtung, Sie haben nur noch {0} übrig. # DE
```

At runtime the program inserts the actual dynamic amount of available disk space, and only the static strings that need to be localized are kept in the properties file. Now back to the code generation idea. For instance functions or methods, or whatever you want to call them, exist in almost every programming language. There are slight differences, however, the main idea is the same. Have a look at two equivalent functions, the first in Ruby and the second in PHP:
def welcome(name)
    puts "howdy #{name}"
end

function welcome($name) {
    echo "howdy" . $name;
}

So "}" becomes "end", "def" becomes "function" and so on. For each language there are characteristic static structures, and only the dynamic content needs to be updated. What REST Compile does is that a CodeGenerator factory class loads for example a PHP5Generator class that then in turn loads its particular Templates_PHP.properties file. In consequence if another language shall be added, the new generator simply needs to be registered in the CodeGenerator factory, a new template file has to be created, and finally the dynamic code generating functions have to be added to the generator, and done. This I18N message abuse allows code generation to happen completely on the client side. The whole application could be downloaded and be ran offline.

5.2.1 Code generation in practice

In the following the way REST Describe & Compile generates code will be presented by means of a concrete example. In order to keep the example as short as possible, we describe code generation with a dynamically typed language: PHP 5.

5.2.1.1 Static parts of the code

The process starts with a static Java properties file with the following contents:

```java
# abstract rest request super class
restRequestClass = 
abstract class RestRequest {
    
    /\ constructor 
\tpublic function __construct() {
    
    /\ the POST function 
\tpublic function doPostCall($request, $postArgs) {
    
    \t\t// initialize the session 
\t\t$session = curl_init($request);
    
    \t\t// set cURL options 
    \t\tcurl_setopt ($session, CURLOPT_POST, true);
    \t\tcurl_setopt ($session, CURLOPT_POSTFIELDS, $postArgs);
    \t\tcurl_setopt($session, CURLOPT_HEADER, true);
    \t\tcurl_setopt($session, CURLOPT_RETURNTRANSFER, true);
    
    \t\t// do the POST and then close the session 
    \t\t$response = curl_exec($session);
    \t\tcurl_close($session);
    
    \t\treturn $response;
    
    \t\t// the GET function 
    \tfunction doGetCall($request) {
    
    \t\t// initialize the session 
    \t\t$session = curl_init($request);
```

Code generation with Java properties files
Automatic Multi Language Program Library Generation for REST APIs

```java
// set curl options
curl_setopt($session, CURLOPT_HEADER, true);
curl_setopt($session, CURLOPT_RETURNTRANSFER, true);

// do the GET and then close the session
$response = curl_exec($session);
curl_close($session);
return $response;
```

5.2.1.2 Dynamic parts of the code

This file describes the static part of a RestRequest super class. The dynamic part is added at the very end of the file, finally replacing the `{0}` placeholder. Now let’s have a look at the Java code that will fill this gap. The idea is to insert a case statement for each HTTP error that may occur, as documented by the API’s textual description, and in consequence as reflected by the corresponding WADL.

```java
/**
 * @param faults
 * @return
 */
private String prepareFaults(Vector faults) {
    String errorHandling = "";
    if (faults != null) {
        Iterator faultsIterator = faults.iterator();
        while (faultsIterator.hasNext()) {
            HTTPError fault = (HTTPError) faultsIterator.next();
            errorHandling += "\t\t\tcase " + fault.getStatus() + ":\n\t\t\ttrigger_error(" + fault.getMessage() + ", E_USER_ERROR);
            "\t\t\tbreak; // exit with error code " + fault.getStatus() + "\n";
        }
    }
    return errorHandling;
}
```

What this function does is that it creates a long string of all the errors that can occur. Therefore the function iterates over a vector of faults that has been created based on the WADL file, and concatenates all the error handling operations to the error handling string.
5.2.1.3 Bringing dynamic and static parts together

The final step consists of bringing the two pieces together. This happens via the Google Web Toolkit I18N message class as outlined before. In practice the merging happens simply by calling the appropriate I18N interface method with the dynamic part as a parameter which will be inserted into the properties file interface at the position of the \{0\} placeholder, just as any normal internationalization message would.

```java
// the REST request super class
String restRequestClass =
    templates_PHP.restRequestClass(errorHandlingString);
```

The result is a third string with the joined dynamic and static parts (now a string representing a complete PHP class), that in turn has to be integrated into a PHP file template. Little by little this step-wise process constructs a complete set of classes.

5.2.1.4 Code generation and statically typed languages

For languages like Java or C# code generation is a bit more complicated, basically because these languages **miss a way to pre-assign values in constructors**. In dynamically typed languages this is usually allowed. Consider the following PHP constructor:

```php
public function __construct($_newParam = "abc") {
    // code
}
```

The pre-assignment of the variable (in this case the default value is "abc") allows the constructor to be called with zero, or with one argument. In Java, this needs to be emulated by providing two constructors, one with zero, and the other with one argument.

Another issue is that **getters and setters need to be explicitly created**. Other languages allow dynamic getters and setters, for example very elegantly in Ruby:

```ruby
# getter and setter methods
attr_accessor :newParam
```

A last challenge concerns the **requirement for each type to be toString()-able** that means that even if to the outside world a variable is of a certain type A, internally when the request goes on the wire this type A needs to be represented in string form.

All these issues are not very difficult to deal with, however, given the elegance of dynamically typed languages, development of a code generator for statically typed languages is rather annoying, and forces the programmer to think of more things than actually seems necessary at a first glance.

5.3 Dog food in REST Describe & Compile

5.3.1 Quality factors for auto-generated WADL files

The quality of the auto-generated WADL files depends on various factors:

- **Quality of the REST API:**
  REST is all about structuring resources in a sense-making way. The better an API provider has organized his resources, the better will the sample requests reflect this structuring, and
in consequence the WADL. APIs that follow the "true REST" pattern that was introduced in chapter 2 have this property as an intrinsic feature. APIs that follow the "parameterized operation" pattern usually have nothing else than the operation name that could be used as an ordering property.

- **Quality of the request samples:**
The more expressive a given sample request for an API call is, the more potential parameters get covered by the example. In addition to that, the more typical the given sample parameter values are, the better the implication of the general case, and in consequence the estimated type.

- **Number of sample requests:**
The more requests are available for a certain API, the better REST Describe & Compile can optimize the generated WADL by sharing common objects, and get a more fine-grained view on the whole API.

By a pure sample request analysis a lot of properties of a REST API such as resource structure, parameter types, HTTP method etc. can be found. These properties have been treated in the chapters before. However, there is another factor that has not been taken into account yet. Only **executing an API call** gives you a real feeling of how an API behaves, especially what (if at all) XML namespaces it uses, how it reacts on errors, the response times, etc. In the following we want thus introduce what we call the "dog food" principle for REST Describe & Compile.

### 5.3.2 The dog food principle for REST Describe & Compile

"Eating its own dog food" is a common term in the computer industry for using the company's own products in-house, for example Microsoft operating their server infrastructure with IIS on Windows, and not for example Apache on Linux. When a company eats its own dog food, this in generally means that the company trusts the quality of its products. This applies for REST Describe & Compile. We are convinced that the generated code is ready for production use, and therefore are up to using it on our own. In the following we will describe how we have implemented this feature.

### 5.3.3 Implementation of the dog food principle

In a first analysis run, the basic API structure is known. Together with a parameter type analysis, and given the fact that the sample requests usually represent valid API requests, a real API call can be made in order to get some more in deep knowledge of the API's behavior. In short, the exclusive request analysis is combined with a response analysis. This allows for the API's XML namespaces, XML schemas, and potentially error handling mechanisms to be discovered. Technically one of the arising problems is security, as due to the Same Origin Policy JavaScript cannot directly access third party Web Services. Thus, Web Service calls need to be proxied. For the present implementation we decided to extend the already existing PHP 5 code generator and have our Web Service calls be proxied by a local PHP 5 server. Basically the dog food PHP 5 code and the normal PHP 5 code differ only in smaller details. In the following these differences will be examined.

#### 5.3.3.1 Description format for discovered data

One major addition to the normal PHP 5 code concerns XML processing of the discovered data. For REST APIs response analysis there are four pieces of information of interest. These are

- **Default namespace:**
The XML default namespace is usually a company-defined namespace that differentiates their API's XML code from other site's XML.
Other namespaces:
Often API providers use other namespaces in their response XML. Prominent examples are the XPath namespace or the XHTML namespace.

Schema locations:
If companies define own namespaces the schema locations provide a link to the XML schema for users to download and parse it for data processing.

HTTP errors:
REST APIs by definition should use existing and well-defined HTTP status codes\(^\text{24}\) to signal errors or unexpected behavior.

In order to describe this information in a machine-readable way we have defined our own small XML service meta data description format. In the following a short example:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<serviceData xmlns="http://code.google.com/p/rest-api-code-gen">
  <requestData location="http://api.cnet.com/restApi/v1.0/category">
    <validRequest status="200">
      <schemaLocation/>
      <defaultNamespace>
        http://api.cnet.com/restApi/v1.0/ns
      </defaultNamespace>
      <otherNamespaces>http://www.w3.org/1999/xlink</otherNamespaces>
    </validRequest>
    <erroneousRequests>
      <erroneousRequest status="400">
        <schemaLocation/>
        <defaultNamespace/>
      </erroneousRequest>
      [...]
    </erroneousRequests>
  </requestData>
<serviceData>
```

The main element is `serviceData` with a child element `requestData` for each request, where the request's URI is included in the `@location` attribute. The requestData element has two child elements: `validRequest` and `erroneousRequests`, where it is to be noted that the first is singular, and the second is plural. We will explain this decision later. Both elements in turn have the same child elements: `schemaLocation`, `defaultNamespace`, `otherNamespaces`, and finally a `@status` attribute indicating the HTTP status. The dog food PHP 5 code generates such XML representation during response analysis.

### 5.3.3.2 The art of provoking faults

With the original sample request there is normally one request available that results in a correct API call. Execution of such a valid request will result in a response with HTTP status code 200 (OK), and the namespaces of valid requests will be contained. Short, this case describes the scenario where everything goes fine. However, the cases of failure are often of a lot more interest. How does the API react if there are many calls in a short period of time (possible limitation of requests per second by IP), how does an API handle missing, or erroneous parameters or login data? In the following we will present strategies to intentionally cause faults in order to see how an API reacts.
Try to cause "missing parameter" errors:
In order to provoke this class of errors, there are two obvious strategies:

- **Leave out parts of the parameters:**
  The shortcoming of this strategy is that some parameters are optional, so it is quite common to hit a legal combination of optional and required parameters. In addition to that it is relatively "expensive" to create a combination of parameters to be included in the request, and parameters to be left out.

  Instead of calling
  
  ```
  GET http://example.com/api/operation?param1=value1&param2=value2
  ```
  
  we would call
  
  ```
  ```

- **Leave out all parameters:**
  This strategy has the advantage of being very simple, but still effective. Most API calls have at least one required parameter, so simply calling the API operation without any operation at all will very likely fail, and this is what we want.

  Instead of calling
  
  ```
  GET http://example.com/api/operation?param1=value1&param2=value2
  ```
  
  we would call
  
  ```
  GET http://example.com/api/operation.
  ```

Try to provoke "invalid parameter" errors:
This kind of error can be provoked in two ways:

- **Modify existing parameter values:**
  This strategy is very effective and quite simple to apply: simply add some sequence like the string "abc" to each value will very likely break the call.

  Instead of calling
  
  ```
  GET http://example.com/api/operation?param1=value1&param2=value2
  ```
  
  we would call
  
  ```
  GET http://example.com/api/operation?param1=value1abc&param2=value2abc.
  ```

- **Add additional parameters:**
  The goal of this strategy is to see how an API reacts on unexpected events. In order to apply this strategy, a new imaginary parameter like "&abc=abc" is added to the valid parameters.

  Instead of calling
  
  ```
  GET http://example.com/api/operation?param1=value1&param2=value2
  ```
  
  we would call
  
  ```
  GET http://example.com/api/operation?param1=value1&param2=value2&abc=abc.
  ```

Try to provoke "invalid resource" errors:
This family of errors is based on the assumption that a REST API might provide facilities to discover existing resources. We have tried out the following strategy:
Modify the resource location:
The idea of this strategy is simply to add a string like "abc" to the resource's base address. The results, however, have been rather disappointing with no API with which we tested this behavior providing any resource discovery facilities. Instead we just got the HTTP status code 404 (Not Found) back, without any additional resource location information.

Instead of calling
GET http://example.com/api/operation?param=value
we would call

The successful ones of these strategies in combination with the valid request give a relatively widespread view on how an API behaves. In order to also test the API behavior in case of repetitive calls within a very short period of time, we place the next call as soon as the response to the previous call has been received. Please note: this is different from "bombing" a Web Service with requests, as we wait for the answers before sending new requests.

These strategies are also good for breaking "authentication parameter" based APIs. An authentication parameter is a parameter that identifies a user with a special token, a key value, or some sort of password. We cover thus a broad spectrum of potential sources of errors:

- Invalid parameter
- Missing parameter
- Unexpected parameter
- Invalid resource
- Missing authentication data
- Repetitive calls

5.3.3.3 Workflow of the dog food feature

The extended workflow with the dog food feature consists of two phases: phase one that implies pure request-only analysis, and phase two that also analyzes the response.

- Phase 1 (request-only analysis)
  1. Analyze a request URI and generate a first request-based WADL
  2. Use this WADL file to generate request code

- Phase 2 (additional response analysis)
  3. Execute the generated code on the server, i.e. place a real API request
  4. Harvest the returned response and discover the contained namespaces, errors and XML schemas
  5. Refine the previously generated WADL and add the response-based WADL information
  6. Return the final version of the WADL
  7. Generate a final version of the code with proper response handling

Phase 1 runs completely on the client side. In contrast, phase 2 necessarily needs to be run on a server, that proxies the API call (due to the Same Origin Policy that limits JavaScript's level of access). The connection between the two worlds (web application, proxying server) is realized via Ajax. REST Describe & Compile places an Ajax request to the dog food PHP 5 code generator. This code then gets forwarded to a proxy component on the server side. Upon reception the contained code gets executed on the server, and the XML response gets created. The waiting Ajax request can then finally parse the returned XML document that has been defined above.
6 Conclusion and prospect

6.1 Summary

In the first chapter we introduced the necessary definitions and presented some general thoughts on Web Services. Chapter two depicted the problem space of REST Web Services and the way resources can be used. We proposed a classification of REST Web Services and also how the different classes map to generated code. This led us to the conclusion that there is a need for a REST description language. In the following chapter three we presented and analyze a number of such description languages, where the language of our choice, namely WADL, was given special attention. Chapter four began with a design document that defines the requirements of a WADL editor and compiler software application. Afterwards we showed the concrete internal and external architecture of the application, supported by some UML sequence diagrams that illustrated the interaction of the involved classes. Finally chapter five was dedicated to the implementation challenges like code generation, type estimation and the "dog food"-based response analysis extension.

6.2 Conclusion

6.2.1 Code generation

**Code generation based on nothing more than sample requests as an input source works surprisingly well.** The application is able to create working code in PHP 5, Ruby, Python, and Java. As a general observation it can be noted: the better the request samples, the better the generated code. A request with significant values is easy to analyze both from a type estimation point of view, and also from a completeness point of view. The more parameters are present in a sample, the more features of an API can be used directly without fine-tuning the generated WADL. Most REST APIs these days use just GET and POST from the available set of HTTP methods. For these APIs the generated code works very well. PUT and DELETE are not supported yet, however, can be easily added, as just the super class generators of each target language need to be modified. Depending on the API, the generated code is able to handle API errors properly by throwing exceptions that correspond to the HTTP error classes of the API. The generated code supports also advanced features like parameter chaining due to setters that return an instance of the object. This allows both parameter sequence-oriented instantiations, and chained setter-oriented instantiations of classes.

**Code generation based on request and response analysis as an input source works very well.** The additional effort of response analysis pays off quickly. The supplemental information gains (XML namespaces, XML schemas, fault behavior) allow for more accurate code to be generated. In addition to that special API behavior (like call limits or non-standard error handling) can be detected.

**Code generation based on uploaded WADLs as an input source works very well.** Depending on the quality of the WADL, the generated code is able to deal with exceptions, handle different HTTP errors appropriately, and the generated code is thus a solid code base to built on top of.

6.2.2 Type estimation

**The presented approach of a two-way fall-through hierarchy of regular expressions is simple, however very powerful.** Given the many ambiguous situations (is 1 a boolean, or an
integer value, is an access key value of 1234567890 a string, or an integer value; what type is a value that may be either 1, or else must be null, to name just a few of the issues) the results are within the expected scope. The thing is that with REST types do not matter as much as they matter with SOAP/WSDL. In case of doubt it is always save to default to a string-based type without losing much significant information.

The biggest issues are API-specific custom types (like sort = {date, relevance}) that cannot be detected by a request-only analysis, and in case where there is no request XML schema, neither can be detected by an additional response analysis. At present there are two ways to handle this: ignore the custom type and default to a string-based type, or manually define the valid options. For now we decided to go for the first option.

6.2.3 WADL

WADL as a (REST) Web Service description language has passed the acid test. In all our tests with various APIs of all categories ("true REST", "parameterized operation", both dynamic and static variants) the language is expressive enough to describe the APIs themselves and their behavior. Especially the language's capability to describe dynamic resources (like .../userId/1234/...) makes it a realistic candidate for becoming the REST description language.

WADL generation and WADL editing work smoothly. The editor is a valuable tool for relatively quick WADL creation and editing. It is context-sensitive and just offers the particularly allowed operations in accordance with the WADL specification. This guarantees the generated WADL files to be valid. Uploaded WADLs are parsed for correctness and can be easily edited once they have been imported. For bigger files, it is however difficult to overlook the whole WADL structure. The application's code folding feature is of great help, but it is just the nature of things that from a certain WADL file size on, separation is inevitable. Due to the resource-oriented structure of WADL this is a straightforward process that can be triggered with an appropriate selection of sample requests.

6.3 Prospect and future areas of research

6.3.1 Does REST need WADL?

One of the fundamental questions that has been discussed quite passionately in the blogosphere is whether REST needs a description language or not. As there is basically just one serious candidate for such a language, the question can be reduced to "Does REST need WADL?" We as the authors of REST Describe & Compile obviously believe the answer to be "yes", or at least "it does not hurt". We have shown that code generation indeed does work well, even if the WADL file is just generated on the fly based on request URI analysis and response parsing. No one will expect an automatic code-generating system to create applications that can be used just out of the box. Code generation has always been a utility, not a complete solution. Of course you will need to build your own classes around the generated ones, however, code generation will save you the time-consuming and cumbersome task of doing stupid things by yourself that could be done by a machine. We believe that REST Describe & Compile does a not too bad job on that, run the del.icio.us API demo that is built in the application as a prove. We thus feel like heading in the right direction if we continue our work on REST Describe & Compile. WADL may not be the silver bullet of REST Web Service description, but it does a more than solid job on describing such services. Hopefully REST Describe & Compile will help to strengthen its position.
6.3.2 Improve the quality of the generated code

At present the quality of the generated code is already not bad. It supports each language's exception handling mechanism to deal with faults, it supports parameter chaining, and it is completely object-oriented. However, there is still a long way to go. Not only is the pure quantity of languages to be augmented (1st priority: add C# support), but also the code quality needs to be further improved, especially the XML-handling where true namespace and schema support are on top of the queue. For Java this will probably mean the integration of JAXB\textsuperscript{25}, for other languages there have to be found equivalent approaches.

6.3.3 Improve type estimation

The issues with API-specific custom types outlined before can be easily ignored since the APIs themselves should handle these errors, however, it would be much nicer to have support for this kind of type safety already on the client side. In Java there is the enum construct that provides a language-level implementation of enumerated types, and other languages as well support enumerated types either natively or support can be easily added.

The pure presence of enumerated type support does not yet mean that the valid enumeration values are known by means of automated type estimation. Given the absence of XML schema for the request parameters for most of today's APIs, the only way to find out these values is the analysis of the human-readable documentation. This could be done via screen scraping, however, this approach would be rather brittle, but maybe worth a try. Most API documentation is auto-generated with some kind of template system. Once the screen scraper has understood the structure, further analysis might eventually be reduced to an XPath pointer into the documentation in order to find the valid values.

6.3.4 Improve the GUI

At the time when we started our work on REST Describe & Compile, the underlying Google Web Toolkit (GWT) was released in version 1.3.3, half a year later release candidate 1 of version 1.4.4 has been released, and with the new version come new widgets that will definitely find its way into the application, such as the split panels that allow the layout to be more flexible by dynamically resizing the panels of interest, like the sample request panel on sample insertion, the WADL panel on WADL editing and so on.

Up- and download that at present are realized by simple textareas could be made much more comfortable by adding server-side processing like true file upload, and zipped archives download.
## Annex A

### Time schedule

<table>
<thead>
<tr>
<th>Date range</th>
<th>Task</th>
<th>Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 Feb – 23 Feb</td>
<td>Read about code generation, get to know the concepts and terms. Get to know Google Web Toolkit, do some &quot;hello world&quot; things.</td>
<td>Code generation books (see the printed publications in the bibliography) Google Web Toolkit Eclipse</td>
</tr>
<tr>
<td>26 Feb – 09 Mar</td>
<td>Write the general introductory parts of the report. Code the scaffold of the application, do basic things like syntax highlighting, DOM operations and tree widget operations.</td>
<td>RESTful Web Services resources online Google Web Toolkit Eclipse</td>
</tr>
<tr>
<td>12 Mar – 31 Mar</td>
<td>Read as much as possible on REST description, get to know the key persons' opinions. Read, understand and question the WADL spec.</td>
<td>WADL spec RESTful Web Services resources online</td>
</tr>
<tr>
<td>02 Apr – 27 Apr</td>
<td>Code a first working beta that covers at least the REST Describe part.</td>
<td>WADL spec Google Web Toolkit Eclipse</td>
</tr>
<tr>
<td>30 Apr – 04 Mai</td>
<td>Write down everything you have learnt, create an overview of available Web Service description languages, provide samples in order to understand the specs.</td>
<td>WADL spec RESTful Web Services resources online</td>
</tr>
<tr>
<td>07 Mai – 25 Mai</td>
<td>Implement code generation in at least PHP, others as possible. Refresh your Python and Ruby knowledge.</td>
<td>WADL spec Various Python and Ruby resources and tutorials Google Web Toolkit Eclipse</td>
</tr>
<tr>
<td>28 Mai – 05 Jun</td>
<td>Improve code generation quality, add at least Java support, remember the encountered difficulties with statically typed languages.</td>
<td>WADL spec Google Web Toolkit Eclipse</td>
</tr>
<tr>
<td>06 Jun – 18 Jun</td>
<td>Finish the report, add UML sequence diagrams in order to clarify the architecture, add screenshots, explain what might be unclear, review the report. Explain type estimation and code generation, create diagrams where necessary.</td>
<td>WADL spec Poseidon for UML OpenOffice</td>
</tr>
<tr>
<td>Date</td>
<td>Activity</td>
<td>Tools/Software</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>19 Jun</td>
<td>Present at ENSIMAG University, Grenoble, France</td>
<td></td>
</tr>
<tr>
<td>20 Jun – 26 Jun</td>
<td>Prepare slides for TSS-JS</td>
<td>Power Point</td>
</tr>
<tr>
<td>27 Jun</td>
<td><strong>Presentation at The Server Side Java Symposium, Barcelona, Spain</strong></td>
<td></td>
</tr>
<tr>
<td>02 Jul – 06 Jul</td>
<td>Start implementing the dog food feature</td>
<td>Eclipse, Google Web Toolkit</td>
</tr>
<tr>
<td>07 Jul – 08 Jul</td>
<td>Improve code, fix HTTP basic auth in Ruby</td>
<td>Ruby doc</td>
</tr>
<tr>
<td>09 Jul – 14 Jul</td>
<td>Add dynamic resources feature</td>
<td>Eclipse, Google Web Toolkit</td>
</tr>
<tr>
<td>16 Jul – 19 Jul</td>
<td>Produce YouTube screencasts, make PR</td>
<td>Movie editing software</td>
</tr>
<tr>
<td>20 Jul – 23 Jul</td>
<td>Find bugs, test with as many APIs as possible, create a repository of demo requests</td>
<td>Various REST APIs, talk to John Musser from ProgrammableWeb.com</td>
</tr>
<tr>
<td>24 Jul – 29 Jul</td>
<td>Fix bugs, get the final beta ready for the presentation, freeze beta 0.7.1</td>
<td>Eclipse, Google Web Toolkit</td>
</tr>
<tr>
<td>30 Jul – 02 Aug</td>
<td>Finalize report, correct typos, print, binding</td>
<td>Ink cartridges, copy shop</td>
</tr>
<tr>
<td>03 Aug</td>
<td><strong>Presentation at University of Karlsruhe, Germany</strong></td>
<td></td>
</tr>
</tbody>
</table>

---

In Annex A, the document presents a detailed schedule for a project involving automatic multi-language program library generation for REST APIs. The timeline includes key dates and activities such as presentations, software development tasks, and preparation for YouTube screencasts and bug fixes. The activities are spread across June and July, with specific tools and software mentioned for each task.
Bibliography

Foreword

REST, and in consequence REST APIs, are a very young area of research. There is not much printed documentation available. Richardson's and Ruby's "RESTful Web Services", that can be seen as the REST primer, appeared just in May 2007. At this time we had already reached the implementation phase (see Annex A). The subject being so new, REST resources are rather blogs, mailing lists, or articles, than books. It is to be noted that the informality of the REST discussion does not mean that this discussion is not conducted in a professional way. The reverse is true. It means just that REST topics are usually not being discussed in scientific papers, but rather in just as scientific, however a little more easy-going blog entries. In the blogosphere there can be met people like Tim Bray (XML co-author), Rich Salz (named contributor to the HTTP/1.0 and HTTP/1.1 specs), and many other engineers at Yahoo, IBM, Sun, or Google to name but a few. Blogs have a tradition of so-called "Permalinks", permanent links that are meant to stay. For this reason, then for the absence of printed resources, and finally for the high quality of the referenced resources, we have decided to create a list of electronic sources below.

Printed publications

- **RESTful Web Services**, by Leonard Richardson and Sam Ruby, O'Reilly Media Inc. (May 8, 2007), ISBN 978-0596529260

Electronic sources

All links have been successfully tested on August 1st, 2007.

---


2 List of available SOAP toolkits: http://soaplite.com/resources.html#TOOLKITS

3 The S stands for Simple: http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for-simple


6 Jakob Nielsen's Alertbox on URLs as UI: http://www.useit.com/alertbox/990321.html
Some examples of textual and/or XML schema REST API descriptions:

- Amazon (Item Lookup): http://docs.amazonwebservices.com/AWSEcommerceService/2007-02-22
- del.icio.us (Tag Rename): http://del.icio.us/help/api/tags
- Flickr (Add Tags): http://www.flickr.com/services/api/flickr.photos.addTags.html
- Yahoo Photo Service (List Photos): http://developer.yahoo.com/photos/V3.0/listPhotos.html

8 Sun's Norman Walsh on the complexity of WSDL: http://norman.walsh.name/2005/02/24/wsdlp8

9 List of REST Web Service descriptions maintained by David B. Orchard: http://www.pacificspirit.com/Authoring/REST

10 W3C overview on how to describe REST Web Services with WSDL 2.0: http://www.w3.org/2005/Talks/1115-hh-k-ecows/#(1)

11 WRDL (Web Resource Description Language): http://www.prescod.net/rest/wrdl/wrdl.html

12 NSDL (Norm's Service Description Language): http://norman.walsh.name/2005/03/12/nsdl

13 SMEX-D (Simple Message Exchange Descriptor): http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX-D


16 WDL (Web Description Language): http://www.pacificspirit.com/Authoring/WDL

17 WADL (Web Application Description Language): https://wadl.dev.java.net

18 Google Checkout API: http://code.google.com/apis/checkout/index.html

19 eBay REST API: http://developer.ebay.com/developercenter/rest

20 Do we need WADL: http://bitworking.org/news/193/Do-we-need-WADL


24 List of W3C-defined HTTP Status Codes: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html