Validation and Interactivity of Web API Documentation

Peter J. Danielsen
and Alan Jeffrey
Bell Labs
Alcatel-Lucent
Naperville, IL, USA

Abstract—**Many Web APIs (by which we mean ones using HTTP as the application protocol) do not publish a machine-readable API description (in a language such as WADL or WSDL) but only provide human-readable documentation, usually in HTML. This documentation may be machine-generated, or it may be hand-edited in which case there is the possibility of errors being introduced into the API description. In this paper we present a Web Interface Language (WIfL) vocabulary for API documentation, which is intended to be embedded in HTML using RDFa annotations. We present the semantics of WIfL, including a formal presentation of inheritance and validation. We discuss our WIfL tools, which include a dynamically generated console for interacting with an API's reference implementation, and a validator which can check an API for internal consistency.**

I. Introduction

In recent years, the number of web applications offering Application Programming Interfaces (APIs) has increased dramatically. Figure 1 [1], shows the rapid growth in the number of APIs listed in the ProgrammableWeb directory since 2005. Figure 2 [2], breaks down the types of APIs in the directory and shows that REST[1] [3] is in the majority and is growing faster than any other style.

When an application exposes its data through a web API, it enables the creation of other applications that consume the data. The API becomes more valuable as the number of applications that use it increases. A key to converting potential consumers to actual consumers is to make it easy for them to learn the API and to try it. One way to do this is through accurate, complete, interactive API documentation.

This paper focuses on improving web API documentation in its most common format found on the web: HTML [4]. While other formats have been designed to document APIs (such as WSDL [5] and WADL [6]), the majority of web APIs today are documented in English prose, formatted in HTML, and do not follow a convention for machine-readable processing. We discuss embedding a machine-readable API description in the HTML using RDFa [7], and two advanced documentation features it enables: an interactive console and document validation.

[1]ProgrammableWeb API styles are self-reported by the API owners, and APIs reported as RESTful may not satisfy all of Fielding's architectural requirements. To avoid terminological arguments, we will use the phrase "web API" to describe APIs which are using HTTP as an application protocol.

II. Requirements

Looking at example API documents, such as the Twitter API or the ProgrammableWeb API, we discover some features of API descriptions in HTML.

Firstly, the authoring workflow for different APIs may be quite different. Some APIs (such as Twitter's) are machine generated, and some (such as ProgrammableWeb's) are hand-crafted. The common artifact in both cases is the HTML document itself, which motivates our interest in embedding machine-readable API descriptions into the human-readable HTML.

Secondly, the logical structure of the API does not necessarily align with the structure of the API document. For example, query parameters which are shared among all requests (such as an apikey) may be described only once in an overview section. Also, an API description is often split between many different pages, for example one page per resource class, together with an overview page. For this reason, we are interested in hyperlinked API descriptions, which allow the logical structure and document structure of an API description to differ.

Thirdly, it is quite common for API documents to include example client-server interactions, which are used to illustrate idiomatic uses of the API. A 2010 study [4] found that 75% of the studied APIs provided example requests and responses. Examples are often hand-crafted, and so are subject to typographic error, and to becoming out-of-date as the API specification changes.

These features lead us to our requirements: a machine-readable API description format which can be embedded into HTML, supports hyperlinking, and allows for specification and validation of example dialogs.

III. Model

A. Vocabulary

In Figure 3 there is an example API, rendered in a browser, together with its HTML source. This API shows some characteristics which are common in API descriptions.

The API is described as a collection of resource classes, in this case Store (the entire API), Products (the product catalog) and Product (an individual product for sale). Each resource class has an associated URI template [8], for example the Product template is http://example.com/store/api/products/{id}.

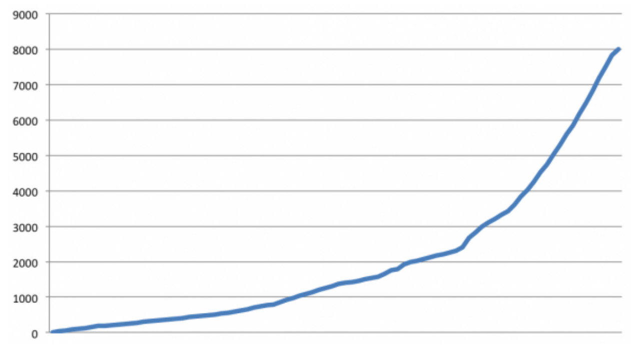


Fig. 1. API Growth 2005-2012

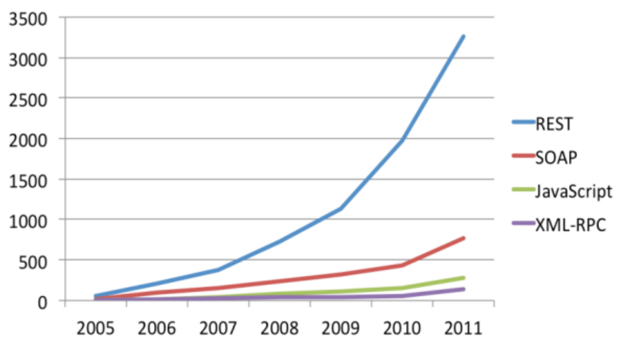
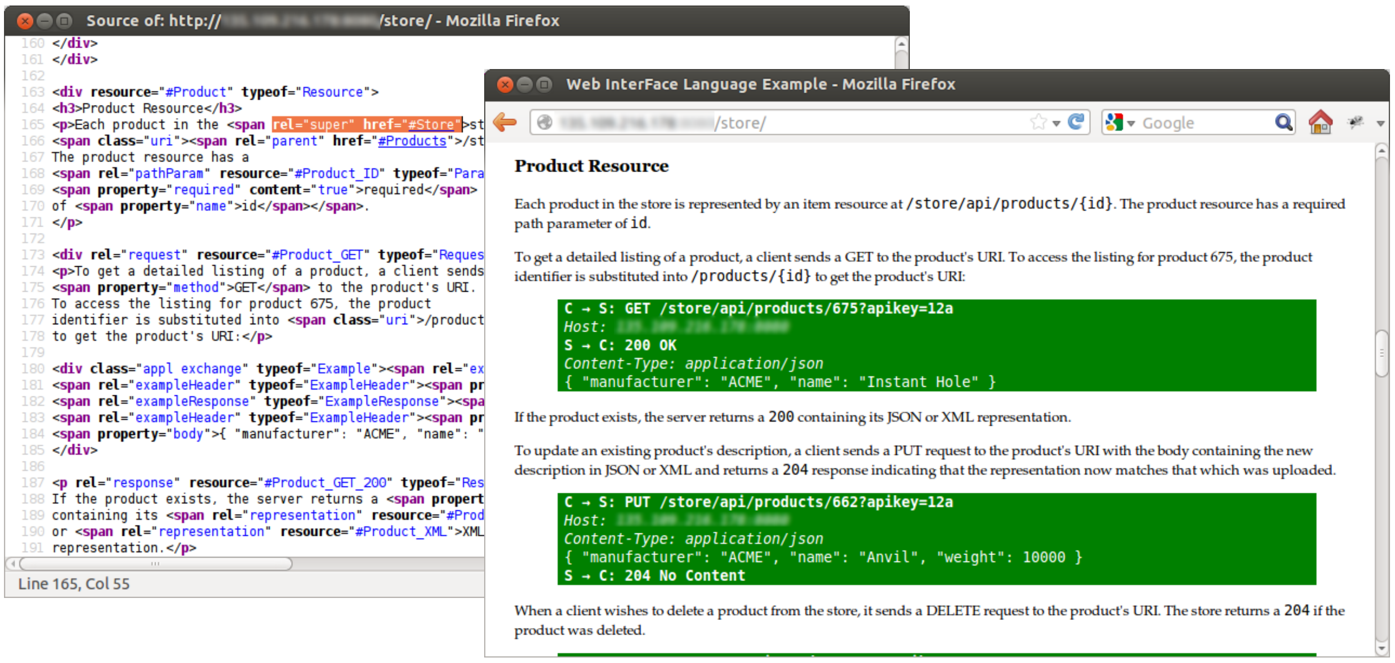


Fig. 2. API Protocols & Styles



```
</div>
</div>

<div resource="#Product" typeof="Resource">
<h3>Product Resource</h3>
<p>Each product in the <span rel="super" href="#Store">st
<span class="uri"><span rel="parent" href="#Products">/st
The product resource has a
<span rel="pathParam" resource="#Product_ID" typeof="Para
<span property="required" content="true">required</span>
of <span property="name">id</span></span>.
</p>

<div rel="request" resource="#Product_GET" typeof="Reques
<p>To get a detailed listing of a product, a client sends
<span property="method">GET</span> to the product's URI.
To access the listing for product 675, the product
identifier is substituted into <span class="uri">/product
to get the product's URI:</p>

<div class="appl exchange" typeof="Example"><span rel="ex
<span rel="exampleHeader" typeof="ExampleHeader"><span pr
<span rel="exampleResponse" typeof="ExampleResponse"><spa
<span rel="exampleHeader" typeof="ExampleHeader"><span pr
<span property="body">{ "manufacturer": "ACME", "name": "
</div>

<p rel="response" resource="#Product_GET_200" typeof="Res
If the product exists, the server returns a <span propert
containing its <span rel="representation" resource="#Prod
or <span rel="representation" resource="#Product_XML">XML
representation.</p>
```

Line 165, Col 55

Product Resource

Each product in the store is represented by an item resource at `/store/api/products/{id}`. The product resource has a required path parameter of `id`.

To get a detailed listing of a product, a client sends a GET to the product's URI. To access the listing for product 675, the product identifier is substituted into `/products/{id}` to get the product's URI:

```
C → S: GET /store/api/products/675?apikey=12a
Host: [redacted]
S → C: 200 OK
Content-Type: application/json
{ "manufacturer": "ACME", "name": "Instant Hole" }
```

If the product exists, the server returns a `200` containing its JSON or XML representation.

To update an existing product's description, a client sends a PUT request to the product's URI with the body containing the new description in JSON or XML and returns a `204` response indicating that the representation now matches that which was uploaded.

```
C → S: PUT /store/api/products/662?apikey=12a
Host: [redacted]
Content-Type: application/json
{ "manufacturer": "ACME", "name": "Anvil", "weight": 10000 }
S → C: 204 No Content
```

When a client wishes to delete a product from the store, it sends a DELETE request to the product's URI. The store returns a `204` if the product was deleted.

Fig. 3. Example API with WIfL annotations

URI templates can be expanded with a data binding to produce a particular resource URI, for example expanding the Product template with data binding { id: "123" } produces the URI http://example.com/store/api/products/123.

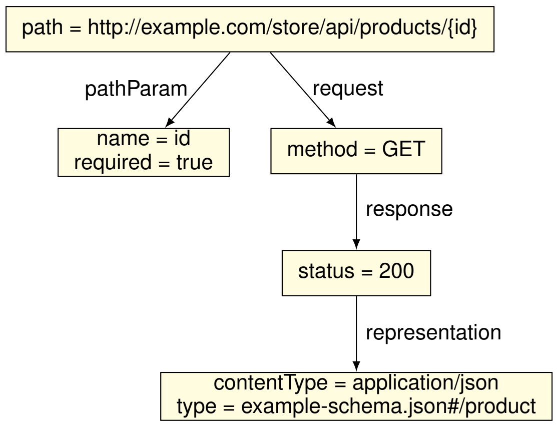
Each resource class has associated requests, with a method (GET, PUT, POST or DELETE), parameter descriptions (such as the id parameter of Product), and representations. The parameters are often typed (for example, id may be expected to be a non-negative integer) and the representations often have a schema (such as an XML Schema [9] or JSON Schema [10]).

Each request has associated responses, with possible status codes (such as 200 OK or 404 Not Found), header parameters and representations.

For example, the Product class has:

- URI template http://example.com/store/api/products/{id}, where the id parameter is mandatory, and
- a GET request, which has a 200 OK response, whose payload has media type application/json, conforming to the JSON schema example-schema.json#/product.

Expressed in the WIfL vocabulary, this is:



path = http://example.com/store/api/products/{id}
pathParam
request
name = id
required = true
method = GET
response
status = 200
representation
contentType = application/json
type = example-schema.json#/product

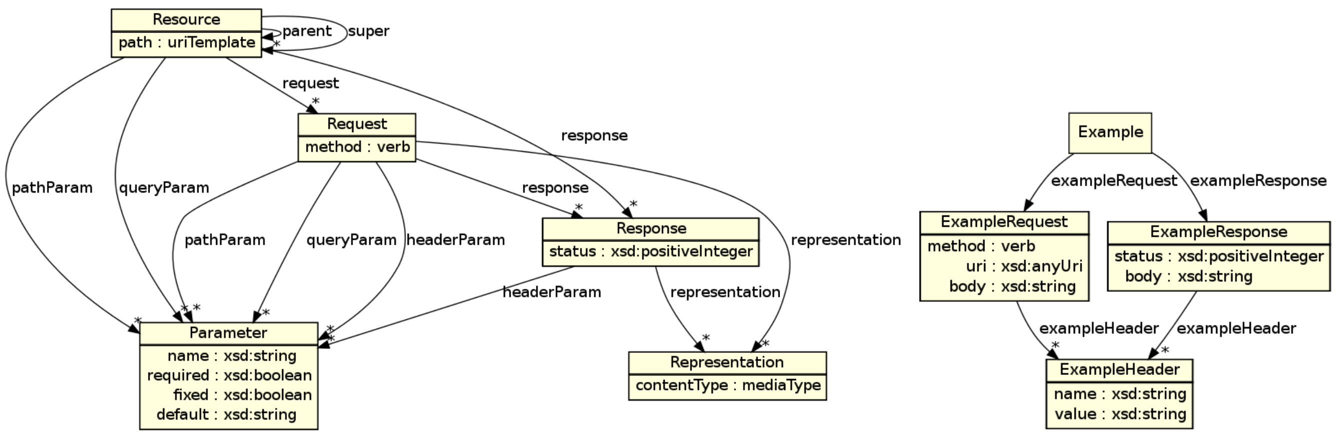


Fig. 4. WIfL vocabulary

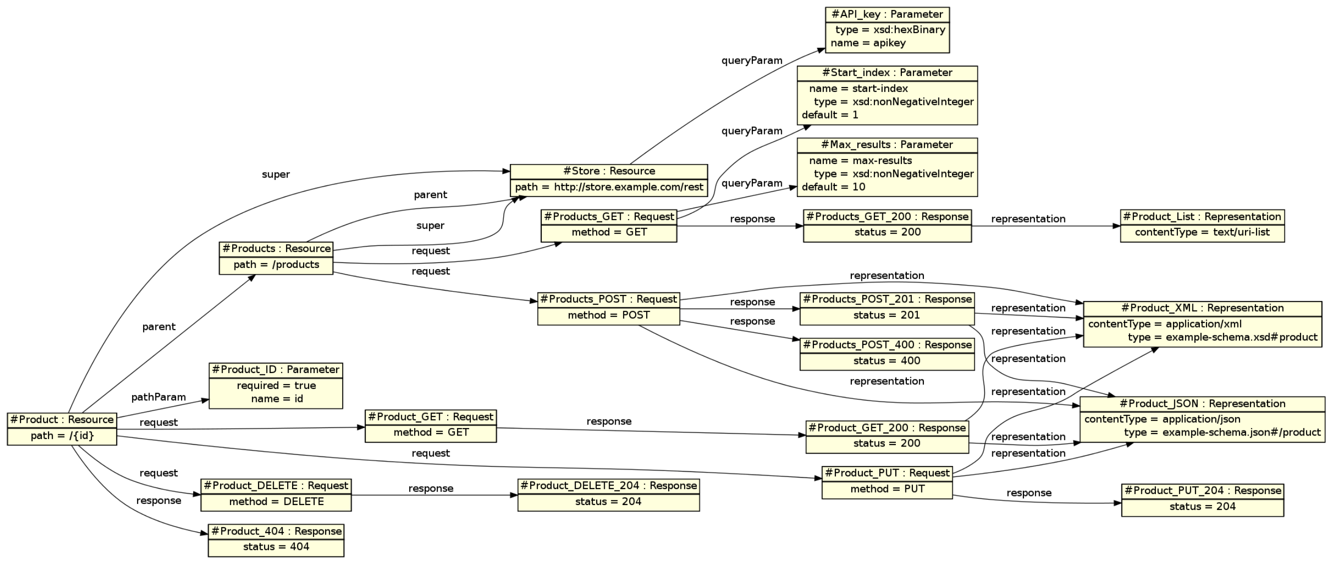


Fig. 5. Example API representation in WIfL

This graph can be embedded in the HTML description of the API using RDFa annotations [7]. For example the description of the JSON representation could be serialized as:

```
<p about="#Product_JSON">
  The representation has media type
  <span property="contentType">application/json</span>
  and should conform to our
 <a rel="type" href="example-schema.json#/product">schema</a>.
</p>
```

The vocabulary for WIfL API descriptions is given in Figure 4. The object diagram showing the RDF graph extracted from the example API description is given in Figure 5.

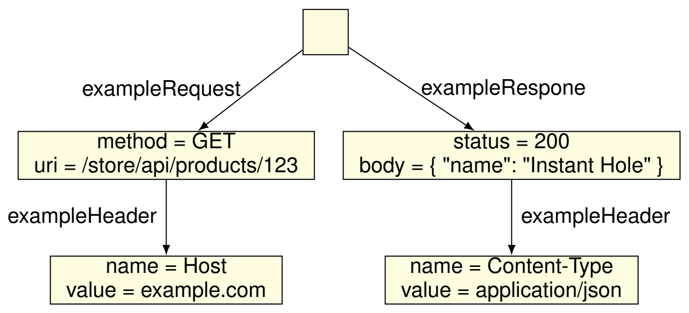
As well as API descriptions, WIfL documents can contain example client-server dialogs. For example, if a client sends:

```
GET /store/api/products/123 HTTP/1.1
Host: example.com
```

and the server responds:

```
200 OK
Content-Type: application/json
{ "name": "Instant Hole" }
```

then the dialog can be included in the HTML document by including the RDF graph:

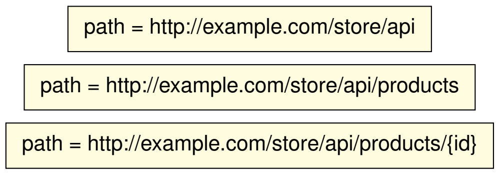


To support such example dialogs, we provide the Example class in Figure 4.

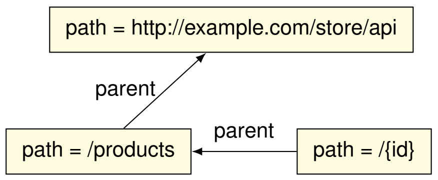
B. Inheritance

One problem with the basic WIfL model described in the previous section is that it violates the DRY principle (Don't Repeat Yourself).

The first violation is in the URI templates, for example there is a lot of repetition in our store example:


path = http://example.com/store/api
path = http://example.com/store/api/products
path = http://example.com/store/api/products/{id}

We reduce this in the same manner as WADL [6], by allowing resource classes to have an optional parent. A child's URI template may be relative rather than absolute, and it is resolved by its parent URI template. Since path parameters are given in the URI template, children also inherit path parameters from their parent. For example, the store resources are:



To normalize a WIfL description which includes parent inheritance, we use a deduction rule:

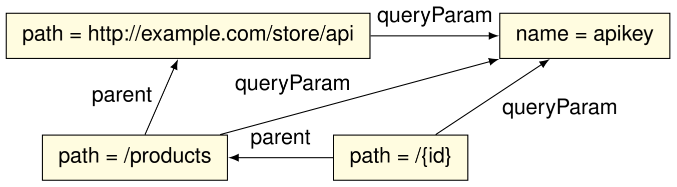
- if $p \xrightarrow{\text{parent}} q \xrightarrow{\text{pathParam}} r$ then $p \xrightarrow{\text{pathParam}} r$.

We also define a relationship $p \xrightarrow{\text{uriTemplate}} u$ (where p is a resource class and u is its URI template) as follows:

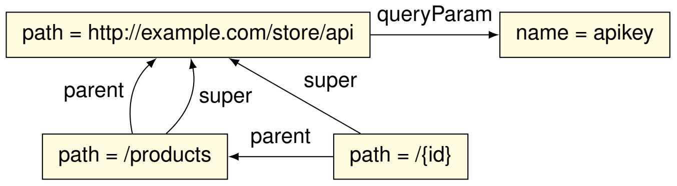
- if $\nexists q \,.\, p \xrightarrow{\text{parent}} q$ and $p \xrightarrow{\text{path}} u$ then $p \xrightarrow{\text{uriTemplate}} u$, and
- if $p \xrightarrow{\text{parent}} q \xrightarrow{\text{uriTemplate}} t$ and $p \xrightarrow{\text{path}} u$ then $p \xrightarrow{\text{uriTemplate}} tu$.

Note that resolution of URI templates is just given by concatenation: if a resource's parent has URI template t and the resource has path u then the resource's URI template is tu. We do not attempt to perform URI resolution [11] on templates, and indeed we suspect that no satisfactory[2] algorithm for URI template resolution exists.

The second violation is in the use of parameters, requests and responses. For example, RDF supports hyperlinking, so we can specify that every resource shares an apikey parameter, but to do so the hyperlink has to be present explicitly in every resource description:



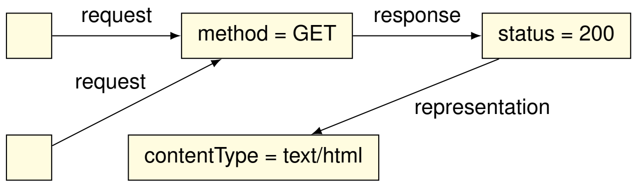
We also reduce this using a similar technique to WADL, by allowing resource classes to have superclasses, from which they inherit query parameters, header parameters, requests and responses (but not the path or any path parameters). For example, the store example is now further simplified:



The deduction rules for inheritance from superclasses are:

- if $p \xrightarrow{\text{super}} q \xrightarrow{\text{headerParam}} r$ then $p \xrightarrow{\text{headerParam}} r$,
- if $p \xrightarrow{\text{super}} q \xrightarrow{\text{queryParam}} r$ then $p \xrightarrow{\text{queryParam}} r$,
- if $p \xrightarrow{\text{super}} q \xrightarrow{\text{request}} r$ then $p \xrightarrow{\text{request}} r$, and
- if $p \xrightarrow{\text{super}} q \xrightarrow{\text{response}} r$ then $p \xrightarrow{\text{response}} r$.

Having defined inheritance for a resource, we extend this to requests. We cannot, however, just define a relationship $r \xrightarrow{\text{uriTemplate}} u$ when r is a request, since the same request may be shared among many resources. For example we could have specified that multiple resources can return an HTML representation:



In this case, the GET request has no single URI template, but instead must be given a context resource. We will write $p \vDash q \xrightarrow{\text{uriTemplate}} u$ whenever, in resource context p, request q has URI template u, and similarly for the other inherited properties. For all the other properties, the definition is quite simple:

- if $p \xrightarrow{\text{queryParam}} r$ or $q \xrightarrow{\text{queryParam}} r$ then $p \vDash q \xrightarrow{\text{queryParam}} r$,
- if $p \xrightarrow{\text{headerParam}} r$ or $q \xrightarrow{\text{headerParam}} r$ then $p \vDash q \xrightarrow{\text{headerParam}} r$,
- if $p \xrightarrow{\text{pathParam}} r$ or $q \xrightarrow{\text{pathParam}} r$ then $p \vDash q \xrightarrow{\text{pathParam}} r$, and
- if $p \xrightarrow{\text{response}} r$ or $q \xrightarrow{\text{response}} r$ then $p \vDash q \xrightarrow{\text{response}} r$.

[2]In that resolving the URI templates and then expanding should give the same result as expanding the templates and then resolving the resulting URIs.

For the derived URI template, the rule is slightly more complex, since there is special syntax for query parameters in the URI template RFC [8]):

- if $p \xrightarrow{\text{uriTemplate}} u$ and $\{k \mid p \vDash q \xrightarrow{\text{queryParam}} \cdot \xrightarrow{\text{name}} k\} = \{k_1, \ldots, k_n\}$ then $p \vDash q \xrightarrow{\text{uriTemplate}} u\{?k_1, \ldots ,k_n\}$.

For example, in the context of the Product resource class (with deduced query parameter named apikey and deduced URI template http://example.com/store/api/products/{id}), the GET method has the deduced URI template http://example.com/store/api/products/{id}{?apikey}.

C. Validity

Our primary motivation for including WIfL annotations in API documentation is to support validation of documents, and in particular to validate example dialogs.

A document is valid whenever, for every example x we can find a resource class r such that x is an instance of r (which we write $\vDash x : r$). In our definition of validity, we shall assume that each type t mentioned in the API description has a matching set of possible values $[\![t]\!]$ (for example if t is an XML schema type, then $[\![t]\!]$ is the set of all valid XML documents of that type). We will also make use of URI template expansion, and write $u[\sigma]$ for the URI given by expanding URI template u with data binding σ.

For any example x and resource class r, we define $\vDash x : r$ whenever:

- if $x \xrightarrow{\text{exampleRequest}} y$ and $x \xrightarrow{\text{exampleResponse}} z$ then there exists $r \xrightarrow{\text{request}} q$ and $r \vDash q \xrightarrow{\text{response}} p$ such that $r \vDash y : q$ and $\vDash z : p$.

For any example request x, resource class r and request class q, we define $r \vDash x : q$ whenever there exists a data binding σ such that:

- $r \vDash q \xrightarrow{\text{uriTemplate}} t$ and $x \xrightarrow{\text{uri}} u$ and $x \xrightarrow{\text{exampleHeader}} h$ and $\text{Host} \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$ and $t[\sigma] = \text{http://}v/u$,
- $q \xrightarrow{\text{method}} m$ and $x \xrightarrow{\text{method}} m$,
- if $r \vDash q \xrightarrow{\text{pathParam}} p$ or $r \vDash q \xrightarrow{\text{queryParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{type}} t$ and $(k, v) \in \sigma$ then $v \in [\![t]\!]$,
- if $r \vDash q \xrightarrow{\text{pathParam}} p$ or $r \vDash q \xrightarrow{\text{queryParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{fixed}} \text{true}$ and $(k, v) \in \sigma$ then $p \xrightarrow{\text{default}} v$,
- if $r \vDash q \xrightarrow{\text{pathParam}} p$ or $r \vDash q \xrightarrow{\text{queryParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{required}} \text{true}$ then $(k, v) \in \sigma$,
- if $r \vDash q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{type}} t$, and $x \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$, then $v \in [\![t]\!]$,
- if $r \vDash q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{fixed}} \text{true}$, and $x \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$, then $p \xrightarrow{\text{default}} v$,
- if $r \vDash q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{required}} \text{true}$, then $x \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$, and
- if $x \xrightarrow{\text{body}} v$ then $x \xrightarrow{\text{exampleHeader}} h$ and $\text{Content-Type} \xleftarrow{\text{name}} h \xrightarrow{\text{value}} c$ and $r \vDash q \xrightarrow{\text{representation}} p$ and $c \xleftarrow{\text{contentType}} p \xrightarrow{\text{type}} t$ and $v \in [\![t]\!]$.

For any example response x and response class q, we define $\vDash x : q$ whenever:

- $q \xrightarrow{\text{status}} s$ and $x \xrightarrow{\text{status}} s$,
- if $q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{type}} t$, and $y \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$, then $v \in [\![t]\!]$,
- if $r \vDash q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{fixed}} \text{true}$, and $x \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$, then $p \xrightarrow{\text{default}} v$,
- if $q \xrightarrow{\text{headerParam}} p$ and $k \xleftarrow{\text{name}} p \xrightarrow{\text{required}} \text{true}$, then $x \xrightarrow{\text{exampleHeader}} h$ and $k \xleftarrow{\text{name}} h \xrightarrow{\text{value}} v$.
- if $x \xrightarrow{\text{body}} v$ then $x \xrightarrow{\text{exampleHeader}} h$ and $\text{Content-Type} \xleftarrow{\text{name}} h \xrightarrow{\text{value}} c$ and $q \xrightarrow{\text{representation}} p$ and $c \xleftarrow{\text{contentType}} p \xrightarrow{\text{type}} t$ and $v \in [\![t]\!]$.

This completes the formal specification of WIfL.

IV. Implementation

A. URI Template Matching

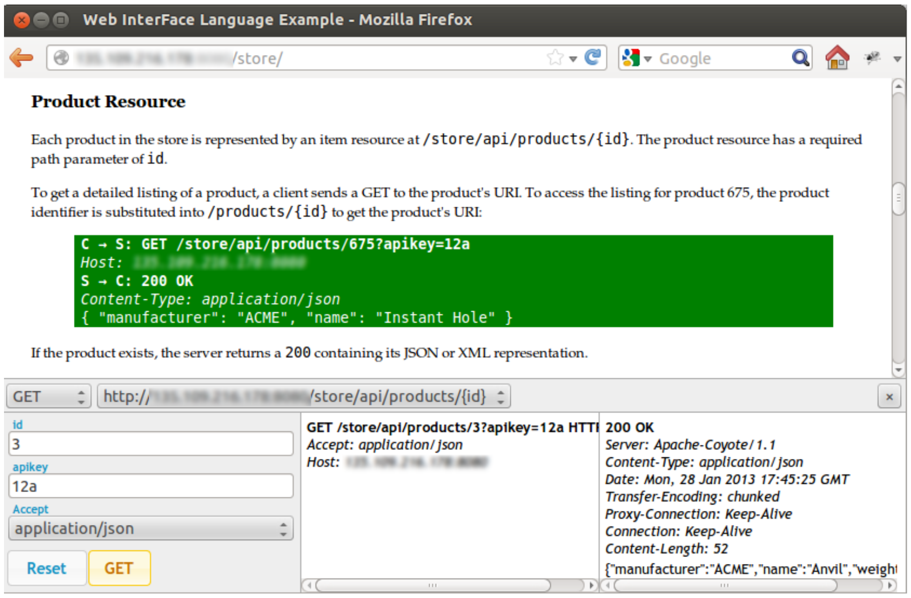
In validation, we are given a URI u, a URI template t, and we need to find a data binding σ such that $t[\sigma] = u$. That is, we need to *match* u against t.

We would also like to be able to perform *lookup* on an API (that is, given a URI, find the resource class it matches) and check for *overlap* (that is, are there distinct resources with a common URI they both match). Both of these problems can be solved using nondeterministic finite automata (NFAs) to represent URI templates.

Gregorio [12] (one of the authors of the URI Template RFC [8]) has said that matching was not considered to be a use case in the design of URI templates:

> URI Templates are for expansion and not parsing, so the use case of trying to figure out which value goes with which variable is not a supported use case.

In particular, there are many URI templates which match ambiguously, for example the URI template {a,b,c} can match the URI 1,2 in many different ways such as the data binding {a=1,b=2} or the data binding {a=1,c=2}. For this reason, we use nondeterministic automata (rather than deterministic ones) to implement URI templates.



Product Resource

Each product in the store is represented by an item resource at `/store/api/products/{id}`. The product resource has a required path parameter of `id`.

To get a detailed listing of a product, a client sends a GET to the product's URI. To access the listing for product 675, the product identifier is substituted into `/products/{id}` to get the product's URI:

```
C → S: GET /store/api/products/675?apikey=12a
Host: [redacted]
S → C: 200 OK
Content-Type: application/json
{ "manufacturer": "ACME", "name": "Instant Hole" }
```

If the product exists, the server returns a `200` containing its JSON or XML representation.

GET | http://[redacted]/store/api/products/{id}

id: 3
apikey: 12a
Accept: application/json

```
GET /store/api/products/3?apikey=12a HTTI
Accept: application/json
Host: [redacted]
```

```
200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Date: Mon, 28 Jan 2013 17:45:25 GMT
Transfer-Encoding: chunked
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Content-Length: 52
{"manufacturer":"ACME","name":"Anvil","weigh
```

Fig. 6. WIfL console

Moreover, URI templates use a copying semantics for repeated variables, for instance the URI template {a}/{a} will match the URI 1/1 but not the URI 1/2. Recognizing repeated strings is the canonical language which cannot be recognized by an NFA. For this reason, we do not implement the entire URI template language, but restrict ourselves to URI templates with no repeated variables.

We also make a technical restriction to only treating data bindings with values or arrays, not with associative arrays, for example our implementation will not match {?a*} against ?b=1.

The implementation of NFAs is standard, but (to track the variable assignments) we attach a partial function to each transition, and to each accepting state, which we term its *action*. Given an accepting run:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$$

where each q_i has a transition action f_i and q_n has an acceptor action g_n then the resulting value is:

$$f_1(f_2(\cdots f_n(g_n(\{\}), a_n) \cdots, a_2), a_1)$$

For example, the URI template {x} generates a one state automaton:

$$q_0 \xrightarrow{a} q_0$$

whose accepting action is the function:

$$\mathsf{function}(\sigma)\{\mathsf{if}(!\sigma.x)\{\sigma.x = [\,]; \mathsf{return}\,\sigma;\}\}$$

and whose transition action is the function:

$$\mathsf{function}(\sigma, a)\{\sigma.x.\mathsf{unshift}(a); \mathsf{return}\,\sigma;\}$$

Running this automaton on the string $a_1, \ldots, a_n$ generates the expected data binding $\{x : [a_1, \ldots, a_n]\}$.

We also associate each action with an inverse action, which allows us to implement URI template expansion on the same NFA as we built to perform URI template matching. Given a value σ we search for an accepting run:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$$

such that the result of:

$$g_n^{-1}(f_n^{-1}(\cdots f_2^{-1}(f_1^{-1}(\sigma, a_1), a_2) \cdots, a_n)) = \{\}$$

For example in the URI Template {x}, the inverse of the accepting function is:

$$\mathsf{function}(\sigma)\{\mathsf{if}(!\sigma.x == [\,])\{\mathsf{delete}\,\sigma.x; \mathsf{return}\,\sigma;\}\}$$

and the inverse of the transition action is the function:

$$\mathsf{function}(\sigma, a)\{\mathsf{if}(\sigma.x[0] == a)\{\sigma.x.\mathsf{shift}(); \mathsf{return}\,\sigma;\}$$

Running the inverted automaton on the data binding $\{x : [a_1, \ldots, a_n]\}$ generates the string $a_1, \ldots, a_n$ as expected.

Assuming that each f_i and g_i has an appropriate inverse, it is routine to check that expansion is the inverse of matching, and to implement both of these using backtracking recursive functions.

Using these NFAs, it is straightforward to implement URI template matching and expansion. We can also implement lookup, as follows:

- For each resource class r, build the NFA for its URI template t, then remove its transition actions and replace its accepting action with one that returns r. The resulting NFA will match URIs which match t, and will return value r.
- Take the set of all such NFAs and determinize them to produce a deterministic finite-state automaton (DFA). (This is possible because the NFAs do not have transition actions, otherwise we would have a problem since transducers cannot always be determinized [13]). The resulting DFA will match URIs which match any t_i, and will return value r_i.

Moreover, having built the DFA for the lookup function, it is is straightforward to check to see if any resource classes have overlapping URI templates: just check to see if any accepting state in the DFA has more than one return value.

There is complexity in this automaton-based approach to URI templates, but it has some advantages over straightforward recursive descent parsing:

- The matching and expansion functions are guaranteed to be inverses.
- Once the DFA has been constructed, the lookup function runs in linear time in the length of the URI.
- We can detect overlapping URI templates.

B. *Interactive Console*

Figure 6 shows a screenshot of the WIfL console. This allows a reader of an API document to quickly create HTTP requests to interact with the API's reference implementation. A reader can read an example dialog, click on it, and open up an console which they can use to interact with the API's reference implementation.

The console is implemented as a JavaScript library, using the jQuery [14] framework and the Green Turtle [15] RDFa processing engine.

- The console first extracts RDFa from the page, and follows the WIfL hyperlinks. It keeps extracting RDFa from the downloaded HTML until it has resolved all of the WIfL hyperlinks.
- It then converts the resolved RDFa into a JavaScript model of WIfL, and constructs NFA representations of the URI templates, as discussed in the previous section.
- When an example is selected, the URI is extracted from the example, and the lookup function is used to find the appropriate resource and request from the API description. The console form is built with the parameters given in the API description, and prepopulated with the parameter values given in the example dialog.
- When the form's action is selected, an appropriate XMLHttpRequest is created, and the request and response displayed in the console.

This console allows readers to interact quickly with an API's reference implementation, which would otherwise require a command-line use of curl or wget.

C. *Validator*

Figure 7 shows a screenshot of the WIfL validator.

The main audience for the validator is the team writing the API documentation, and it allows them to test their documentation for internal consistency. The screenshot shows that an invalid interaction has been highlighted: the request claims to have content type application/json, but the body is actually application/x-www-form-urlencoded.

As well as benefiting the technical writing team, readers of the API documentation benefit from the validator. Readers can use the validator in the interactive console, to check to see if their expectations about data validity are matched by the console. The screenshot shows an error message given by invalid user input: the XML payload does not match the XML schema.

The validator implements the specification given in Section III-C. It uses the same WIfL model as the console, and uses the JSON Schema library [16] and the libxml XML Schema library [17], ported to JavaScript [18] using the emscripten [19] LLVM bytecode interpreter. Since xmllint can take a few seconds to validate a document, it is run in a background thread using a web worker [20].

As well as running in a browser context, the validator can be run in headless mode inside node.js. This allows reports on validity to be generated during documentation builds.

V. Related Work

The most common languages for web API description are WADL [6] and WSDL [5]. These (especially WADL) have a similar API model to WIfL, but they are serialized as XML rather than being embedded in HTML. Other XML formats include REST Chart [21] and RIDDL [22]. Generating and maintaining a WADL or WSDL file as well as human-readable documentation can be difficult, especially for a technical writing team.

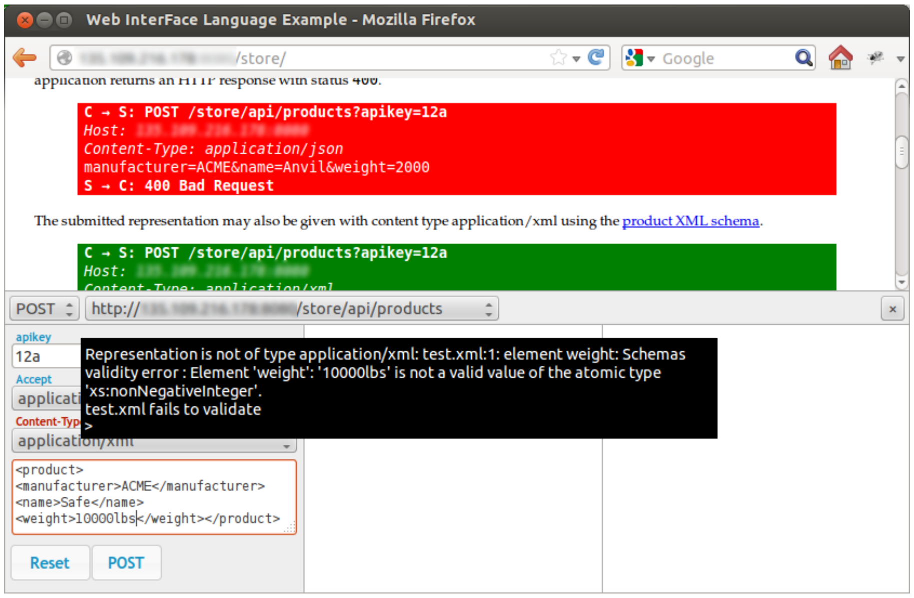
A previous attempt at embedding API descriptions in HTML was the poshformat (Plain Old Semantic HTML) hRESTs [23] (HTML for RESTful Services). Its model was very similar to the WSDL 1.1 model of SOAP-based web services, and adapted features of the RDFa-based SA-REST [24] (Semantic Annotations for REST). The hRESTs microformat requires the HTML document structure to match the logical structure of the API description, which is in conflict with our hyperlinking requirement. There is no console or validator for hRESTs.

The closest relative to our WIfL console is the Apigee API Console [25]. The main differences between our approaches is that Apigee uses WADL/XML rather than WIfl/RDFa, and that Apigee requires the API WADL to be uploaded to their servers, rather than embedding the console as JavaScript in any HTML page. Apigee does not support API example validation.

VI. Conclusions

Embedding machine-readable information in an API's HTML documentation expands its utility as a learning vehicle beyond passive reading. One addition is an interactive console, dynamically generated from the embedded description, which allows the human reader to experiment with the API from within the documentation. Another is a validation tool, which verifies that examples in the documentation really are exemplary of and conformant with the API.

Such tools provide benefits to several distinct groups of users: external developers (those wishing to learn an API for use in their applications), technical writers (those who document an API), and internal developers (those who implement the application that exposes an API). External developers



application returns an HTTP response with status 400.

```
C → S: POST /store/api/products?apikey=12a
Host: [redacted]
Content-Type: application/json
manufacturer=ACME&name=Anvil&weight=2000
S → C: 400 Bad Request
```

The submitted representation may also be given with content type application/xml using the product XML schema.

```
C → S: POST /store/api/products?apikey=12a
Host: [redacted]
Content-Type: application/xml
```

POST http://[redacted]/store/api/products

apikey: 12a

Accept: applicati

Content-Typ: application/xml

```
<product>
<manufacturer>ACME</manufacturer>
<name>Safe</name>
<weight>10000lbs</weight></product>
```

```
Representation is not of type application/xml: test.xml:1: element weight: Schemas
validity error : Element 'weight': '10000lbs' is not a valid value of the atomic type
'xs:nonNegativeInteger'.
test.xml fails to validate
>
```

Fig. 7. WIfL validator

use the console to instantly explore the API as they read its documentation. Technical writers validate their examples as they create them using a browser, well before a working implementation of the API exists, and they use the validation tool during the documentation build process for quality assurance. An internal developer, working in a code-first style with JAX-RS [26] or in a contract-first style with WADL, may dynamically generate an HTML document (e.g. using XSLT) with a console to test their implementation as they develop it.

Finally, the addition of API descriptions to HTML, which is already being visited by web crawlers, enables the discovery of APIs and their details that may not be otherwise available in the online API directories of today.

References

[1] A. Duvander, "8,000 APIs: Rise of the enterprise," 2012, http://blog.programmableweb.com/2012/11/26/8000-apis-rise-of-the-enterprise/.

[2] J. Musser, "Open APIs, what's hot, what's not," 2012, http://www.slideshare.net/jmusser/j-musser-apishotnotgluecon2012.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," 2000, http://www.ics.uci.edu/fielding/pubs/dissertation/rest_arch_style.htm.

[4] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating web APIs on the world wide web," in *Proc. European Conf. Web Services*, 2010, pp. 107 –114.

[5] "Web services description language (WSDL) version 2.0 part 1: Core language," W3C Recommendation, 2013, http://www.w3.org/TR/wsdl20/.

[6] M. Hadley, "Web application description language," W3C Member Submission, 2009, http://www.w3.org/Submission/wadl/.

[7] B. Adida, M. Birbeck, S. McCarron, and I. Herman, "RDFa core 1.1," W3C Recommendation, 2012, http://www.w3.org/TR/rdfa-core/.

[8] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard, "Uri template," IETF RFC 6570, 2012.

[9] D. C. Fallside and P. Walmsley, "XML schema part 0: Primer second edition," W3C Recommendation, 2004.

[10] F. Galiegue and K. Zyp, "JSON schema: core definitions and terminology," IETF Internet Draft, 2013, http://tools.ietf.org/html/draft-zyp-json-schema.

[11] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," IETF RFC 3986, 2005.

[12] J. Gregorio, "URI templates: comma-separated variable lists," W3C URI Mailing List, 2011, http://lists.w3.org/Archives/Public/uri/2011Aug/0024.html.

[13] M. Lothaire, *Applied Combinatorics on Words*. Cambridge University Press, 2005.

[14] J. Resig *et al.*, "jquery," http://jquery.com/.

[15] A. Milowski, "Green-turtle: An implementation of the RDFa 1.1 API for browsers," http://code.google.com/p/green-turtle/.

[16] K. Zyp *et al.*, "JSON Schema specifications, reference schemas, and a CommonJS implementation," https://github.com/kriszyp/json-schema.

[17] "The XML C parser and toolkit of Gnome," http://www.xmlsoft.org/.

[18] A. Zakai, "Port of libxml to JavaScript using Emscripten," https://github.com/kripken/xml.js.

[19] A. Zakai *et al.*, "Emscripten: An LLVM-to-JavaScript compiler," https://github.com/kripken/emscripten.

[20] I. Hickson, "Web workers," W3C Candidate Recommendation, 2012, http://www.w3.org/TR/workers/.

[21] L. Li and W. Chou, "Design and describe REST API without violating REST: A Petri net based approach," in *Proc. Int. Conf. Web Services*, 2011, pp. 508–515.

[22] J. Mangler, P. P. Beran, and E. Schikuta, "On the origin of services using RIDDL for description, evolution and composition of RESTful services," in *Proc. Int. Conf. Cluster, Cloud and Grid Computing*, 2010, pp. 505 –508.

[23] J. Kopecky, K. Gomadam, and T. Vitvar, "hRESTs: An HTML microformat for describing RESTful web services," in *Proc. Int. Conf. Web Intelligence and Intelligent Agent Technology*, 2008, pp. 619–625.

[24] A. P. Sheth, K. Gomadam, and J. Lathem, "SA-REST: Semantically interoperable and easier-to-use services and mashups," *IEEE Internet Computing*, vol. 11, no. 6, pp. 91–94, 2007.

[25] Apigee Corp., "What's an API console?" http://apigee.com/docs/enterprise/content/whats-api-console.

[26] "Java API for RESTful services (JAX-RS)," JSR 339, http://jax-rs-spec.java.net.