Extracting Models from Web API Documentation

Rolando Rodríguez[1], Roberto Espinosa[1], Devis Bianchini[2], Irene Garrigós[3], Jose-Norberto Mazón[3], and Jose Jacobo Zubcoff[4]

[1] WaKe Research, Dept. of Computer Science
University of Matanzas "Camilo Cienfuegos", Cuba
{rolando.rodriguez,roberto.espinosa}@umcc.cu
[2] Dept. of Information Engineering, University of Brescia, Italy
bianchin@ing.unibs.it
[3] WaKe Research, Dept. of Software and Computing Systems
University of Alicante, Spain
{igarrigos,jnmazon}@dlsi.ua.es
[4] WaKe Research, Dept. of Marine Sciences and Applied Biology
University of Alicante, Spain
jose.zubcoff@ua.es

Abstract. In order to develop web mashups, designers need an in-depth understanding of each Web API they are using. However, Web API documentation is rather heterogeneous, represented by big HTML files or collection of files in which it is difficult to identify elements such as API methods and how they can be invoked. Models have been widely recognized as first-citizen artifacts for documenting software applications, abstracting from implementation details, thus becoming good candidates to raise the level of automation of web mashup development. In this paper we present an approach for extracting models from Web API documentation. Our contributions are (i) a metamodel for standardizing the information extracted from Web APIs documentation; and (ii) a method for the extraction of models by parsing HTML files containing the Web API documentation, discovering useful data, and automatically generating the corresponding models (that conform to the defined metamodel).

1 Introduction

Web mashups are low-cost, personalized web applications, designed and implemented to be used for short periods of time and built starting from a set of predefined Web APIs. The great success of Web APIs basically relies on a very simple technological stack, based on HTTP, XML and JSON, and the extensive use of URIs. Nowadays, Web APIs are used to access and aggregate large quantity of data, like Flickr and Facebook, or to expose on the web contents from legacy systems. To promote the adoption of Web APIs for web mashup development, the ProgrammableWeb public repository[1] has been made available,

[1] http://www.programmableweb.com

where Web API providers share their own components and web designers can look for Web APIs they need to compose new web mashups without implementing them from scratch. Currently, the repository registers more than 6,000 Web APIs (a number that is continuously growing) and more than 6,600 user-defined mashups.

In this context, providing web designers with the required information to effectively find Web APIs they need and compose them in web mashups is becoming a more and more critical issue. Unfortunately, this task is hampered by the absence of a standard structure in Web API documentations (like the WSDL specification for SOAP-based Web services). On the one hand, Web API consumers are not constrained to adhere to any description language, on the other hand, information extracted from Web API documentation must be performed automatically, in a transparent way for consumers, starting from plain HTML documentation used to describe Web API usage.

For decades, models have provided developers with a standard and visual documentation for understanding software (e.g, UML in software engineering or ER model in databases). The same idea can be applied to Web APIs, where different kind of documentation formats may mislead designers. Having a common model can improve understanding and supporting them in using Web APIs. Also, if models are used, the level of automation of all the phases of Web mashup development is raised: semantic enrichment of Web API descriptions would be pursued to improve their selection [5] or to improve their automatic comparison for substitutability purposes (often referred to as *Web API migration* [2]); CASE tools may be implemented to ease web mashup composition and to enable automatic code generation [7].

Bearing these issues in mind, in this paper we introduce an approach for extracting models from Web API documentation. In particular, our contribution is twofold: (i) a metamodel for standardizing the data related to Web APIs documentation; and (ii) a method for extracting models (conformed to the Web API metamodel) by discovering useful data in the HTML files that contain the Web API documentation. Our aim is to perform a first step toward a computer-assisted extraction and semantic annotation of Web API models for web mashup composition purposes.

The paper is organized as follows: Section 2 provides an overview of the proposal and a motivating example. Our approach is detailed in Section 3, where the metamodel is described, and in Section 4, where the model extraction procedure from the Web API documentation is presented step by step with the help of the motivating example. A comparison with the state of the art to underline the cutting-edge elements of our proposal is discussed in Section 5. Finally, Section 6 closes the paper providing some hints about future work.

2 A Web API Model Extractor

Our approach aims at obtaining Web API models from a set of HTML files describing the Web API documentation. To do so, the first step concerns the analysis of a significant number of Web APIs with the aim of building a metamodel for them. Once we have the metamodel, a Web API model extractor is designed. This extractor is composed of two steps: (i) parsing the HTML files that contain the documentation in order to discover useful data, and (ii) creating a model (that conforms to the Web API metamodel) by using those data.

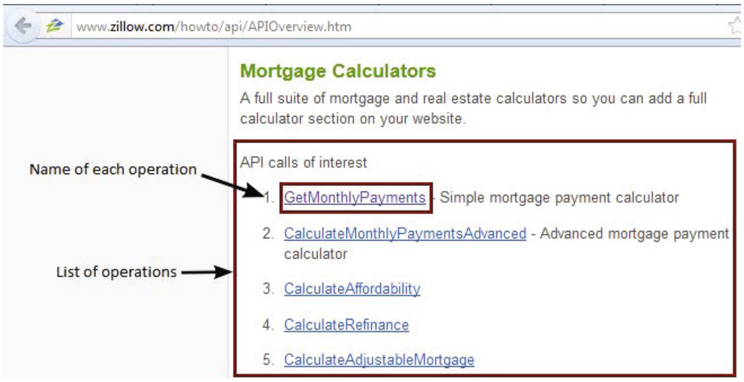
The problem addressed here is related to the one of designing proper wrappers to load contents of Web pages, such as Lixto[2], that has been developed for extracting product pricing from already known Web sources [3]. Nevertheless, Web API documentation is contained in rather heterogeneous (in format and content) and unfamiliar HTML files, thus hampering the task of discovering useful data, i.e. identify elements such as Web API methods (or operations), corresponding parameters to invoke them and which is the output provided by method invocations. Therefore, it is crucial to use some kind of "a priori" knowledge to identify the right portions of the HTML documentation which correspond to Web API elements. The documentation of a Web API is divided in two parts: a list of Web API operations, and a list of parameters of each Web API. After analyzing the structure of several Web API documentations (based on a representative population) on the ProgrammableWeb repository, we found that these two parts are represented in one or several web pages. We chose this repository since it is, to the best of our knowledge, the most complete collection of Web APIs shared among web mashup designers.

Our findings show that HTML documentation of any Web API presents recurrent patterns of tags (such as <`ul`>, <`table`> and so on) that can be used to discover useful data for our purpose. There are different tags used to highlight the name of the operations or parameters, e.g. <`h1`> or <`b`>, that can be considered as well. In order to know the most used tags for representing operations and their parameters, we conducted an analysis of a random sample of 30 Web APIs selected from the ProgrammableWeb site. This representative sample gives us enough information about tags used for enclosing operations and parameters. We use the ANalysis Of Variance [15] (ANOVA) technique, which is the most appropriate test with which to discover the most frequent tags. In our experiment, ANOVA is used to compare the means of usage of all tags by computing $p-values$, thus determining which are the most frequent tags from the analysis of the random sample. As a result, we can conclude that the <`ul`> and <`ol`> tags were the most used (ANOVA $p-value = 0.0196$) for enclosing lists of operations, while <`table`> was the main significantly different tag for enclosing parameters (ANOVA $p-value = 0.0136$). As a matter of fact, in this paper, we focus on these more used tags according to the results of our empirical analysis. Moreover, some recurrent terms contained in Web API documentations, such as *service*, *api*, *operation*, *inputs*, *outputs*, *method*, *parameter* and so on, could be exploited as additional knowledge to guide the discovery of useful data.

For the sake of understandability, a running example based on a Web API from ProgrammableWeb is presented throughout the paper. The running example is

[2] `http://www.lixto.com/`

based on the Zillow API[3], which provides real estate and mortgage data. Among the data about the Web API that can be extracted from ProgrammableWeb, there is the Zillow Web API documentation URL from which to obtain the list of operations of the API (as shown in Fig. 1(a)). In this figure, it is marked that each of the items representing an operation refers to a URL in which information of the parameters is defined. These parameters are described in a table as shown in Fig. 1(b). Throughout the paper, our approach is applied to Zillow Web API to show how a model is obtained from this documentation.



www.zillow.com/howto/api/APIOverview.htm

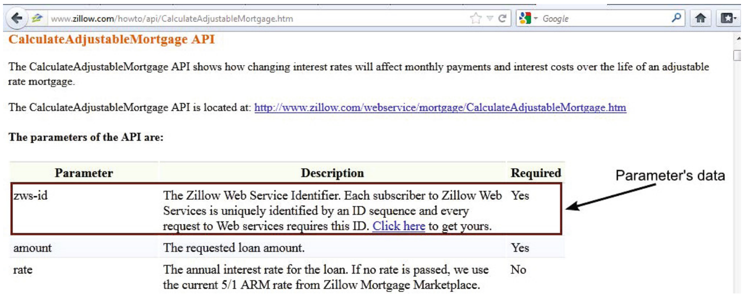
Mortgage Calculators

A full suite of mortgage and real estate calculators so you can add a full calculator section on your website.

API calls of interest

1. GetMonthlyPayments - Simple mortgage payment calculator
2. CalculateMonthlyPaymentsAdvanced - Advanced mortgage payment calculator
3. CalculateAffordability
4. CalculateRefinance
5. CalculateAdjustableMortgage

(a) Some of the operations of Zillow API.



www.zillow.com/howto/api/CalculateAdjustableMortgage.htm

CalculateAdjustableMortgage API

The CalculateAdjustableMortgage API shows how changing interest rates will affect monthly payments and interest costs over the life of an adjustable rate mortgage.

The CalculateAdjustableMortgage API is located at: http://www.zillow.com/webservice/mortgage/CalculateAdjustableMortgage.htm

The parameters of the API are:

Parameter	Description	Required
zws-id	The Zillow Web Service Identifier. Each subscriber to Zillow Web Services is uniquely identified by an ID sequence and every request to Web services requires this ID. Click here to get yours.	Yes
amount	The requested loan amount.	Yes
rate	The annual interest rate for the loan. If no rate is passed, we use the current 5/1 ARM rate from Zillow Mortgage Marketplace.	No

(b) Parameters of `CalculateAdjustableMortgage` from Zillow API.

Fig. 1. Sample screenshots of the Zillow Web API documentation

3 A Metamodel for Web APIs

Under the model-driven umbrella, and according to [11], "a model is a description of (part of) a system written in a well-defined language", while "a well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer". Therefore, on the one hand, a model must focus on those important parts of a system, thus avoiding

[3] `http://www.programmableweb.com/api/zillow`

superfluous details. On the other hand, well defined languages can be designed by means of metamodeling [4], which provides the foundation for creating models in a meaningful, precise and consistent manner.

Our Web API metamodel contains those useful concepts for representing models of a Web API in a standardized manner, thus dealing with the heterogeneity of Web API documentation. As aforementioned, models of Web APIs (based on our metamodel) can support web mashup designers during Web API retrieval and composition from huge repositories. Interestingly, the rough process of creating Web API documentation can be ameliorated by using models as well.

The definition of our metamodel (Fig. 2) is based on an analysis of two sources: (i) a significant number of Web APIs (taking into account a variety of formats and information contained in the documentation), and (ii) previous related work on modeling Web APIs [5]. In the following, we describe in detail the concepts included in our metamodel.

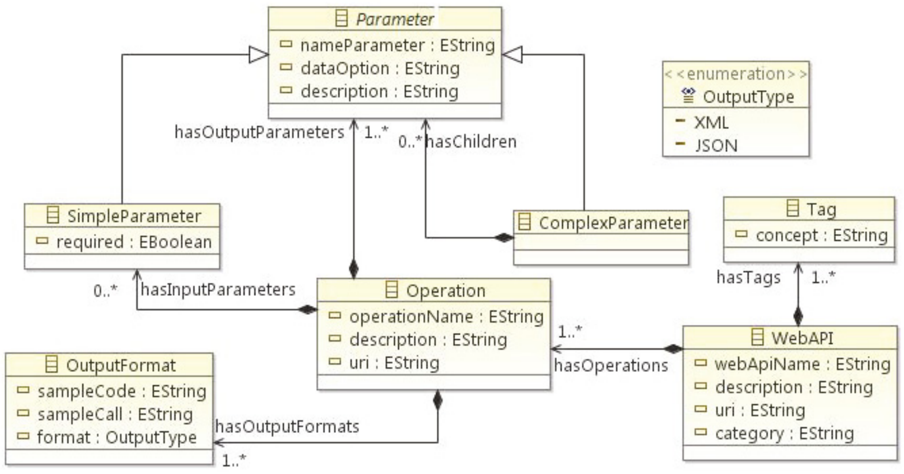


Fig. 2. A metamodel for Web APIs

WebAPI. This is the main container class for the other elements in the Web API model. It defines a name and a general description of the Web API. The `WebAPI` class contains a category which the Web API belongs to, a set of operations that can be called with the Web API and a URI from which the Web API documentation can be accessed. It contains a set of tags with which is semantically annotated.

Tag. This class defines a set of concepts with which the Web API can be semantically annotated.

Operation. It describes the methods implemented within the Web API. Each operation has a name and a description to explain the functionality of the operation. The class `Operation` can have zero or more input parameters, but it must always contain, at least, one output parameter. Since Web APIs may output results from method invocations according to several formats (e.g. XML or JSON), the metamodel contains the `OutputFormat` class associated to each operation. An output parameter of an operation may be simple or may contain a collection of parameters nested with their respective values. i.e., a complex parameter.

Parameter. It is an abstract class that represents either an input or an output parameter. This class contains the name of the parameter, a value type for that parameter, and a description. The `Parameter` class is the base class for defining simple and complex parameters.

SimpleParameter. This class inherits from the `Parameter` class and defines a simple input or output parameter for an operation. In the context of Web APIs, *simple* means that the parameter is in the form `attribute=value`, where value is a simple type. This class contains a boolean attribute to indicate if the parameter is required or not. The `SimpleParameter` class can be used as an input parameter or as part of an output parameter.

ComplexParameter. It inherits from the `Parameter` class and represents a complex parameter type. This class can contain other parameters (by means of the `hasChildren` association) that give it the *complex* nature, as being related to other parameters. As shown in Fig. 2, the `ComplexParameter` class can be only used as an output parameter.

OutputFormat. This class represents the information related to the different formats that can be returned in an output of each Web API. It stores the name of the format (by means of an enumeration called `OutputType`) and sample excerpts of source code.

While required data for creating `WebAPI` and `Tag` classes can be acquired from the ProgrammableWeb site, data for creating the remainder of the classes are found in the specific Web API documentation (see Fig. 1).

Our Web API metamodel has been implemented by using the Eclipse Modeling framework (EMF)[4]. The EMF project is a modeling framework and code generation facility for building applications based on a structured model and metamodels.

4 Model Extraction from Web API Documentation

Our approach for extracting models from Web API documentation has two main stages: (i) parsing HTML pages containing the documentation of the Web API to discover required data (i.e., generic Web API data, operation data and parameters data), and (ii) using these data for generating a model of the Web API (conformed to the metamodel). An overview of our approach is shown in Fig. 3.

[4] `http://www.eclipse.org/emf`

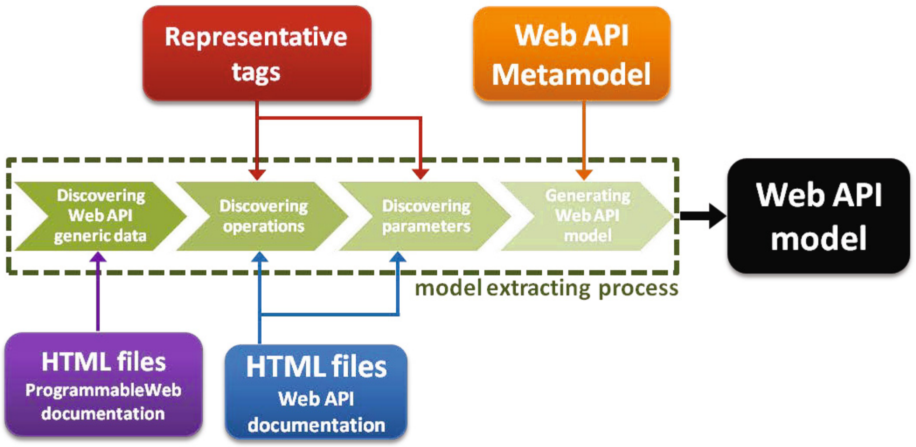


Fig. 3. Overview of our Web API model extractor

4.1 Discovering Data from Web API Documentation

The first step when collecting data is invoking methods from the ProgrammableWeb site to extract useful information about Web APIs[5]. In particular, they enable:

– to retrieve Web APIs by category or tags; given a category c, the `api.programmableweb.com/apis/-/`c method is invoked and the list of Web APIs that have been categorized in c is returned; within the ProgrammableWeb repository Web APIs are classified in 67 categories such as `mapping`, `payment`, `search`; tag-based Web API retrieval is performed in the same way;
– to retrieve the details of a given Web API; given a Web API $\mathcal{W}$, the `api.programmableweb.com/apis/`$\mathcal{W}$ method is invoked to retrieve all the detailed information about the Web API $\mathcal{W}$.

Once data from ProgrammableWeb site has been acquired, the URI of the Web API documentation is used for obtaining the remainder of data. A set of steps and heuristics for parsing Web API documentation have been defined and implemented in a well-known tool for defining ETL (Extract/Transform/Load) processes, named Pentaho Data Integration (aka Kettle)[6].

According to our preliminary statistical study, operations are mainly represented as <`ol`> or <`ul`> tags. Enclosed by these tags each operation is contained in a <`li`> tag with some decoration to visually highlight the name of the operations, e.g., underlined (<`u`>), bold faces (<`b`>), header type (<`h1`>) and so on[7]. In order to discover these data, the steps performed are as follows:

[5] `http://api.programmableweb.com/`
[6] `http://kettle.pentaho.com/`
[7] There are alternatives, e.g. combinations of <`span`> and <`div`> tags intentionally marked with `id` or `class` that are then defined in style sheets. Considering these tags is part of our future work.

1. cleaning the HTML page in order to get a well-formed XML document suitable for further processing; specifically, HtmlCleaner[8], an open-source HTML parser written in Java, is used;
2. extracting every piece of HTML code between tags that structure the required data; in this case, the focus is on the `<ul>`, `<ol>` and `<li>` tags;
3. applying several heuristics in order to ameliorate the detection of operation names, namely:
   - discard those pieces of HTML code in which the word *operation* or a synonym such as *method* or *call* are not presented in the previous piece of code; a thesaurus has been properly created to consider these terms;
   - text enclosed by style tags such as `<h1>` or `<b>` is likely to represent operations, since they are normally highlighted due to its importance for documentation purposes;
   - text enclosed by `<a>` tags is likely to represent operations, since it refers to the URL of the documentation of the operation; for example, `<li><h4><a href=''operations/getusers.html''>GetUsers</a>...`, where `GetUsers` is the name of the operation, and the URL to get the documentation of that operation is `operations/getusers.html`.

In our running example, the Web page of the Zillow API documentation is `http://www.zillow.com/howto/api/APIOverview.htm`. A sample excerpt is acquired and shown as follows:

```
<div class="api-overview">
 <h4>Home Valuation</h4>
 <p>Search results list , Zestimate<sup>&reg;</sup>, Rent Zestimate<sup>&reg;</sup>,
    home valuations , home valuation charts , comparable houses , and market trend charts.</p>
 <p class="no-margin">API calls of interest:</p>
 <ul>
  <li><a href="/howto/api/GetZestimate.htm">GetZestimate</a></li>
  <li><a href="/howto/api/GetSearchResults.htm">GetSearchResults</a></li>
   ...
</ul> </div>
```

Our ETL process provides the functionality required for detecting that this piece of HTML code contains the required data, parsing these data and extracting the name of each operation (enclosed in each `<li>` tag) together with the corresponding URL (combination of the current URL and the *href* attribute of the `<a>` tag). For example, `GetZestimate` is an operation and the URL that provides information about it is `http://www.zillow.com/howto/api/GetZestimate.htm`..

Next step is using each of the retrieved URL to navigate through documentation in order to acquire the information related to each operation. Note that the URL can be the same in which the operations are listed (i.e., operations and parameters can be in the same website). The steps to perform are as follows:

1. recovering each operation website;
2. cleaning the HTML page in order to get a well-formed XML document as aforementioned;

[8] `http://htmlcleaner.sourceforge.net/`

3. focusing on pieces of code enclosed by <`table`> tags;
4. discarding those pieces of HTML code in which the words *parameter* or *response* or some synonyms are not presented in the previous piece of code (also for this purpose, we rely on the thesaurus of potentially related terms);
5. extracting data from tables; the header of the table indicates the name of the concept (parameter, description, etc.) and other rows indicate values; data from these tables can be extracted from <`tr`> and <`td`> or <`th`> tags.

Recalling our running example, a sample excerpt of code for `GetZestimate` operation is as follows:

```
<h4>The parameters of the API are:</h4>
<table class="improvements" summary="parameters␣of␣the␣GetZestimate␣API">
 <thead>
  <tr>
   <th>Parameter</th><th>Description</th><th>Required</th>
  </tr>
 </thead>
 <tbody>
  <tr>
    <td>zws-id</td>
    <td>The Zillow Web Service Identifier. Each
    subscriber to Zillow Web Services is uniquely
    identified by an ID sequence and every request
    to Web services requires this ID.</td>
    <td>Yes</td>
  </tr>
   ...
 </tbody>
</table>
```

Parameters are structured in tables in which the first row indicates that the first cell is the `Parameter`, the second one is the `Description` and, finally, the third one indicates if the parameter is `Required` or not. In the code excerpt of our running example above, `zws-id is` considered as a required parameter, being `The Zillow Web Service Identifier...` its description.

4.2 Creating a Web API Model

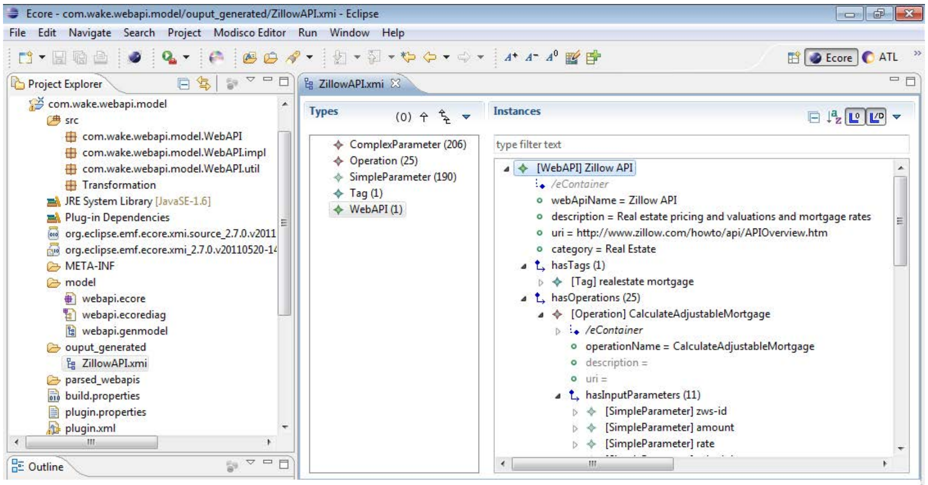
Once we have acquired data required from the Web API documentation, the corresponding model is created. Our metamodel has been included in an EMF plugin that contains all the new functionality required to generate Web API models, since EMF provides facilities for dynamically creating models that conform to a metamodel. To this aim, from the metamodel, several libraries can be derived:

com.wake.model.webapi.WebAPI. It contains general code for interfaces and factories to create the Java class to allow web designers to create elements of a model dynamically.

com.wake.model.webapi.WebAPI.impl. It contains specific code for generating Java classes tailored to our metamodel.

com.wake.model.webapi.WebAPI.util. It contains the `AdapterFactory` that provides facilities for creating classes via `create()` methods and giving them values via *getter* and *setter* methods.

Fig. 4 shows the Web API model extracted from our running example. The `WebAPI` class *Zillow API* and all the data related to it, including operations and parameters, are generated.


Ecore - com.wake.webapi.model/ouput_generated/ZillowAPI.xmi - Eclipse
File Edit Navigate Search Project Modisco Editor Run Window Help
Ecore
ATL
Project Explorer
ZillowAPI.xmi
Types
Instances
type filter text
Outline

```
com.wake.webapi.model
  src
    com.wake.webapi.model.WebAPI
    com.wake.webapi.model.WebAPI.impl
    com.wake.webapi.model.WebAPI.util
    Transformation
  JRE System Library [JavaSE-1.6]
  Plug-in Dependencies
  org.eclipse.emf.ecore.xmi.source_2.7.0.v2011
  org.eclipse.emf.ecore.xmi_2.7.0.v20110520-14
  META-INF
  model
    webapi.ecore
    webapi.ecorediag
    webapi.genmodel
  ouput_generated
    ZillowAPI.xmi
  parsed_webapis
  build.properties
  plugin.properties
  plugin.xml
```

```
ComplexParameter (206)
Operation (25)
SimpleParameter (190)
Tag (1)
WebAPI (1)
```

```
[WebAPI] Zillow API
  /eContainer
  webApiName = Zillow API
  description = Real estate pricing and valuations and mortgage rates
  uri = http://www.zillow.com/howto/api/APIOverview.htm
  category = Real Estate
  hasTags (1)
    [Tag] realestate mortgage
  hasOperations (25)
    [Operation] CalculateAdjustableMortgage
      /eContainer
      operationName = CalculateAdjustableMortgage
      description =
      uri =
      hasInputParameters (11)
        [SimpleParameter] zws-id
        [SimpleParameter] amount
        [SimpleParameter] rate
```

Fig. 4. Sample model of Zillow Web API

5 Related Work

We compared our approach with other related efforts, distinguishing among approaches which impose their own model to which API developers must adhere [1,8], approaches which automate the extraction of models from API documentation [6,9,14], approaches which provide a slight support for structuring the API documentation [12,13].

Some solutions such as WADL (Web Application Description Language) [8] have been developed for Web APIs to be the counterpart of the WSDL standard. In [1] a formal model based on Datalog rules is proposed to capture all the aspects of a mashup component or Web API (called there *mashlet*). Unfortunately, these proposals are too complex to be adopted in a Web 2.0 context, where Web APIs providers do not want to be hampered by the adoption of new, high-level, abstract languages or formalisms and prefer to use plain HTML documentation (on which they are more skilled).

The application of Model Driven Engineering techniques to extract models from Web APIs code has been described in [9], that is focused on object-oriented API specification. The tool described in [9] is designed to obtain model from API source code, while our goal is to start from plain, unstructured HTML documentation. Also approaches like MoDisco [6] and SM@RT [14] only work when the source code is available. Wazaabi[9] extracts GUI models from SWT, JFS and Swing, working only with these kinds of APIs.

The problem of supporting the structuring of Web APIs starting from HTML documentation has been addressed in tools like SWEET [13], which enables the use of the hRESTS formalism to identify the Web API elements (methods, inputs, outputs and so on) within the HTML documentation, with the goal of

[9] http://wazaabi.org

assisting their semantic annotation. Within the SWEET tool, all these tasks are mainly manually performed. This is basically the approach taken by other tools, like LOOMP [12] or the one described in [10]. Nevertheless, the statistical analysis described in our paper showed that HTML documentation could be very complex, including many additional contents out of the relevant ones for Web API description, and a support to identify the latter ones is crucial. Our approach provides a semi-automated method to support Web API model extraction.

6 Conclusions

Extracting models from Web API documentation is an interesting task to be done with the aim of providing a standard and visual representation of Web APIs. These models can be used for helping designers in finding and combining the required Web APIs within a specific mashup. With this aim, our approach is based on (i) a metamodel for standardizing the information extracted from Web APIs documentation; and (ii) a method for the extraction of models by parsing HTML files containing the Web API documentation, discovering useful data, and automatically generating the corresponding models (that conform to the defined metamodel).

As a short-term future work, several experiments will be conducted in order to validate our approach. Our experiments will consist of using our approach for obtaining models for each Web API documentation in the ProgrammableWeb site and manually comparing several measures (e.g., quantity of operations and parameters correctly retrieved) to study precision and recall. As a long-term future work, our plan is using our model-driven approach for guiding users in the process of discovery, semantic annotation and composition of Web APIs. As a matter of fact, our next step in this sense will be the definition of measures and techniques for increasing the level of automation in the semantic annotation of the Web APIs.

Acknowledgements. This work has been partially funded by IN.MIND project from University of Alicante (Spain), and MANTRA project (GV/2011/035) from Valencia Government (Spain).

References

1. Abiteboul, S., Greenshpan, O., Milo, T.: Modeling the Mashup Space. In: Proc. of the Workshop on Web Information and Data Management, pp. 87–94 (2008)
2. Bartolomei, T.T., Czarnecki, K., Lämmel, R., van der Storm, T.: Study of an API Migration for Two XML APIs. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 42–61. Springer, Heidelberg (2010)
3. Baumgartner, R., Gottlob, G., Herzog, M.: Scalable web data extraction for online market intelligence. PVLDB 2(2), 1512–1523 (2009)
4. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)

5. Bianchini, D., Antonellis, V.D., Melchiori, M.: Semantics-Enabled Web API Organization and Recommendation. In: Proc. of International Workshop on Web Information Systems Modeling, WISM 2011, Brussels, Belgium, pp. 34–43 (2011)
6. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE 2010), pp. 173–174 (2010)
7. Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., Francalanci, C.: DashMash: A Mashup Environment for End User Development. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 152–166. Springer, Heidelberg (2011)
8. Hadley, M.: Web application description language. Tech. rep., W3C (2009)
9. Izquierdo, J.C., Jouault, F., Cabot, J., Molina, J.G.: API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering. Information and Software Technology 54, 257–273 (2012)
10. Kiryakov, A., Popov, B., Terziev, I., Manov, D., Ognyanoff, D.: Semantic annotation, indexing, and retrieval. Journal on Web Semantics 2, 49–79 (2004)
11. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Practice and Promise of The Model Driven Architecture. Addison Wesley (2003)
12. Luczak, M., Heese, R.: Linked Data Authoring for non-Experts. In: Proceedings of the Workshop on Linked Data on the Web (2009)
13. Maleshkova, M., Pedrinaci, C., Domingue, J.: Semantic annotation of Web APIs with SWEET. In: Proc. of the 6th Workshop on Scripting and Development for the Semantic Web (2010)
14. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating Synchronization Engines between Running Systems and Their Model-Based Views. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 140–154. Springer, Heidelberg (2010)
15. Winer, B., Brown, D., Michels, K.: Statistical Principles in Experimental Design. McGraw-Hill (1991)