Ajax API Self-adaptive Framework for End-to-end User

Xiang Li[1, 2], Zhiyong Feng[1, 2], Keman Huang[1, 2], Shizhan Chen[*, 1, 2]
[1]Tianjin Key Laboratory of Cognitive Computing and Application
[2]School of Computer Science and Technology, Tianjin University
{xiguozi, zyfeng, keman.huang, shizhan}@tju.edu.cn

ABSTRACT
Web developers often use Ajax API to build the rich Internet application (*RIA*). Due to the uncertainty of the environment, automatically switching among different Ajax APIs with similar functionality is important to guarantee the end-to-end performance. However, it is challenging and time-consuming because it needs to manually modify codes based on the API documentation. In this paper, we propose a framework to address the self-adaption and difficulty in invoking Ajax API. The Ajax API wrapping model, consisting of the specific and abstract components, is proposed to automatically construct the grammatical and functional semantic relations between Ajax APIs. Then switching module is introduced to support the automatic switching among different Ajax APIs, according to the user preference and QoS of Ajax APIs. Taking the map APIs, i.e. Google Map, Baidu Map, Gaode Map, 51 Map and Tencent Map as an example, the demo shows that the framework can facilitate the construction of RIA and improve adaptability of the application. The process of selection and switching in the different Ajax APIs is automatic and transparent to the users.

Categories and Subject Descriptors
D.1 [Software]: Programming Techniques — Automatic Programming; D.2.3 [Software]: Software Engineering—Coding Tools and Techniques.

Keywords
Ajax API, End-user Development, Self-adaption, Dynamic Components Composition

1. INTRODUCTION
In Web 2.0, because of the asynchronous nature and high interactivity, Ajax [1] is extremely popular for the Web application developing. Moreover, with the trend of Open API, more and more Ajax application providers open their Ajax APIs based on their application to third parties. However, it is challenging and time-consuming for the Ajax API discovery, invoking and runtime switching. Taking the usage of the popular map application as a typical scenario, when a user wants to embed a map into his application, he must find an available map API over the Internet. And then he has to study the API documentation. Also, he must be a JavaScript and Ajax literate. Above of these result in a steep learning curve.

* is corresponding author

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742834.

Furthermore, once the user completes his map-based application, however the API cannot be invoked or it is updated, he has to repeat previous complex work again to guarantee his application's availability. Overall, the difficulty of finding an available API, time-consuming process of studying API document, high learning threshold for user, and complexity of API adaption makes it challenging to construct high-quality application based on Ajax API.

Recently, some researches on mashup [4] and dynamic web service selection [5], aim at wrapping and organizing of SOAP or RESTful APIs, and designing mechanisms for invocation and composition. However, because invoking pattern is different between Ajax API and SOAP or REST API, these mechanisms cannot be directly applied on Ajax APIs. In [2], the authors propose a method that can discovery and recommend APIs conveniently according to the syntax relations of classes and methods. But it is not helpful to invoke and switch APIs. In [6], [3], researchers utilize abstract interfaces to complete self-adaptive switching in syntax layer. Due to the lack of semantic information of API, they build abstract interfaces by manual programming. And when different APIs are switched by those frameworks, these frameworks cannot solve the problem of parameters adaptions.

Our paper constructs a self-adaptive framework for the automatic switching among different Ajax APIs to guarantee the RIA's availability in the long-time running. First, the wrapping module is introduced to construct the Ajax API's semantic-aware model based on its documentation, which consists of the *specific component (SC) and the abstract component (AC)*. Then the switching module is constructed to support the automatically replacement of the disable Ajax API, including the alternative candidate selection and the parameter adaption. Finally, the demonstration on the map API application shows that our framework can effectively support the self-adaption between different APIs and facilitate the usage of the APIs.

The rest of the paper is organized as follows: Section 2 introduces our self-adaptive framework and section 3 shows the details of the implementation. Section 4 presents the demonstration of our framework. Section 5 concludes our work.

2. SELF-ADAPTIVE ARCHITECTURE
Figure 1 gives the overview architecture of our ***S***elf-***A***daptive ***A***jax ***A***PI ***F***ramework (*SAAAF*). The SAAAF locates between the original Ajax APIs and the end users. In order to facilitate its description, we introduces four different denotations as follows:

a) ***Abstract application***: is consisted of abstract components in specific order. It is a mapping to requirement of end users.

b) ***Concrete application***: is executive code of Ajax application that is built by more suitable Ajax API.

c) ***Specific component (SC)***: is an encapsulation for a function and some objects from an Ajax API. The specific component is an executive unit and has a specific semantics.

d) ***Abstract component (AC)***: is semantic abstract of specific components and represents semantics of the specific components.

As shown in the Figure 1, when the end user needs to invoke one Ajax API to create an RIA, he/she will send their request to the framework. Then the *wrapping Ajax API module* is used to wrap the SCs and ACs to construct the *Ajax application resource library* based on the API's documentation. The SCs represent the full-fledged callable components with specific functional semantics. They wrap functions and objects from the original Ajax API according to their grammatical relations. The ACs represent the functional semantics in SCs, and an AC corresponds to one or more SCs. The *Ajax API switching module* will construct the *concrete application* for the end-user. During the long-time running, if the used API becomes invalid due to the dynamic environment, the switching module could switch the Ajax API to the one with the same function and make the switching process transparent for the end-users.

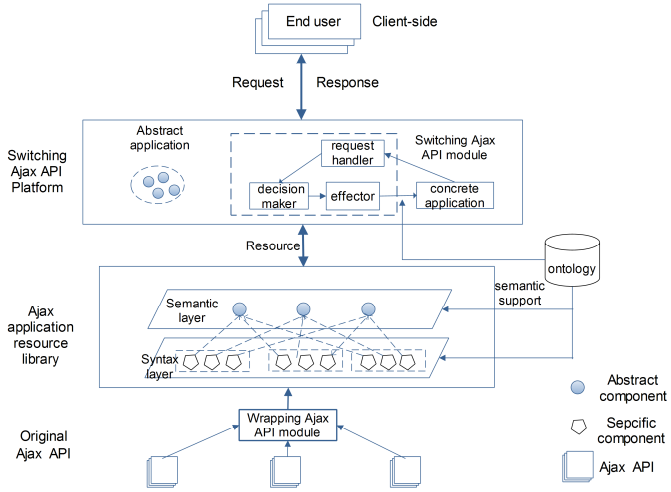

Fig. 1*: Architecture of the Self-Adaptive Ajax API framework*

2.1 Wrapping Ajax API module

Based on the documentations of the Ajax APIs, we can extract the name, parameters, return of the function as well as the invoking relation between objects and functions. Therefore, the SC can be formally defined as follows:

$$SC = <object\_set, function, parameter\_set, code> \quad (1)$$

where *object_set* is a set of objects which is indispensable to invoke the function, *function* is one specific functionality in the API document, *parameter_set* is a set of parameters for the *function* and *object_set*, *code* is executive code of JavaScript.

Obviously, an original Ajax API can be wrapped into many SCs. Furthermore, for each SC, we can add the semantic annotation based on DBpedia according to the methodology shown in [7]. Then the SCs with similar functional semantics will be organized into an AC. Due to the space limitation, the detail for the wrapping is left out. Table 1 shows the wrapping example of SC and AC.

Table 1. Example of the specific and abstract component

Specific Component	Abstract Component
{ Id:2, APIid:1, Object_set:[{name:"map",class:"BMap .Map"},{name:"navigation",class:"BM ap.NavigationControl"}], Function:"addControl", Parameters:"", Code:"map.addControl(….)..", Ab_Component_ID:2}	{Id:2 Functional semantics:"Navigation Control Map" }

2.2 Switching Ajax API module

The switching Ajax API module consists of three sub-module, including *request handler*, *decision maker* and *effector*. It will support the selection and the composition of SCs to fulfill the user's request, as well as the automatic switching between different Ajax APIs.

SCs' Selection and Composition: Firstly, the *request handler* accepts a user's request, and then gets the information of *abstract application* based on URL and sends this information to *decision maker* and *effector*. The *decision m*aker selects one API according to the given information and sends the decision to *effector*. The *effector* gets indispensable SCs according to both ACs of *abstract application* and selected Ajax API. Then, the *effector* assembles the executive codes from SCs to produce initial *concrete application*.

Algorithm 1: Semantics-based Parameters Replacement.

Input: *LPSC* is a list of parameters of selected SCs;

SPVAA is a set of parameters values for an *abstract application;*

CCA is codes of the initial *concrete application;*

Output: *C'* is codes with values of parameters

1. V_set = *SPVAA*;
2. P_list=*LPSC*;
3. C=*CCA*
4. **foreach**(p ∈P_list) *//p is one parameter in the P_list*
5. **foreach**(v∈V_set) *//v is one parameter value in the V_set*
6. **if**(p.semantic_annotation==v.semantc_annotation)
7. replace p.name with v.value in the C
8. **end foreach**
9. **end foreach**
10. **if**(∃p is not replaced && ∃v is not used)
11. create SPARQL with p.sem_anno and v.sem_anno
12. send SPARQL to the DBpedia to query the values
13. replace p in C with values from DBpedia
14. Return C'

Automatic Switching: If the Ajax API becomes invalid due to the update or policy change, the *wrapping Ajax API module* will select another SC with the same functionality to replace the disable one. Here we simply select the one with the same functional semantics and the highest QoS. Other methodologies about how to select the optimized replacement [8] can be easily integrated into our framework. As we already added semantic annotations to parameters of every SC, the most important task left is to map existing inputs into the selected SC's parameters.

Algorithm 1 shows the detail of our semantics-based parameters replacement. Line 01~03 initialize variables, line 04~09 fill in parameters based semantic annotations, line 11-13 get necessary values from DBpedia by SPARQL, line 14 generate the integrated *concrete application*.

3. IMPLEMENTATION

Our Ajax API self-adaptive framework is implemented based on Node.js (http://nodejs.org/) and Apache Wink RESTful web Service (http://wink.apache.org/). MongoDB (http://www.mongodb.org/) is used as the database of framework for the Ajax application resource library.

The *wrapping Ajax API module* stores SCs and ACs to Ajax application resource library. The *wrapping Ajax API module* consists of an API's document crawler, an API's document analytic program, SCs generator and semantic annotation adder. The SCs generator stores SCs into the MongoDB. The semantic annotation adder utilizes DBpedia spotlight (https://github.com/dbpedia-spotlight/dbpedia-spotlight).

For the switching module, as shown in Figure 2, the *effector* is divided into the *data collector* and the *code generator*; the *request handler* and *decision maker* is implemented by RESTful Web Services:

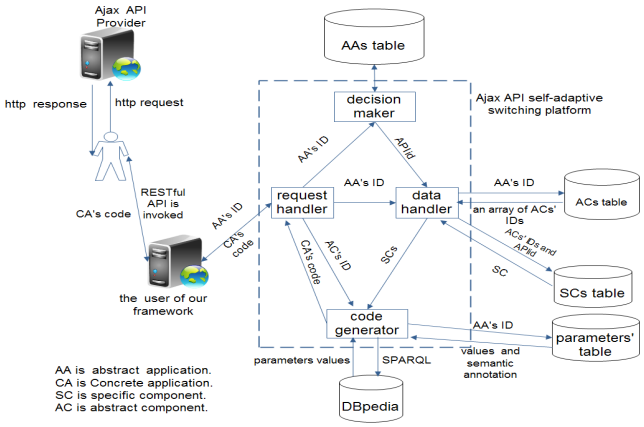

Fig. 2*: Architecture of switching Ajax API platform*

a) **Request handler**

Request handler gets the ID of *abstract application* and sends it to decision maker and data handler.

b) **Decision maker**

Decision maker gets the ID of *abstract application*, and then queries which APIs can be used and selects one Ajax API according to the rank of their QoS.

c) **Data handler**

Data handler requests the ID of selected Ajax API to *decision maker*. And it queries MongoDB to get codes and parameters of specific components.

d) **Code generator**

Code generator combines each codes of SC by ACs order to generate an initial *concrete application*. It fills in parameters in the initial *concrete application*. Therein, code generator utilizes the DBpedia SPARQL endpoint (http://dbpedia.org/sparql) to query necessary parameters' values.

4. DEMONSTRATION

4.1 Ajax API Document Wrapping

Figure 3 shows the user interface of our *wrapping Ajax API module*. It can crawl the specific API document over Internet based on the given URL. Also it supports users to submit the API documents directly. According to the method discussed in Section 2, when the document is available, the module will analyze and extract information of functions and objects from the document, acquire the syntax relations between functions and objects, generate executive codes and store SCs into the database, and finally add semantic annotations to SCs and organize them into ACs.

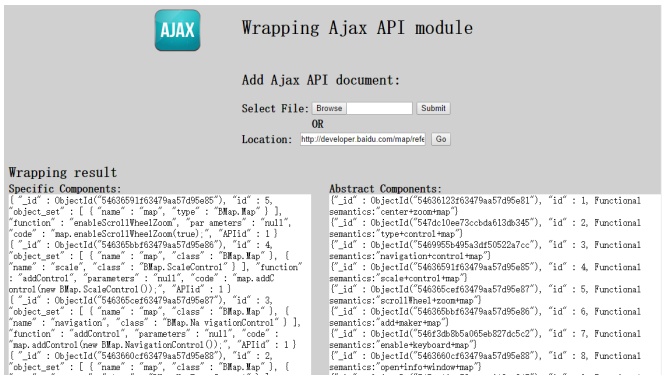

Fig. 3*: User interface of the Wrapping Ajax API Module. The top center part allows users to submit the Ajax API document or input its URL. The bottom left is the specific components extracted from the document and the bottom right is the abstract components generated based on the specific components.*

4. 2 Ajax API Switching

In Figure 4, taking the map applications as an example, we use Tencent Map to create a map-based application which is the basic map with navigation control, maptype control and scale control. We can visit through "http://127.0.0.1/map?Aappid=1" to get the application. When errors occur in the Tencent Map API, our framework detects the failure and replaces Tencent Map API with 51 Map API，which has the highest QoS among another four map APIs. Note that this switching process is transparent to the end-user, which means that the end-user can still visit the same URL to fetch the application and get the same user-experience while the map has been changed into 51 Map API instead.

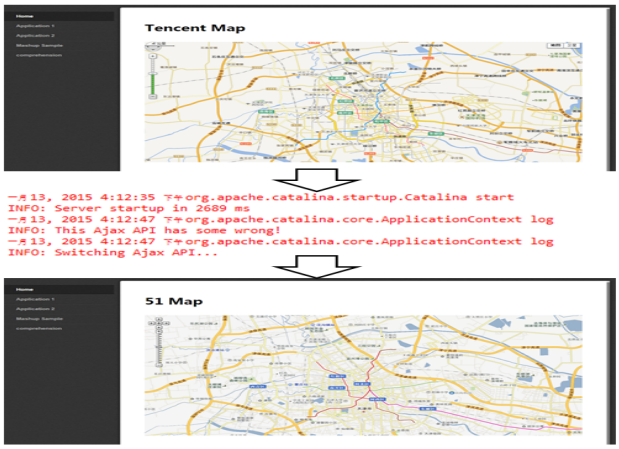

Fig. 4: *Effect of our demo in a map application. Notes that when our framework detects the failure of current used API, it can switch to the available one transparently.*

4. 3 Using Ajax API in the Mashup

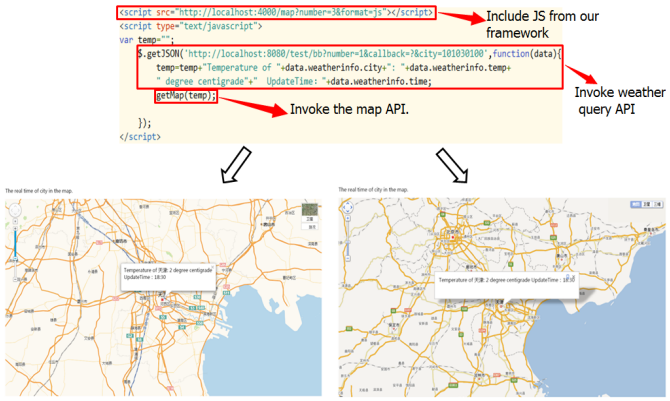

Fig. 5: *Creating a Mashup. Notes that use our framework can invoke Ajax API conveniently in the mashup and return the best Ajax API automatically every time.*

We create a simple mashup that can get city weather and show it in the map. The mashup is consisting of a weather query API (http://www.weather.com.cn/data/sk/101030100.html) and a map API which is given by our framework. To invoke the map API in the mashup, as shown in Figure 5, we only need two steps, as follows:

a) Include a JS in the mashup as follows.

"<script src="http://127.0.0.1/map?number=3&format=js">"

b) Add a invoking function such as "getApp(temp)". The "temp" is acquired from weather query API.

Obviously, if the initial map API becomes invalid during the long-time running, it will automatically transfer to another available one so that the mashup can keep available and the developer doesn't need to manually replace the unavailable API at present.

The screencast of our demonstration is available at *https://www.youtube.com/watch?v=vSl3s76Nj8Q&feature=youtu.be*.

5. Conclusion

Our Ajax API self-adaptive framework is a tool for building a mashup, which makes the automatically switching between Ajax API candidates transparent to the end-user. A wrapping module is introduced to automatically analyze the API documentation, which can wrap and store Ajax API codes in syntax layer as specific components, as well as enrich semantics information and organize the specific components into abstract components. The switching module is presented to select and compose the specific components into mashup to fulfill the user's request, as well as to automatically and transparently replace the unavailable API with available one. The demonstration shows that our framework can effectively facilitate the understanding of the API document, the automatic switching between APIs as well as the usage in the mashup.

In the future, we continue to extend the mechanism for encapsulating, and adapting Ajax APIs. We will also design optimal methods to capture user requirements.

6. ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China grant 61373035, 61173155, the Tianjin Research Program of Application Foundation and Advanced Technology grant 14JCYBJC15600, and the National High-Tech Research and Development Program of China grant 2013AA013204.

7. REFERENCES

[1] Garrett, J.J. Ajax: A New Approach to Web Applications, 2005,http://www.adaptivepath.com/publications/essays/archives/000385.php.

[2] W. Chan, et al., Searching connected API subgraph via text phrases, Proc. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, pp. 1-11.

[3] S. Lingyan, et al., A QoS-based Self-adaptive Framework for OpenAPI, Proc. 2011 Seventh International Conference on Computational Intelligence and Security (CIS), 2011, pp. 204-208.

[4] L. Xuanzhe, et al., Composing Data-Driven Service Mashups with Tag-Based Semantic Annotations, Proc. 2011 IEEE International Conference on Web Services (ICWS), 2011, pp. 243-250.

[5] S. Nakajima, An Architecture of Dynamically Adaptive PHP-based Web Applications, Proc. 18th Asia Pacific Software Engineering Conference (APSEC) , 2011, pp. 203-210.

[6] E. Khanfir, et al., A Web Service Selection Framework Based on User's Context and QoS, Proc. 2014 IEEE International Conference on Web Services (ICWS), 2014, pp. 708-711.

[7] Z. Zhen, et al., Semantic Annotation for Web Services Based on DBpedia, Proc. IEEE 7th International Symposium on Service Oriented System Engineering (SOSE), 2013, pp. 280-285.

[8] H. Keman, et al., Recommendation in an Evolving Service Ecosystem Based on Network Prediction, Automation Science and Engineering, IEEE Transactions on, vol. 11, no. 3, 2014, pp. 906-920.