API Description Generator for Python Apps on Google App Engine

Leah Chatkeonopadol
Department of Computer Science
University of California, Santa Barbara
chatkeon@cs.ucsb.edu

Smruthi Manjunath
Department of Computer Science
University of California, Santa Barbara
smanjunath@cs.ucsb.edu

Abstract—**As web applications become more popular, there is an increasingly large number of APIs available for web developers to use. Correspondingly, there is an increasing need for API descriptions so that programmers can easily understand and interface with existing APIs. The generation of API descriptions is currently a manual, time-consuming, tedious process. We propose to solve this problem with our Python API Description Generator. Our service uses static and dynamic parsing of a web app's source code to identify and extract the information that would be useful for other developers to have in order to build upon the aforementioned app. Then it automatically generates the API description from this data and displays it on demand. Furthermore, we provide validation of the API description according to certain guidelines, and if errors are found in the structure of the API, appropriate error messages will be output.**

We designed and implemented an API Description Generator to run on Google App Engine Python apps. It runs in the background so that it will not affect the behavior of the original app. In addition to quickly and conveniently generating API descriptions, our tool also assists in the testing phase of the development process because its design requires the developers to execute every logical path in the app and thus thoroughly test it.

General Terms
Design, web applications, APIs

Keywords
Google App Engine, API description, webapp2, Python 2.7, automatic code generation

1. Introduction

Web applications are becoming increasingly popular due to their ease of development and usage. There are a number of resources available to help programmers, and web apps can be widely deployed since they only require a browser rather than any specific architecture. In addition, many APIs for web apps already exist and can be integrated into newly-written code. However, in order to properly use these services, developers must thoroughly understand the APIs and their interfaces. This highlights the need for some sort of API description to prevent developers from having to read the code and its documentation, should it exist.

For instance, consider this scenario: you have written an API that will be useful for many different applications, and you want to make it easy for other developers to reuse your code. In order to accomplish this, you need to provide an API description that other programmers can read to quickly understand how your API works. This implies you now need to sit down and manually write a description of your app, a process that is both time-consuming and tedious.

Or imagine you are developing a web app and wish to make use of an existing API. You want to know what resources and operations the API contains, but to learn this, you will need to go through the source code and any available documentation. If an API description existed, it would save you time and effort in analyzing the API and enabling you to use it. Unfortunately, few APIs include descriptions, and there is currently no way to obtain an API description automatically.

To address this need, we have developed an API description generator. This tool is automated so that programmers will not have to go through the trouble of manually generating descriptions for each new API. Our primary goal was to generate overviews that describe the basic details and operations of given APIs. The description generator was implemented in Python and is designed

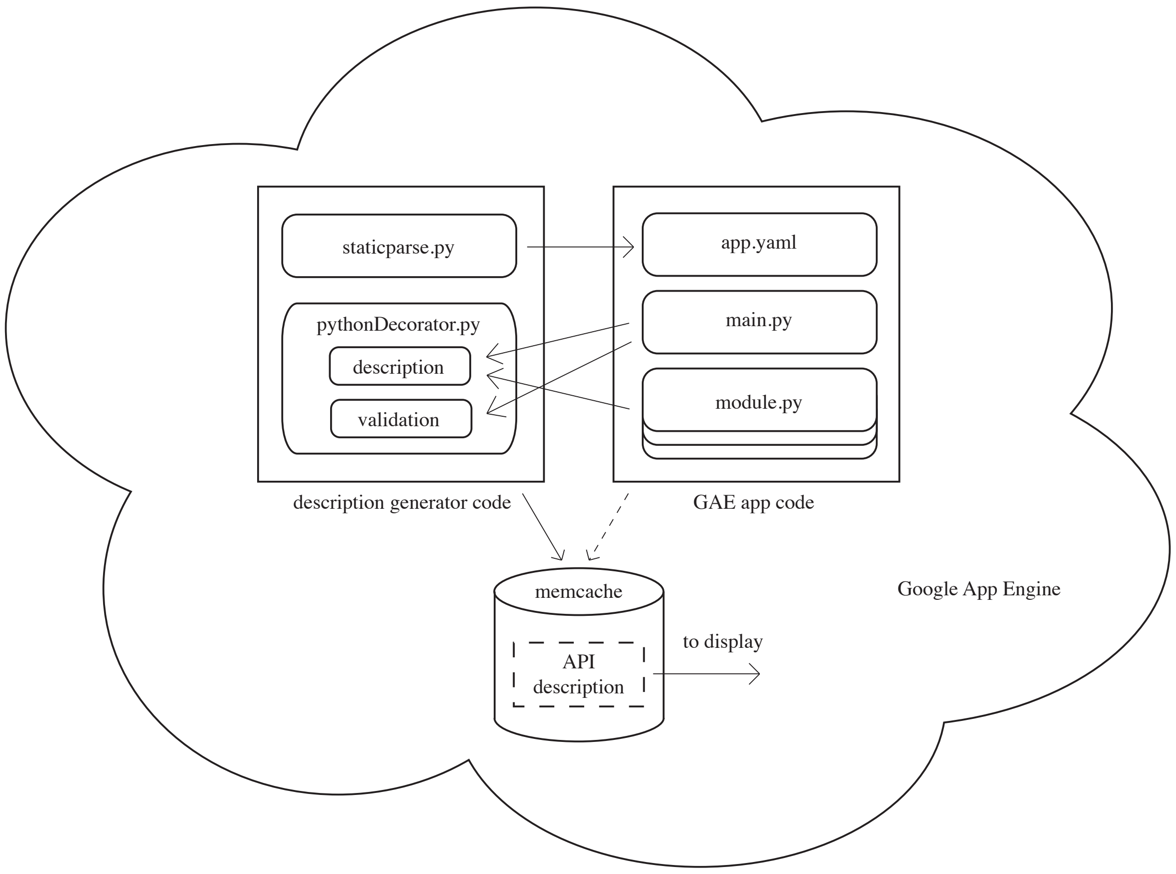


Fig. 1: Service Architecture

for Python apps, specifically those written for Google App Engine (GAE).

The rest of this paper is outlined as follows: Section 2 describes some related work in this area. Section 3 details the challenges we faced during development. Sections 4 and 5 present our design approach and implementation, while Section 6 covers the usage of our description generator. Sections 7 and 8 mention the limitations of our service and possible future work. Finally, section 9 contains our conclusions.

2. Related Work

The main work that has been done in this area is a similar API description generator for Java web apps, implemented by the RACE lab [1]. Their tool, which employs static parsing and the use of JAX-RS annotations, formed the basis of our work. This is also work in progress to enable the automatic generation of code to implement APIs from given API descriptions [2]. This project is actually the inverse of ours.

API documentation generators, such as javadoc [3] and pydoc [4], also exist. They perform a function similar to that of our service, but they produce documentation rather than an API description. While documentation contains some of the same information as an API description, it serves a different purpose, so it does not conveniently provide the information needed for a developer to quickly understand and reuse an API.

3. Challenges

The main challenge we faced in approaching this project was the fact that Python is a dynamically-typed language. As such, information about data types—and consequently, information about variables, methods, and classes—cannot be obtained from static parsing. To address this, we chose to use a combination of static and dynamic parsing. Static parsing was used on the configuration file to get basic information about the application, and dynamic parsing was used on the source code via Python decorators. We also required that developers who use our tool properly document their code according to a prespecified format.

Another challenge we encountered is the lack of standards for API descriptions. Because API descriptions are not commonly used yet, there are no formal definitions for a valid API description or what it should include. Current terminology is unclear and it is difficult to anticipate what information developers might need. In response to this problem, we adopted a working set of guidelines based on [5].

4. Design

In order to simplify the project, we designed our service for Google App Engine apps.

4.1 Google App Engine

Google App Engine (GAE) is a Platform-as-a-Service that allows people to host web apps using Google's servers. The GAE servers automatically handle scaling and load balancing so that developers do not have to worry about the management or deployment of applications. Furthermore, GAE provides a simple SDK, including a local development server, as well as extensive support for development, deployment, and monitoring of apps. All of these features, which are free within a certain limit, make GAE an excellent choice for web app developers. Therefore, we anticipate that GAE will be widely-used and will continue to grow, so we have designed our description generator specifically for GAE apps. At the moment, GAE supports web apps written in Java, Python, and Go. We have chosen to focus on Python.

GAE apps follow a pre-defined format, so we can make certain design assumptions. First of all, GAE apps use the webapp2 framework, a lightweight Python web app framework. Second, while GAE currently supports Python 2.5 and Python 2.7, support for Python 2.5 will be removed soon, so we will assume web apps using our service are written in Python 2.7. Finally, GAE requires a configuration file titled app.yaml to be included with each web app. We have taken the above parameters into consideration in designing our project.

4.2 Service Architecture

The architecture of our service is illustrated above in Figure 1. There are three main parts to our service. First is the web app for which the API description will be generated. This will include a configuration file, app.yaml, and some source code files. There may also be other optional configuration files, but our tool would not access these, so they are not pictured. The next component of our service is the API description generator code. This code comprises of two modules: staticparse.py, which performs static parsing on the app.yaml file, and pythonDecorator.py, which performs dynamic parsing on the app's source code. The latter also validates and generates the API description. This description is then stored in the third part of the figure, the database memcache provided by GAE. When invoked, the validation function will also display the API description (assuming it is valid). Because our description generator runs on top of already complete web apps, the entire service is hosted on GAE's servers.

We designed our project in this way to overcome the challenge of Python's dynamic typing. Our dynamic parsing employs Python decorators to deal with runtime objects, while our static parsing takes advantage of the static information in the configuration file. Furthermore, the way we have designed the validation and display of the API description ensures that any errors in our service or incompatibility between our code and the developers' code will not affect the behavior of the app. Our description generator has no visible effect on the original app until the API description is requested by the user, so it is robust against failure of service. The exact implementation details are given below.

5. Implementation

5.1 Static Parsing

GAE specifies the following basic format for the beginning of an app's configuration file [6]:

```
application: app_name
version: 1
runtime: python27
api_version: 1
```

Listing 1: app.yaml

We call our staticparse module once at the start of API description generation to obtain the application's name, version, compatibility (i.e., version of Python), and identifier (API version). If desired, the developer can also add a one-line comment describing the app between the first and second lines, and we will include it in the API description. In addition, we derive the base URL of the web app from the name of the app, assuming the default GAE domain name $[app\_name].appspot.com$ is used. There is other information given in app.yaml, such as the libraries that the app uses, but we are not currently using this data in our service.

The information gained from static parsing forms the beginning of our API description. It will provide developers who wish to use this API with important environment details.

5.2 Dynamic Parsing

Dynamic parsing in our service is done entirely through the use of decorators, which are a unique Python feature. They are similar to macros and allow the modification of code during runtime. Decorators are identified by a '@' symbol and may be applied to both functions and classes by inserting @decorator_name before the definitions. They act like ordinary functions except that they take function or class objects as inputs and return the same, possibly modified [7]. The decorators are executed as soon as the decorated objects are invoked and run before the code of the method or class in question. A simple decorator function might be defined and used as follows:

```
def decorator_name(self):
  print "Decorating", self.__name__
  # modify the object if desired
  print "Exiting decorator"
  return(self)

@decorator_name
def function1(x,y):
  print "Function has been decorated"
  print x, y

Output:
>> function1(3,4)
>> Decorating function1
>> Exiting decorator
>> Function has been decorated
>> 3 4
```

Listing 2: Python decorator example

Decorators can be stacked. If there are multiple decorators, the innermost decorator will be executed first. For example [8]:

```
@called_last
@called_first
def function2():
  return 1
```

Listing 3: Multiple Python decorators

In our service, we expect every HTTP method and each method-less class to be decorated with our decorator, *description*. By method-less class, we mean those classes that have only attributes and no methods; we consider these to be user-defined data types. Our decorator is invoked each time a decorated method or class is executed. When this happens, the method or class object is passed to our decorator code. From this, we extract the name, class (for a method), and docstring of the object.

To ensure we add elements only once, we maintain a list of already-processed objects. If the current object is not in this list, then we add it to the list and process it before including it in the API description as a resource, operation, or data type. Resources are defined as those classes with HTTP methods, operations are the HTTP methods themselves, and data types, as mentioned above, as method-less classes.

Since Python is dynamically typed, we ask the developers to properly document their code in the docstrings of their methods and classes. After obtaining these docstrings from the objects passed to our decorator, we parse them for the information specified by the developer. Therefore, we have imposed restrictions on the format of the docstring; we require that docstrings for method-less classes be as shown below:

```
"""
Class description line1
[Class description line2]

Attributes:
  name: type: [description]
  name: type: [description]

"""
```

Listing 4: Format of method-less class docstring

Similarly, docstrings for HTTP methods should conform to this format:

```
"""
Description line1
[Description line2]

Args:
  name: type: [description]
  name: type: [description]

Bindings:
  name: type: description: binding type
  name: type: description: binding type

Returns:
  name: type: [description]

Exceptions:
  errorcode1: [cause]
  errorcode2: [cause]

"""
```

Listing 4: Format of HTTP method docstring

Everything in square brackets is optional. If no description is given, we will insert "Unspecified" in that field of the API description. As for the other parts of the docstring:

- Only one or two arguments are expected, where one will be the input of the HTTP method and the other will be a query, should it exist.
- Two types of bindings are possible, url and header; these are input parameters that can be accepted by the operations.
- Only one return type is expected; this should be the output type of the HTTP method if it exists.
- Any HTTP exceptions that could be raised should be specified.
- The category labels (Args, Bindings, Returns, and Exceptions) are mandatory, even if the category is not applicable.
- Bindings and exceptions can be left blank if desired.

In order to generate the API description, we need all of our Python decorators to be called. This can only happen if all of the decorated code is invoked. Therefore, we also require that the developers click through the entire application—i.e., execute every logical path—before they can view the API description. As each decorator is called, the description is generated in the background and stored in memcache. We specify a time-out period of one hour to ensure the API description is always up-to-date. If a longer expiration time is required, it can easily be changed in the decorator code.

5.3 Validation

Before displaying the API description to the user, we check to determine if the description is valid. As mentioned in Section 3, there are no formal standards for the validity of API descriptions, so we have adopted guidelines from [5]. Hence, we consider an API description to be valid if it satisfies the following conditions:

- API has a name (has a name attribute)
- API has at least one base URL (has a base attribute pointing to a non-empty array)
- API has at least one resource (has a resources attribute pointing to a non-empty array)
- Each resource has at least one operation (each resource element has an operations attribute pointing to a non-empty array)
- Each operation has a HTTP method (each operation element has a method attribute)
- There are no references to undefined types
- There are no references to undefined input bindings

[5] also notes that these conditions do allow for some information, such as descriptions, errors, and headers, to be left out of an API specification. The non-functional fields (e.g., license, community, tags) can also be excluded.

We guarantee the first two requirements by taking the name and base URL information of the app from app.yaml. Since the app will not run without a correct configuration file, this is sufficient to fulfill these conditions. The next three requirements are dealt with by checking the internal lists we maintain that store resources, operations, and methods. For example, we verify that the operations list for each given resource is not empty; this satisfies the fourth condition above.

We check the data types by keeping a list of valid data types. This list by default contains the valid primitive data types in Python (e.g., bool, int, string, function, lambda, None, etc.), and we add user-defined data types as they are encountered. Each data type in the API description is then validated against the list before the description is displayed. Finally, we verify input bindings by only accepting url and header bindings (queries are considered to be input parameters rather than input bindings).

If any of the conditions specified above are not satisfied, or if any of the expectations in Section 5.2 are not met, then the following errors will be detected and displayed instead of the API description:

1) No resources found
2) Resource with no operations
3) Incorrect number of arguments
4) Invalid number of return types
5) Undefined data type
6) Information unspecified/unknown

If no errors have been found, the complete and valid API description will be displayed to the user. The entire API description is stored in memcache as a dictionary of strings. To output the description, we simply use a JSON encoding for formatting and write the result to the browser. An example is below in Figure 2.

To date, we have tested our service by locating and downloading webapp2 GAE applications, modifying the docstrings manually to conform to our specifications, and adding our tool to the original source code. Done correctly, this resulted in correct API descriptions for these test apps. We also determined that our code is capable of generating descriptions for webapp apps (the predecessor to webapp2), but required slight modifications to the app's source code.

6. Usage

In order to use our API description generator, developers will need to modify the source code of their app to include our Python decorators. Assuming they have placed our files, staticparse.py and pythonDecorator.py, in the same directory as their code, they will need to import the pythonDecorator module into all .py files that contain HTTP methods or method-less classes. This module will call staticparse.py, so there is no need to include that module. Next, the developers should decorate all HTTP methods and method-less classes with *@pythonDecorator.description*, as shown below.

```
class MainPage(webapp2.RequestHandler):
  @pythonDecorator.description
  def [HTTPmethod](self):
    """
    [properly formatted docstring]

    """
    # code
@pythonDecorator.description
class [DataType]():
  """
  [properly formatted docstring]

  """
  x = 1
  y = 2
  z = 3
```

Listing 5: Example of decorated code

If their method or class is already decorated, they must ensure that our decorator is the innermost one. They should also add docstrings to these methods and classes according to the specifications given above. Then, developers need to include the following code in their main .py file:

```
class APIDes(webapp2.RequestHandler):
  def get(self):
    pythonDecorator.validate(self)
```

Listing 6: Code to be inserted

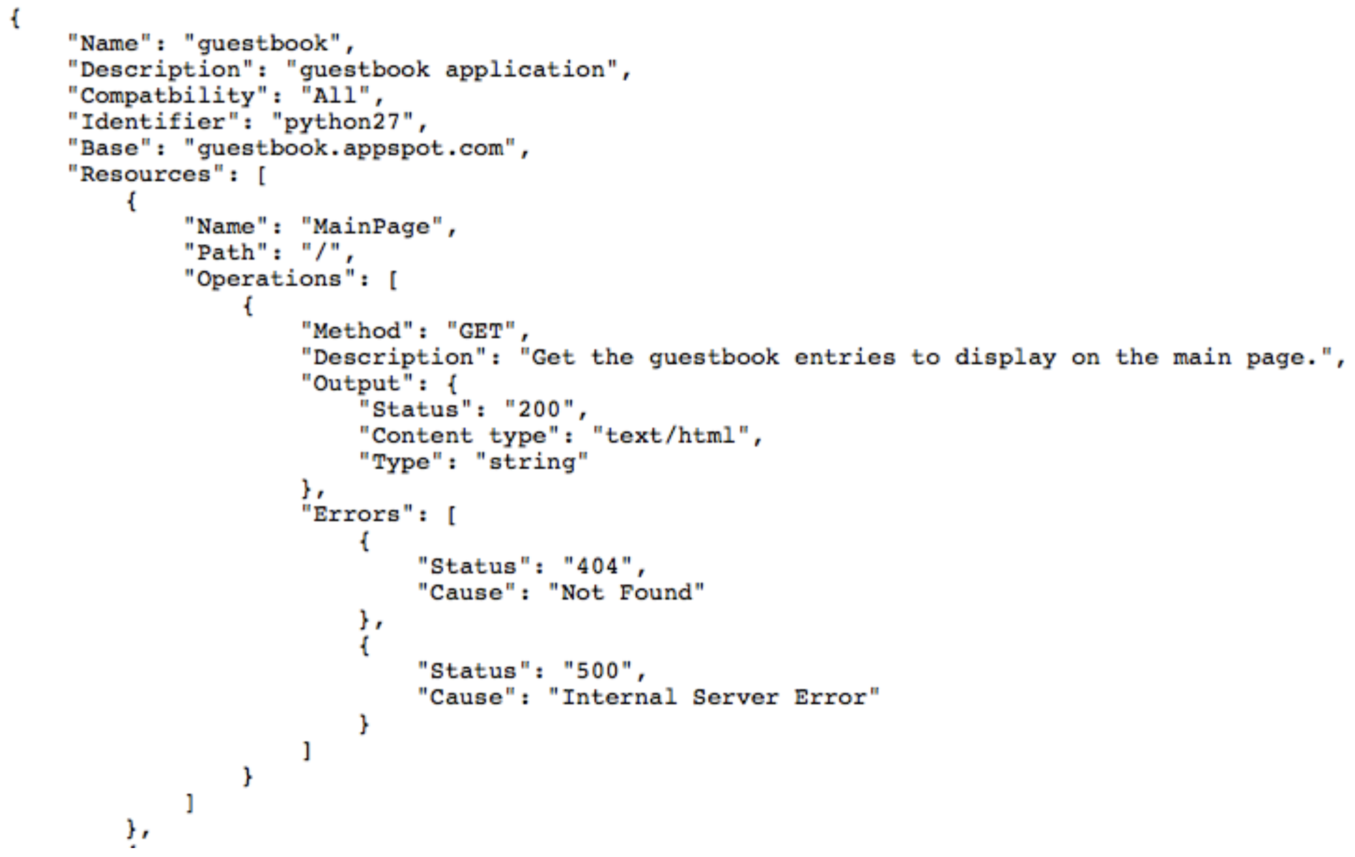
Although this class does contain a HTTP method, it should not be decorated. As the name suggests, this class is invoked when the API description is desired. It will call the *validate* function in the pythonDecorator module, which will validate the stored API description and display either the description or the error(s) found, if they exist.

Finally, developers need to insert an extra parameter, ('/options', APIDes), into the list given in their *webapp2.WSGIApplication()* function. This function is required by the webapp2 framework. It specifies which class (and therefore, which request handler) should handle each URL path. For example [6]:

```
app = webapp2.WSGIApplication([
  ('/', MainPage),
  ('/sign', Guestbook),
  ('/options', APIDes)
  ], debug=True)
```

Listing 7: Inserting '/options'

Doing this will enable users to view the API description by appending '/options' to the base URL of the app.



```
{
    "Name": "guestbook",
    "Description": "guestbook application",
    "Compatbility": "All",
    "Identifier": "python27",
    "Base": "guestbook.appspot.com",
    "Resources": [
        {
            "Name": "MainPage",
            "Path": "/",
            "Operations": [
                {
                    "Method": "GET",
                    "Description": "Get the guestbook entries to display on the main page.",
                    "Output": {
                        "Status": "200",
                        "Content type": "text/html",
                        "Type": "string"
                    },
                    "Errors": [
                        {
                            "Status": "404",
                            "Cause": "Not Found"
                        },
                        {
                            "Status": "500",
                            "Cause": "Internal Server Error"
                        }
                    ]
                }
            ]
        },
```

Fig. 2: API Description

7. Limitations

The primary limitation of our current implementation is the requirement that developers invoke all the decorated methods and classes in their app. On the one hand, this limitation forces developers to ensure that their app has been thoroughly tested, since this is a vital part of the development process; it guarantees that all components of the app function properly. In addition, this design choice allows developers to exclude experimental portions of their app from the API description by simply not decorating the relevant code. We also ensure that the description is always up-to-date, since changes in the source code necessitate the generation of a new description. However, at the same time, the latter is a limitation because the API description resets at each edit. Even minor changes to the app will require the developers to click through the entire app again to generate the most recent version of the description. (It is possible to work around this limitation, though, by simply copying the desired API description and storing it locally in a file so it can be used in the future without the overhead of needing to traverse the entire API.)

Another limitation is the stringent requirements on the format of the docstring. Small formatting errors may cause our service to fail, although the running application will continue to function (the app will only be affected when the API description is requested). Nonetheless, we feel that our specifications will actually assist developers in producing complete documentation by offering a template for them to follow. In fact, it might even set a standard for writing Python docstrings.

Finally, some of the fields we do not provide in our API description, such as license and security, may be useful for other developers to have. We have not included these because they are nonessential features, but we anticipate that they can be easily added.

8. Future Work

In the future, we propose to include an automated testing component in our service to assist the developers by running the entire application without any manual input. This component would analyze the source code of the app for logical execution paths and then follow each of these, thus ensuring that all decorated code is invoked. Combined with the validation function, this would enable the almost instantaneous output of the API description without any added overhead on the developers' part.

We would also like to extend our description generator to be compatible with Python web apps developed for platforms other than GAE. Additionally, we hope to expand our API descriptions to contain fields with nonessential but still relevant information.

9. Conclusion

In conclusion, we have designed an automatic tool that will assist developers by generating API descriptions for their Python web apps. Thus far, we have implemented this service for Google App Engine apps. Our API description generator runs on top of existing complete web apps. To overcome the challenge of dealing with a dynamically typed language, it uses a combination of static and dynamic parsing to generate descriptions of the APIs. We have also implemented validation of the produced API description according to standards proposed by [5].

Our service has some usage limitations, but these can be viewed as advantages in the testing phase of the development process. The restrictions and requirements of our tool, namely, the need to execute all decorated code and to document code in the specified format, will ensure that developers thoroughly test and document their applications. Additionally, the former constraint may be dealt with by employing automated testing as part of the API description generation process. We leave this, and other improvements, to future work.

References

[1] C. Krintz, RACE Lab, Department of Computer Science, University of California, Santa Barbara.

[2] H. Jayathilaka and S. Dimopoulos, RACE Lab, Department of Computer Science, University of California, Santa Barbara, ”RESTCoder”, May 2013.

[3] http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html

[4] http://docs.python.org/2/library/pydoc.html

[5] H. Jayathilaka, Mayhem Lab/RACE Lab, Department of Computer Science, University of California, Santa Barbara, ”API Description Validation Criteria,” May 2013.

[6] https://developers.google.com/appengine/docs/python/gettingstartedpython27/helloworld

[7] http://docs.python.org/2/glossary.html#term-decorator

[8] thadeusb.com/weblog/2010/8/23/python_multiple_decorators