LAPIN AMK
Lapland University of Applied Sciences

REST API

Implementation with Flask-Python

Alemu Musse Bekabil

Thesis
Degree Program in Information Technology

2014

LAPIN AMK
Lapland University of Applied Sciences

Abstract of thesis

Communication, Transport and Technology
Degree Programme in Information Technology

Author	Musse Alemu	Year	2014
Supervisor	Mattila Erkki		
Commissioned by	Oy Aurora Data and Systems Ltd.		
Title of thesis	REST API		
No. of pages + app.	44 + 2		

Communication between various systems is common in the technology world. Thus, this thesis report discussed one of the popular means of communication, REST API. Additionally, it described the six basic constraints of REST conjointly to HATEOAS constraint. Furthermore, it observed all the good advantages REST API has over SOAP. Moreover, it demonstrated practical implementation of RESTful web services.

The report started to discuss from the bigger picture, software architecture, and continued down to API level. Besides, it clearly documented how REST architectural principles are applied in API level. Moreover, it demonstrated the implementation of REST API using Flask-Python micro framework. As a result, the report used classical approach to introduce REST API.

Accordingly, in this report solid theoretical explanation is covered. Moreover, a step-by-step guide for practical use of RESTful web service also shown. Therefore, it suggested all the good reasons why REST is needed as a standard to be a means of communication between systems.

Key words RESTful web service, REST API, HATEOAS, Flask, Python

CONTENTS

LIST OF FIGURES

FORWARD

I would like to thank my thesis commissionaire company Oy Aurora Data and Systems Ltd. for giving me the opportunity to work with them. The thesis idea and practical implementation is organized and monitored by the commissionaire company. Specifically, I would like to thank Mr. Ville Mattila for spending his time for successive brainstorming sessions and commenting the first draft of the thesis document. Similarly, Mr. Heikki Mustonen also allocates his time to assist me concerning programming issues. Thus, with their kind cooperation, I could manage to write a standard thesis report.

SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
OS	Operating System
REST	Representational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Service Description Language
WWW	World Wide Web
XML	Extensible Mark-up Language

1 INTRODUCTION

Nowadays, the production of software product is increasing dramatically. As the production rate grows dramatically, communication between applications becomes vital. Plenty of web applications share resource through World Wide Web (WWW). However, in real world experience, this communication is not easy as it sounds. Hence, each company adopt various technologies and application layers to implement web products. For this reason, the world should discover standard way of communication method that every client easily consumes resources in spite of their technological variation. Thus, REST (Representational State Transfer) appears to make a solution for the implementation of web services. REST is an architectural style for distributed hypermedia systems, which its principles could be applied for communication between them (Fielding 2000). Therefore, this repost prefer to discuss a new way of sharing resources between network-based systems called REST API or RESTful web services.

The first chapter of the report describes the basic essence of Software Architecture, specifically for web applications. Moreover, it uncovers the basic essence of Design Patterns and summarizes the topic by discussing its elements. Following that, it analyses service based software design and software architecture design pattern, commonly known as Service-Oriented Architecture (SOA). Furthermore, it closely considers web service messaging protocol called SOAP (Simple Object Access Protocol) and its corresponding description language WSDL (Web Service Description Language).

After a clear review of preceded web application architectural protocols, chapter two introduces REST. First, it defines the term correctly and it discusses the six core constraints. Besides, the report explains how REST architectural principles are projected in API level. Moreover, this document includes the basic, however most forgotten element, HATEOAS. Finally, the chapter closes by suggesting all the good reasons why developers should adopt REST API as a standard.

The chapter discusses how RESTful web services are implemented. For demonstration, a simple blog post demo application presents a practical implementation of the API. This application uses Flask-Python web framework. Moreover, this section introduces the used technologies and mention reasons why they are chosen for the project. Additionally, it provides a step-by-step approach to show how resources are shared from persistent data source. Hence, the application’s API bases REST principles.

2 SOFTWARE ARCHITECTURE, DESIGN PATTERNS AND WEB SERVICES

2.1 Software Architecture

Software architecture is software's blue print, which is derived from the UML use case. The architecture of a system evolves as decisions are made in terms of feasibility, technical challenges, trade-offs, cohesion between stated requirements, fluctuating needs of stakeholders, and so on. This will require an extensive iterative process to refine the use case to reach for the final architecture. (Gulzar 2003, 50.)

On their work in Software Architecture, Software Engineering Institute at Carnegie Mellon University, defines software architecture as follows,

> *"The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them"* (Software Engineering Institute 2013).

Software architecture is a high-level abstraction of a system like the selection of methodologies, frameworks, scope and goals. Furthermore, it identifies the elements of a system (also referred as architectural elements). Architectural element is a fundamental piece from which a system considered to be constructed (Rozanski & Woods 2011, 20). Instead of defining the details of elements, the architecture designs how elements work, used by other elements and interact to each other. As a result, software architecture only depicts the bigger picture of the application.

After the skeleton of the software is designed, details of architectural patterns will be documented on the design pattern. Software design patterns focus on designing the responsibilities of the module or component. Often as classes and objects level. Hence, software's design pattern fills the smaller picture of the application.

Software architecture insists continuous communication between stakeholders Stakeholder is a traditional term used to refer a business partners who invested money for the software. It helps to correct architectural faults at the early age of the development.

2.2 Design Patterns

As Christopher Alexander, the Austrian architect introduced the notion of patterns in the field of construction.

> *"The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* (Alexander et al. 1977, x.)

This principle also sounds for object-oriented design. Once a given problem is solved, the methodologies should be documented well for further use. These solutions are expressed in terms of objects and the relationships between them. Therefore, design pattern guarantees a reusable designs and architectures. By doing this, a software architect can develop a systematic approach to improve the reusability of the system.

Design patterns are not the direct solution for a problem. They are the templates, which guide to solve a given problem. Furthermore, developers have a formalized standard to document patterns. Thus, they can understand and implement design patterns in their application.

2.2.1 Design Pattern Elements

According to the “Gang of Four (GoF)”, a design pattern has four essential elements. Gang of Four is a group name for authors that consists Gamma E., Helm R., Johnson R. and Vlissides J. They are authors of a well-known software engineering book called Design Patterns. These authors classify design pattern elements as pattern name, problem, solution and consequence.

Pattern name is a handle, normally a word or two, which enables to describe a design problem. It allows drafting the high-level abstraction. Assigning a vocabulary for patterns makes the communication and documentation easier. However, in real world developing experience, finding a perfect phrase has been the hardest job. (Gang of Four 1994.)

Problem is the crucial element of the design pattern in which it explains when these patterns could be applied. It describes the potential problems in the context. The description could be specific design problems. For instance, it could be on presenting algorithms as objects. In other cases, the problem is a list of prerequisite before applying the patterns. (Gang of Four 1994.)

Solution often describes the actual elements, which constitute the design. As it is explained above, solution does not describe a concrete design or implementation. Instead, it produces a high-level description of a design problem and how element arrangements solve it. (Gang of Four 1994.)

Consequences are the results and trade-offs of applying the pattern. Most commonly, consequences in design decision explained implicitly. However, they are critical for evaluating design alternatives and for estimating the costs and benefits of applying the pattern. As a result, knowing these consequences beforehand enables to understand and evaluate them. (Gang of Four 1994.)

2.3 Service-Oriented Architecture

Service-oriented architecture (SOA) is a design pattern build from logical units, which provides services in an application. It is a design philosophy independent of any vendor, product, and technology or industry trend. It can be simply defined as a loosely-coupled architecture designed to meet the business needs of the organization (Linthicum 1999). SOA is a term used to represent a model in which automation logic is decomposed into smaller, distinct units of logic. Collectively, these units comprise a larger piece of business automation logic. Individually, these units can be distributed. Within SOA, these units of logic are known as services. (Erl 2009, 375.)

It is common to distribute automation logic in other technologies too. The difference in SOA is that logic units are designed in such a way to evolve and grow relatively independent from each other. Even if the system allows them to interact to each other, it is must to avoid a model in which outlets form tight connections that result in constrictive interdependence. On the other hand, it is also mandatory to develop the system independently. (Erl 2009, 377.)

2.4 Web Services

As it was discussed above, small functions in big application are realized as services. Thus, web services are the implementation of these services in a networked system. According to W3C Web services are

> *"a software system designed to support interoperable machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems*

interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." (W3C Working Group Note 2004.)

Services could be any specific task, for instance, it could be sharing of files. For that reason, in a web service those services are performed over HTTP (Hyper Text Transfer Protocol). The client (the one who requires the service) sends a request. Most commonly, the request represented as serialized XML format as streams of bytes. Each request corresponds to a given URL. Following that, the web service receives the request. The received XML request will be de-serialized in to a data format. Finally, the web service processes the request and serializes the response to send it back to the client.

Web services in an application are reusable. A reusable web services are designed to suit for different applications, which will provide similar services. To achieve this it should implement at least two fundamental characteristics, statelessness and composability. Statelessness refers to the execution of request handled independently and without any intermediary state waiting for an event. Moreover, composability refers to a composition of different services to produce composite applications. For instance, a well-designed 'add to cart' service could be used for different e-commerce applications with no or less modifications. (Bechara 2009.)

2.4.1 Simple Object Access Protocol (SOAP)

Services are implemented in web services. These web services transfer messages to each other in a protocol called Simple Object Access Protocol (SOAP). Oracle defined SOAP as a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment (Oracle Corporation 2001).

Usually, the Extensible Mark-up Language (XML) based information will be sent through HTTP. A client sends a SOAP request for a service. The request is formatted in an XML format. This SOAP request will be posted through HTTP, if a secured connection is needed, the transport protocol will be HTTPs. Finally, the SOAP server handles the XML document request and responds for the request in the same data format and transport protocol.

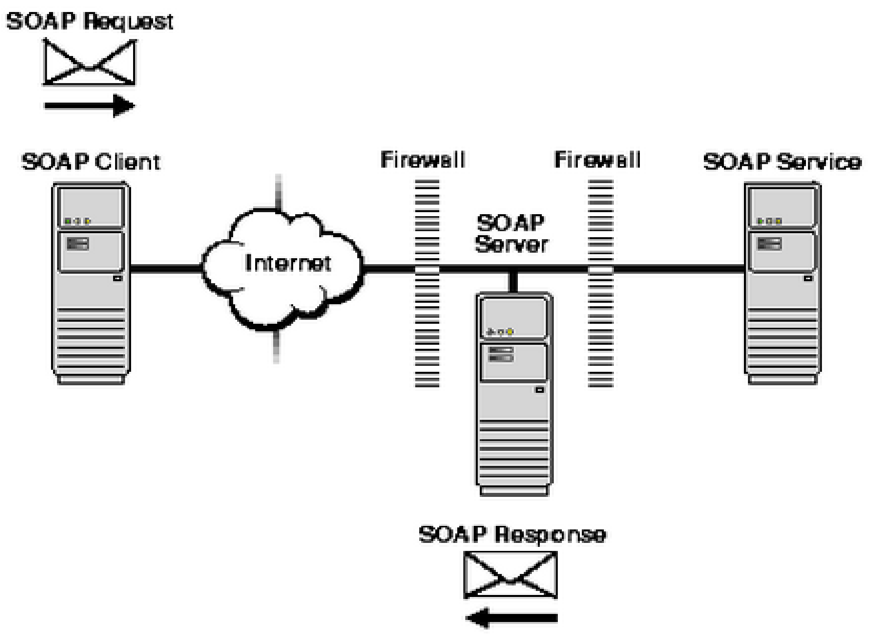
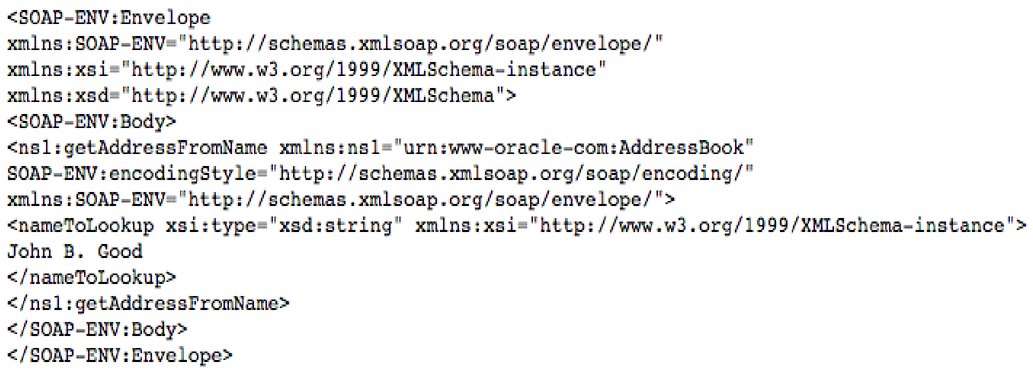


Figure 1. Components of the SOAP architecture (Oracle Corporation 2001)

As it is shown in the above Figure 1, HTTP protocol enables SOAP message to pass through a firewall. Basically, firewalls do not block traffic that crosses port 80, as HTTP does. Therefore, it makes easy the communication between SOAP client and service.

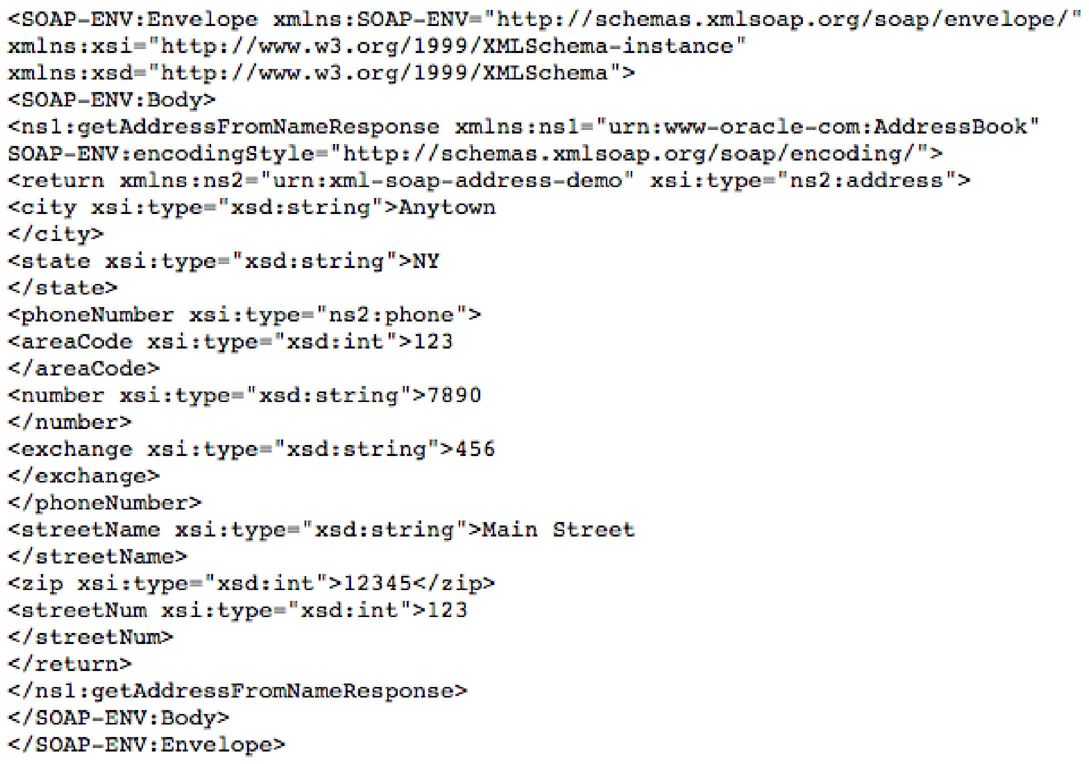
SOAP has its own XML format and rules. As similar to mail service, SOAP message is contained in an envelope. This commonly has a header that includes the namespace. Additionally, the message has body, which contains the actual message to be transferred over HTTP. Figure 2 shows a sample SOAP request for address book listing service.



```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getAddressFromName xmlns:ns1="urn:www-oracle-com:AddressBook"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<nameToLookup xsi:type="xsd:string" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
John B. Good
</nameToLookup>
</ns1:getAddressFromName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2. SOAP request (Oracle Corporation 2001)

Additionally, its corresponding response from the server depicts in Figure 3 below.



```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getAddressFromNameResponse xmlns:ns1="urn:www-oracle-com:AddressBook"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="urn:xml-soap-address-demo" xsi:type="ns2:address">
<city xsi:type="xsd:string">Anytown
</city>
<state xsi:type="xsd:string">NY
</state>
<phoneNumber xsi:type="ns2:phone">
<areaCode xsi:type="xsd:int">123
</areaCode>
<number xsi:type="xsd:string">7890
</number>
<exchange xsi:type="xsd:string">456
</exchange>
</phoneNumber>
<streetName xsi:type="xsd:string">Main Street
</streetName>
<zip xsi:type="xsd:int">12345</zip>
<streetNum xsi:type="xsd:int">123
</streetNum>
</return>
</ns1:getAddressFromNameResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3. SOAP response (Oracle Corporation 2001)

2.4.2 Web Services Description Language (WSDL)

SOAP message is not enough to make a sensible communication between systems. There must be some description about the message. To accomplish this WSDL (Web Services Description Language) is commonly used. Hence, it is the main drawback of using SOAP services. In practical implementation of these services, the messaging structure and the description language, makes the actual message transfer heavy. Moreover, these services often-configured in systems with firewalls that creates various layers for the messaging to be difficult. For that reason, it is becoming out-dated.

WSDL is used to describe network services. Like SOAP, WSDL documents are encoded in XML schema. Furthermore, a WSDL file contains the machine-readable description of how the web service can be called, what parameters it expects, and what data structures it returns. Figure 2 below shows the hierarchy on the communication network endpoints. Thus, WSDL is the description of the endpoints and actual SOAP message that the web service would communicate through HTTP protocol. (Oracle Corporation 2001.)

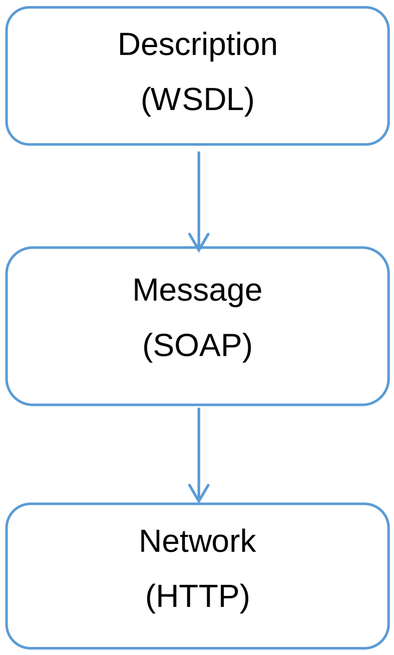


Figure 4. SOAP message

3 REPRESENTATIONAL STATE TRANSFER (REST)

The previous chapter was discussed about the very common network based way of communication, web services with SOAP. It is a means of decomposing application functions to smaller web services and designing these services to function and interact to each other. Additionally, this chapter will elaborate a new architectural paradigm - Representational State Transfer (REST). Firstly, the term will be defined. Secondly, the design principles and core constraints will be reviewed. Finally, it will suggest the good reasons why developers should adopt REST API.

Roy T. Fielding is “the father of REST”. He is the one who introduced REST to the world in his dissertation. Furthermore, he presented REST as an architectural style for distributed hypermedia systems. The design philosophy behind REST is that it extracted from various network-based architectural styles and combined with additional constraints. These will then create new architectural style. (Fielding 2000.)

According to Fielding, REST has six basic constraints. Every constraints build on the top of the other architectural constraint. Moreover, the design starts from “a blank slate” and adds architectural constraints on one to another to build the needed system. Thus, this section covers the six constraints in detail and discusses how RESTful web services can be constructed based on these elements. (Fielding 2000.)

3.1 REST Constraints

3.1.1 Client-Server

The background principle behind client-server constraint focused on the separation of the two components (client and server). Well-designed RESTful API should allow this separation of components and be able to evolve independently from one another. Separation refers that the client is unaware of the details implementation of the server. Similarly, the server is unaware of the implementation and technical dependencies of the client. In addition to their separation, it should also allow the communication between components. (Fielding 2000.)

As it is clearly showed in the Figure 3 below, the server is the one often accepts the client request and provides the needed service. Thus, the client is depicted as a service consumer on RESTful API.

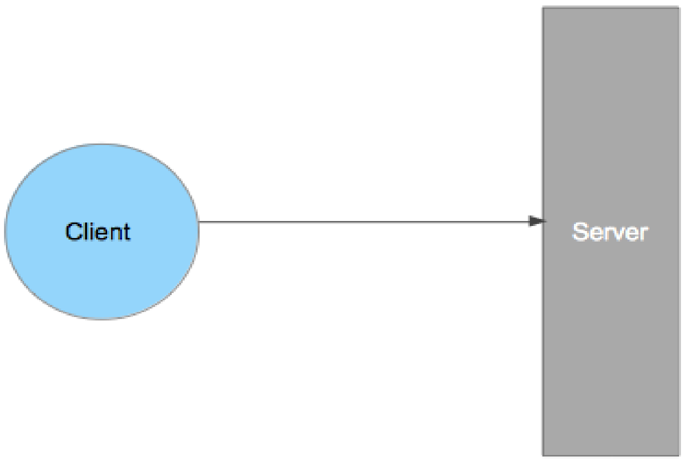


Figure 5. Client-Server

3.1.2 Stateless

The next constraint will be added on the top of client-server architecture, to constitute client-stateless-server. Statelessness is the way of providing service independent of any preceding stored state on the server. Hence, the session is stored and managed in the client side. Therefore, the server can handle many requests at a time, as it is shown in Figure 4 below, without considering the details of the client making the request. (Fielding 2000.)

This constraint boosts the scalability of the system. Since the server “does not care” who made the request, it can provide resources quickly. However, this architectural constraint sent data repeatedly to the server, which reduces network performance.

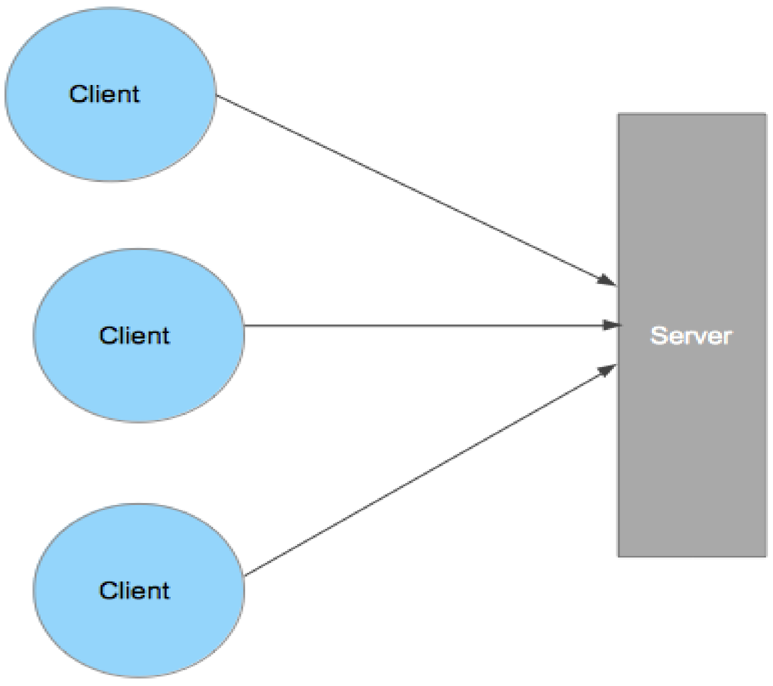


Figure 6. Client-Stateless-Server

3.1.3 Cache

In order to improve the network efficiency, cache constraint is added. As depicted in Figure 5, this new architectural style form client-cache-stateless-server. The server responses could be saved in the client cache. These cached responses help to avoid client to send same request to the server. As a result, it enhances the network efficiency by dodge repeated requests. (Fielding 2000.)

Moreover, this could be done implicitly, explicitly or negotiated. If the client stores the service responses without notifying the server, then it is cached implicitly. On the contrary, when the client notifies the server, it is explicit. Unlike both ways, client and server would negotiate how services could be cached.

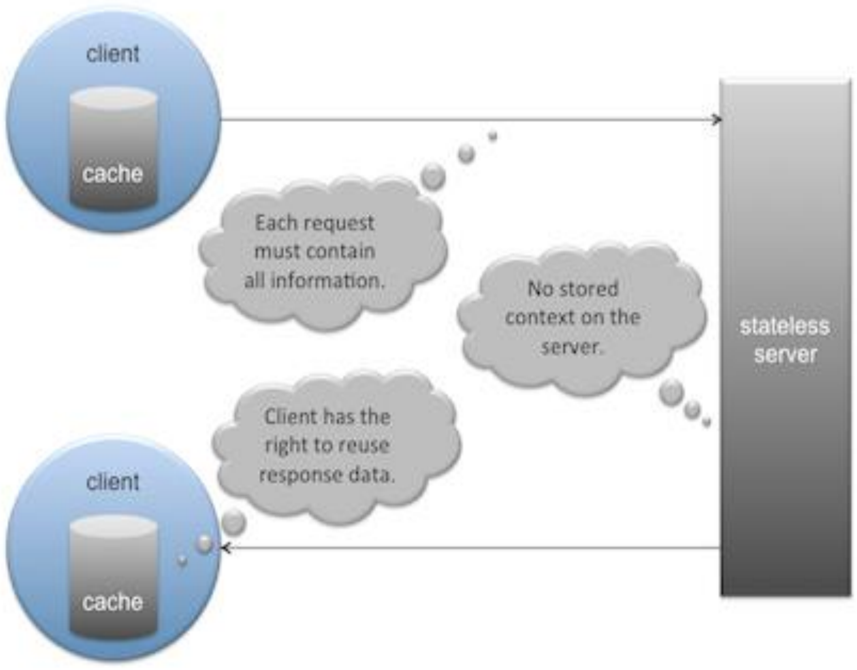

client
cache
Each request must contain all information.
No stored context on the server.
stateless server
Client has the right to reuse response data.
client
cache

Figure 7. Client-Cache-Stateless-Server (Mulloy 2013a)

3.1.4 Uniform Interface

Uniform interface constraint is added on the top of client-cache-stateless-server. This is the fundamental constraint that differentiates REST from any network-based architecture. George Reese explained it well in his book titled, The REST API Design Handbook. He suggested that uniform interface constraint enables to view representations of resources and interact with them through generic, finite set of requests. This could be implemented in a way that resources are referenced via URI and operated on through HTTP verbs. (Fielding 2000 & Reese 2012.)

Figure 6 below shows RESTful state transitions. Furthermore, each state has predictable transitions. The figure also shows the resource has finite states that the client possibly transit.

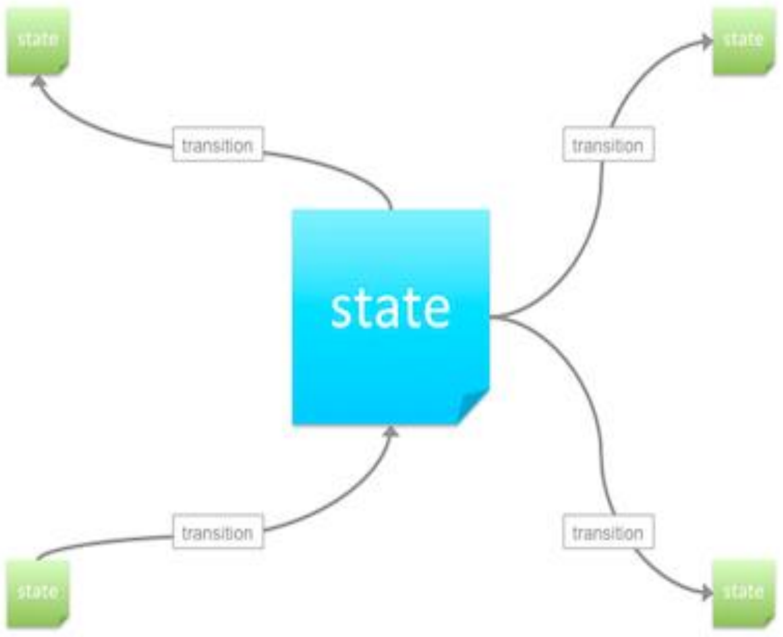


Figure 8. Uniform-Client-Cache-Stateless-Server (Mulloy 2013a)

3.1.5 Layered System

On network-based systems, a client does not often have direct communication to the server. There are both hardware and software intermediaries between them. For instance, real world system could be configured like in Figure 7 below. Therefore, a layered system constraint is added on a uniform-client-cache-stateless-server. (Fielding 2000.)

By having a layered system, RESTful service ensures that clients are able to communicate only with intermediary layer and other layers are invisible for them. This improves the scalability. Since, in RESTful service, intermediary components can communicate each other. This is made possible because of the self-descriptive capability of the content resource. Moreover, the URI for each possible state is visible for mediator technologies.

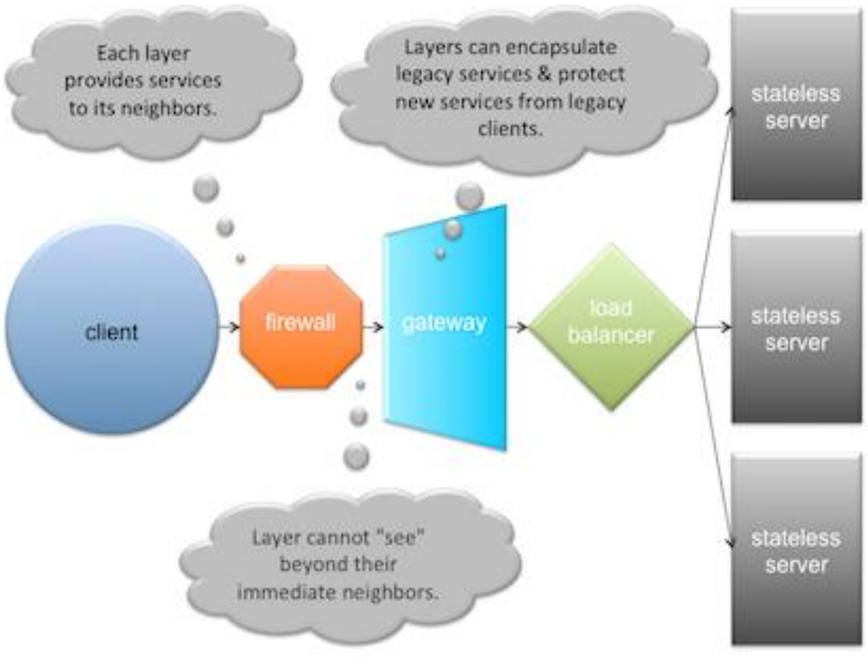

Each layer provides services to its neighbors.
Layers can encapsulate legacy services & protect new services from legacy clients.
stateless server
client
firewall
gateway
load balancer
stateless server
stateless server
Layer cannot "see" beyond their immediate neighbors.

Figure 9. Uniform-Layered-Client-Cache-Stateless-Server (Mulloy 2013a)

3.1.6 Code-On-Demand

Unlike the above constraints, code-on-demand is optional for RESTful services. Basically, it is a means that RESTful service extend client's request by providing executable code. This can be implemented by extending client functionality to download and execute code in the form of Java Applets or JavaScript. Figure 8 shows this scenario below. (Fielding 2000.)

The idea behind optional constraint seems ludicrous. Though, it is rational to include in the architectural design for systems, which has multiple organizational boundaries. Commonly, this optional constraint facilitates the communication unless it is disabled within some context.

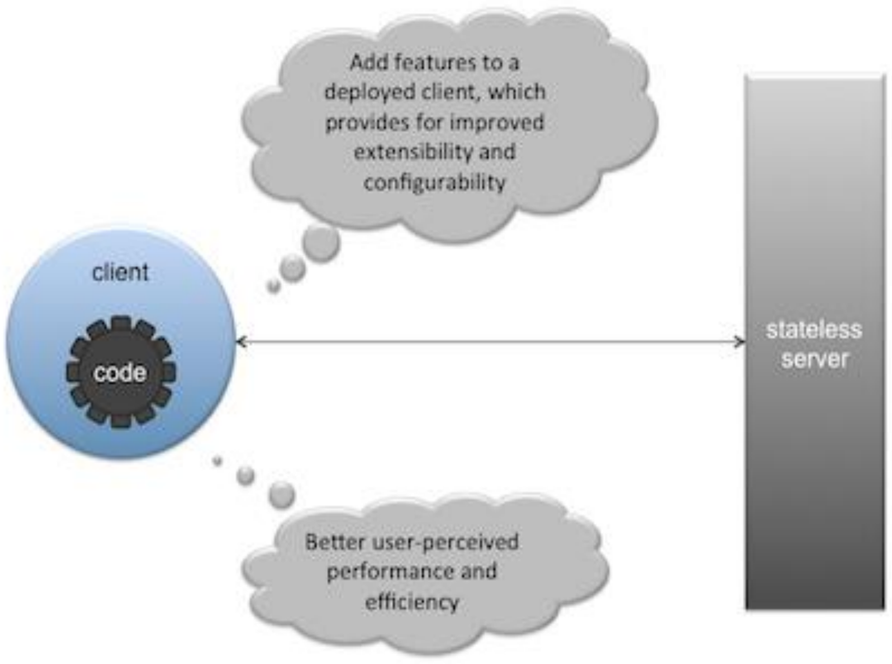

Add features to a deployed client, which provides for improved extensibility and configurability
client
code
stateless server
Better user-perceived performance and efficiency

Figure 10. Code-on-demand (Mulloy 2013a)

3.2 Hypermedia as the Engine of Application State (HATEOAS)

HATEOAS is one of the major constraints of REST. Each resource should have the entry for action and link, which is the list of possible actions that the client could do and the links are the path to the resource. These are key parts of the resource that enable it to be hypertext driven. However, most developers failed to include this constraint. By principle, using HTTP based interface does not make any API RESTful. These APIs use HTTP methods and status code to communicate client and server. On the contrary, they failed the role of hypermedia. Thus, most of “the so called” RESTful services aren’t nothing than implementation of RPC (Remote Procedure Call) on HTTP based interface.

Roy addressed this controversy on his blog published on 2008 titled as REST APIs must be hypertext-driven. He asserts that,

> *“If the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API, Period”* (Fielding 2008).

This means, each states are represented as hypertext. Every state then has a list of states those are possibly reachable from the state. Similar to web implementation, one can traverse from page to another page using the possible hyperlinks, even without referring the site map. Therefore, hypertext is the key constraint in REST API.

This enables REST API to be self-documenting. A single resource has data section that describes itself, action(s) that the client does with the resource and link(s) that the client possibly transit to the next state. These collectively make it self-descriptive. Since, each representations of a state are self-descriptive, clients does not have the responsibility to know about resources coming from the server. Therefore, extra documentation is not needed to define resources.

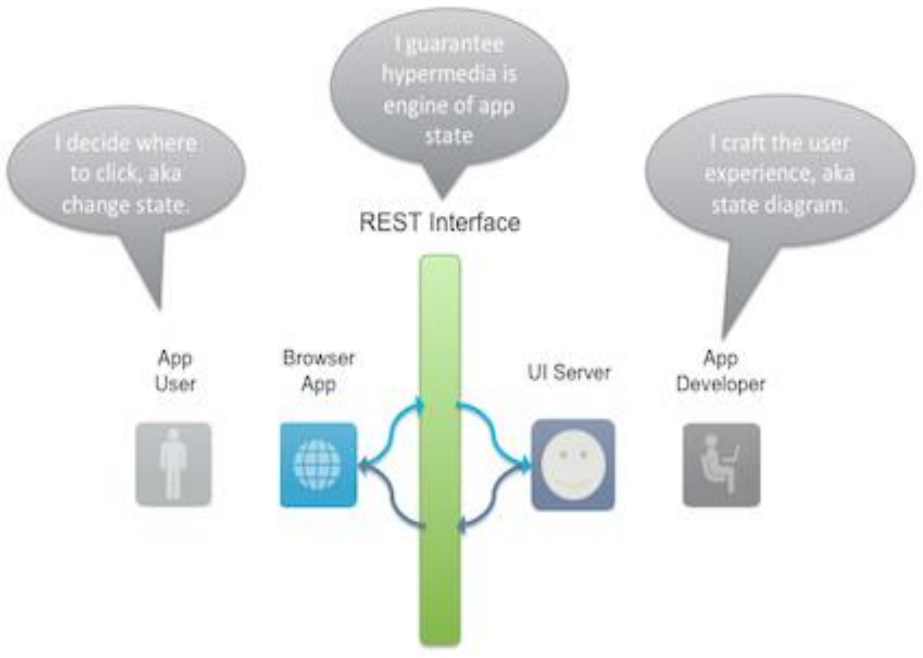

I guarantee hypermedia is engine of app state
I decide where to click, aka change state.
I craft the user experience, aka state diagram.
REST Interface
App User
Browser App
UI Server
App Developer

Figure 11. HATEOAS (Mulloy 2013b)

Figure 9 above depicts the common system interaction. This interaction handled by using REST interface in the middle. Moreover, each user clicks or change of state should be implemented in REST interface. As a result the service is RESTful that the app uses hypermedia as an engine of state transition.

3.3 SOAP and REST

There are plenty of debates happening on comparison between SOAP and REST. This report will not repeat the same thread. Instead, it focuses on the reasons why one should adopt REST while the equivalent web service SOAP is available. Therefore, it presents four core advantages of REST have over SOAP or any other web service technologies.

The first thing to be clarified is that, REST is not only a web service and it is not developed to replace SOAP. As it was mentioned above, REST is an architectural style. However, its design principles could be projected and customized to function as a web service. For this reason, its basic essence is misunderstood and wrongly implemented.

3.3.1 Applied REST is Simple

As it was mentioned in Chapter one, the theoretical principles of SOAP look fine. However, the actual implementation is complex. Specially, working with distributed systems, it is difficult to debug. Since SOAP hides the complexity, it is confusing to identify which part is broken. In addition to this, SOAP implementation is greatly depending on the IDE and programming language. For instance, in Visual Studio SOAP is easy to implement. In the contrary, it lacks to provide best libraries for Python developers.

Unlike SOAP, REST tries to overcome these problems. RESTful web services only use HTTP protocol, which makes it simple, and light weighted. Hence, it is easy for actual implementation and debugging. Moreover, RESTful services are both platform and language-independent. Since, HTTP is the standard, it flexibly work with different platforms (UNIX, Windows and Mac) and programming languages. As a result, a REST API is light weighted, platform and language independent web service.

3.3.2 Use Advantages of HTTP

SOAP uses HTTP only as a transport protocol. However, RESTful services use HTTP's advantage to the most. REST performs all CRUD (Create Read Update Delete) operations using HTTP built-in GET, POST, PUT and DELETE methods.

> GET is a method used to fetch a resource identified by a given URI. It performs the Read operation.
>
> POST is a method used to create a new instance of a resource. It performs the Create operation. Here, post method is applied for new resource, instead of updating the existing one.
>
> PUT is a method used to update a resource. It performs the Update operation. Unlike POST, it only executes to update a resource.
>
> DELETE is a method used to delete a resource identified by a given URI. It performs the DELETE operation.

In addition to HTTP methods, RESTful web services use rich HTTP status codes. This enables the client to utilize the response sent from the server in HTTP status code. Thus, any client who is unaware of the server could understand the response, for the reason that HTTP status codes are standards. Furthermore, the developer is not responsible to write any custom error handling codes. Therefore, a RESTful service can handle errors by using HTTP status codes sent from the server. (Reese 2012.)

3.3.3 Self-Documenting

The key constraint distinguishes REST from other HTTP based RPC is that it is being driven by hypertext. There is some RPC APIs other than SOAP that could possibly use rich advantage of HTTP. As it is depicted in the Figure 1, there is a high dependency with other technologies, to use the services, on the top of HTTP protocol. Unlike other RPC RESTful web services operated by hypertext, which enables them to be self-documenting. The client using the resource could easily understand what to do with the resource responded from the server. This could be done without any description language or separate documentation. As a result, well-designed RESTful web services do not need a separate documentation or description language layer.

3.3.4 Resource-Oriented Architecture (ROA)

SOAP is an implementation of Service Oriented Architecture (SOA). It is driven by services and the interaction between services. However, REST is an architectural style, used to realize the big picture of the software by comprehending the system as a resource. It is best used with ROA. As it was discussed above, this principle could be projected to web services. Thus, in RESTful service, every response from the server is a representation of a resource and the client is traverse from state to state. Each resource in the server has unique identifier called URI (Uniform Resource Identifier) and the client can access that resource with the given URI through HTTP.

Furthermore, REST is not HTTP dependent. It can work well even with other transfer protocols like FTP (File Transfer Protocol). Moreover, REST constraints are built independent of other technology and platform. Therefore, RESTful web services perform sufficiently on systems, which do not use HTTP. Since, most of the web uses HTTP protocol; this report used it as a default transfer protocol.

4 IMPLEMENTATION OF REST API

The previous chapters discussed REST API from theoretical point of view. This chapter will show step-by-step implementation using small blog post web app to demonstrate how a RESTful web service could be applied. This app is developed using Flask-Python web framework. Thus it is quite simple and clear enough to show the basic constraints of RESTful web services. As a result, this report concludes by giving both theoretical and practical explanation about REST API.

Furthermore, the following sub topics show how to use various open source python frameworks and libraries. Hence, this avoids doing everything from scratch. Moreover, it mentions the reasons why these specific frameworks are chosen for the demo application. Therefore, it will give full package guide to begin with RESTful web services.

This demo app is developed on Linux platform, Ubuntu 13.04 Operating System (OS) for the implementation. There are a couple of reasons for this. Firstly, Linux is open source OS, which licensed by GNU GPL (GNU General Public License). Thus, anyone can download and install it for free. Secondly, Ubuntu is easy to work with python projects and git version control systems. It supports different open source frameworks and libraries that are available for python. As a result, consider that there might be some changes in the implementation for users on different platform, especially Windows OS.

4.1 Flask

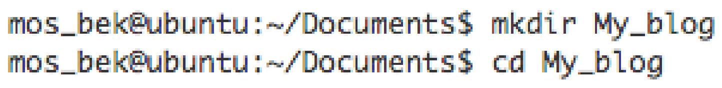
Flask is a micro web framework for python projects, which enables developers to create web applications easily. It is a general framework that could be applied for different scale projects. For instance, it could be used for small web apps to a wide range social networking websites. The fundamental principles are the same, which can easily be projected to any scale web based applications.

There are three reasons for choosing Flask for this demo application. The first one is its simplicity. In Flask, one should not need to know everything from the beginning. The framework could be learned while developing. It has quick start documentation that guide developers to up-and-run the application with basic functionalities. The second reason is its openness. As with Python, Flask software is distributed under a permissive open source license. These makes more accessible and advanced. Finally, it is well-documented open source framework. It has a detailed tutorial besides its quick start documentation. Moreover, it provides up-to-date documentation for every new versions of the framework. Therefore, these reasons made Flask web framework preferable for the blogging web service demo app.

4.2 Virtualenv

Virtualenv is a tool to create isolated Python environments. Technically, it creates independent working directories. So, each virtualenv has its own libraries, which does not share with others. This project developed by using virtualenv version 1.6. (Bicking 2014.)

Here is a Linux terminal command to install virtualenv in the local machine. First, create the project directory in the location, which is easy to access.



```
mos_bek@ubuntu:~/Documents$ mkdir My_blog
mos_bek@ubuntu:~/Documents$ cd My_blog
```

Then install the virtual environment if it is not yet installed in the machine. After this, use the virtualenv command to create distinct virtual environment. Finally, activate the virtual environment.

```
$ pip install virtualenv
$ virtualenv venv
$ source venv/bin/activate
```

4.3 Flask-SQLAlchemy

The SQLAlchemy is an open source product that provides Object Relational Mapper (ORM). Moreover, it usually assists python model classes to be mapped to database tables. Each instance of the database class represented a record in the database table. Additionally, the class variables of the model class associated as the column of the corresponding table. Furthermore, it communicates database queries through model classes and relationships between each other. (SQLAlchemy 2014.)

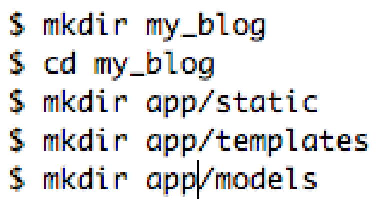
Flask-SQLAlchemy is one of Flask's extension libraries. It provides easy and efficient built-in functionalities to cooperate with SQLAlchemy. It requires SQLAlchemy version 0.6 is the minimum requirement that supports Flask-SQLAlchemy extension. Mainly, it facilitates the work for Flask users by providing useful defaults and extra helpers that make it easier to accomplish common tasks. (Ronacher 2011.)

The next task is to install all the necessary dependencies in the virtual environment library. Here is a list of command to do it.

```
$ venv/bin/pip install flask
$ venv/bin/pip install sqlalchemy
$ venv/bin/pip install flask-sqlalchemy
```

The first command installs the flask micro framework library files to the local virtual environment package. Thus, other flask extension libraries, like flask-sqlalchemy depends on the basic flask package. For that reason, flask installation should be run before its extension libraries. Next to this, there is a command for flask-sqlalchemy to finalize the installation process.

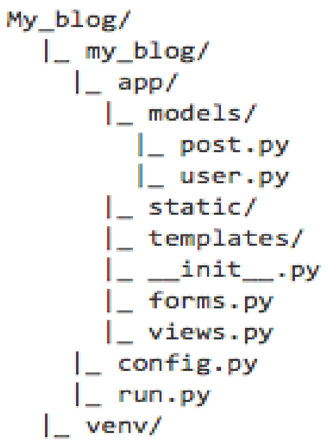
Following this, the working directory and structure could be formatted in such a way that it can be manageable and human readable. Thus, here is the command below to create the working directories.



```
$ mkdir my_blog
$ cd my_blog
$ mkdir app/static
$ mkdir app/templates
$ mkdir app/models
```

The app folder is a parent folder for the application so that the controller, configuration and initialization files could be kept. Static sub-folder is a place to put static files like CSS, Javascript and image files. Likewise, templates sub-folder allows storing HTML template files. Finally, the models sub-folder used to put model classes.

Following this, a guide to create basic files is illustrated. Firstly, the initialization file in app/__init.py__ should be created. Secondly, the handlers (controllers) are created as app/views.py. This file is used to handle user's request. It maps URLs to Flask function. Basically, this function responds browser request by rendering the corresponding HTML file. Finally, a script, which used to start the application by setting up the web server and initializing the application, is created. Thus, this setting script is saved as run.py file. The overall working directory resembles as the image depicted in Figure 10.

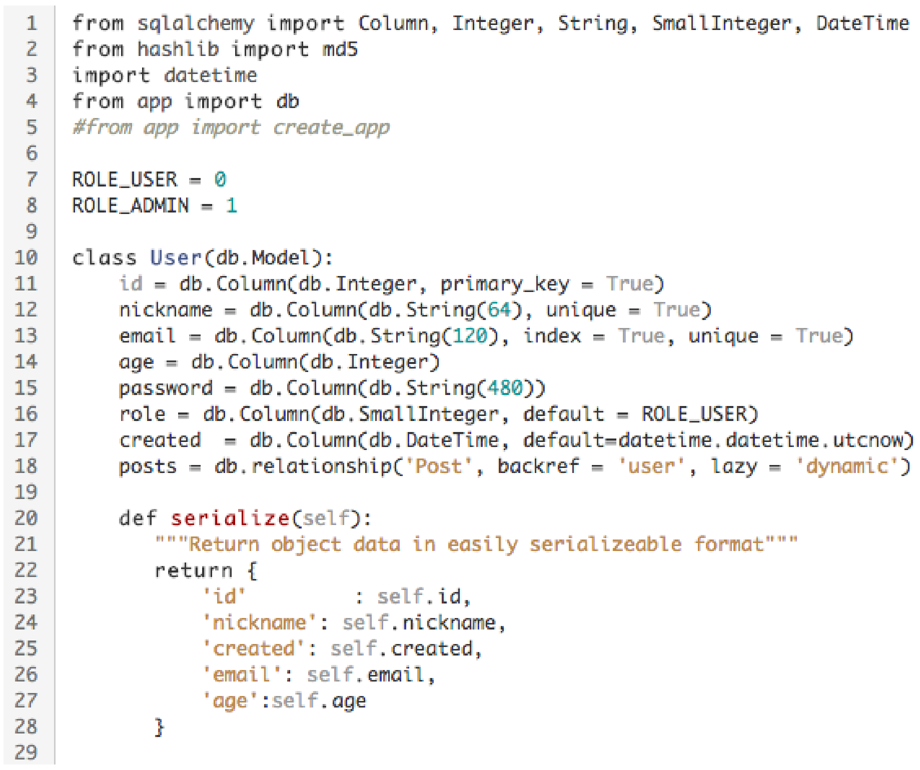


```
My_blog/
   |_ my_blog/
      |_ app/
         |_ models/
            |_ post.py
            |_ user.py
         |_ static/
         |_ templates/
         |_ __init__.py
         |_ forms.py
         |_ views.py
      |_ config.py
      |_ run.py
   |_ venv/
```

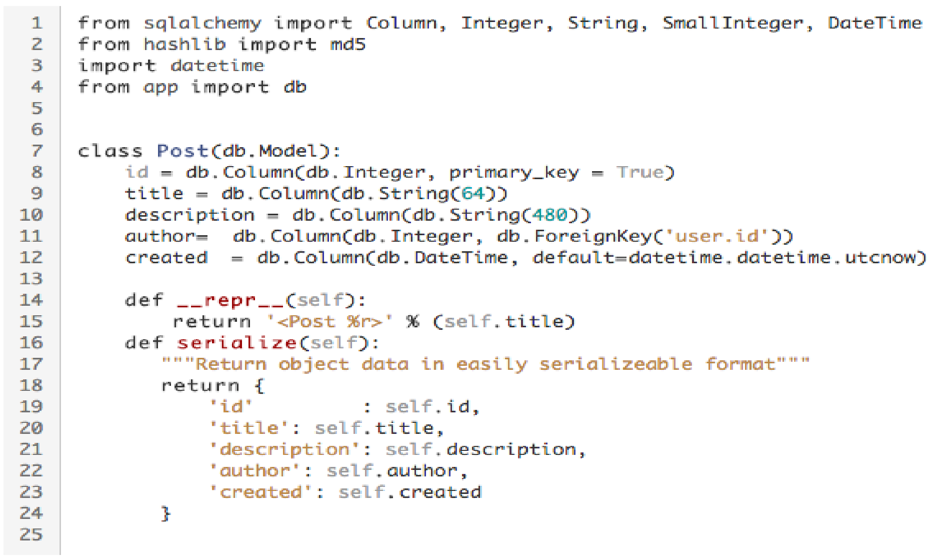
Figure 12. Working directory

This thesis document illustrates how the basic REST API calls work. For that reason, it will not cover how to set up and run Flask application. In addition to this, Miguel Grinberg’s tutorial blog is recommended for beginners (Grinberg 2012). He tried to show how to build Flask based web apps from scratch with adequate explanation.

Furthermore, a RESTful web services is implemented here. Firstly, the model classes are created based on Flask-SQLAlchamey extension. Here is a sample of User and Post model class respectively.



```python
from sqlalchemy import Column, Integer, String, SmallInteger, DateTime
from hashlib import md5
import datetime
from app import db
#from app import create_app

ROLE_USER = 0
ROLE_ADMIN = 1

class User(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    nickname = db.Column(db.String(64), unique = True)
    email = db.Column(db.String(120), index = True, unique = True)
    age = db.Column(db.Integer)
    password = db.Column(db.String(480))
    role = db.Column(db.SmallInteger, default = ROLE_USER)
    created  = db.Column(db.DateTime, default=datetime.datetime.utcnow)
    posts = db.relationship('Post', backref = 'user', lazy = 'dynamic')

    def serialize(self):
       """Return object data in easily serializeable format"""
       return {
           'id'         : self.id,
           'nickname': self.nickname,
           'created': self.created,
           'email': self.email,
           'age':self.age
       }

```



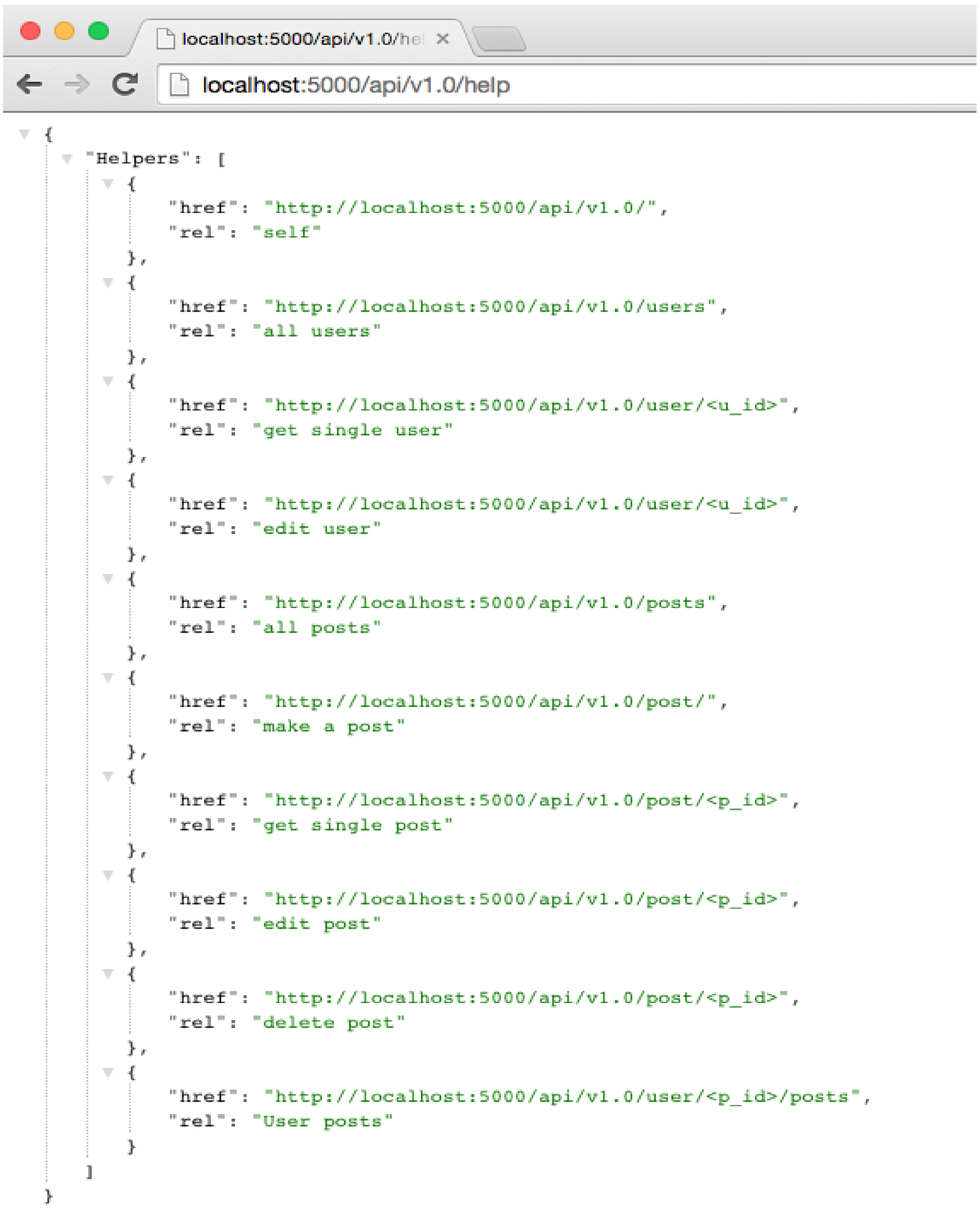
```python
from sqlalchemy import Column, Integer, String, SmallInteger, DateTime
from hashlib import md5
import datetime
from app import db


class Post(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    title = db.Column(db.String(64))
    description = db.Column(db.String(480))
    author=  db.Column(db.Integer, db.ForeignKey('user.id'))
    created  = db.Column(db.DateTime, default=datetime.datetime.utcnow)

    def __repr__(self):
        return '<Post %r>' % (self.title)
    def serialize(self):
       """Return object data in easily serializeable format"""
       return {
           'id'         : self.id,
           'title': self.title,
           'description': self.description,
           'author': self.author,
           'created': self.created
       }

```

The model class is an extension of SQLAlchemy's Model class. Furthermore, the class variables of the model class create the corresponding fields in the database by instantiating Column class from SQLAlchemy object. It is must to pass the field type as a parameter like (String or Integer) while creating the field instances. Moreover, plenty of values could be passed to characterize the fields, which will be created in the database server. For instance, the primary key could be set to handle null values and the like.

Following this, three basic things should be mentioned concerning the implementation. The first notice will be about the wise use of URIs. The URI is constructed as root-url/api/v1.0. So, this URI represents the version one API of the web application running on root-url. In this case, the application is running on localhost port 5000, which implies that the API call follows the pattern http://localhost:5000/api/v1.0. Hence, this allows developers to add web API for the upcoming versions of the application with a combination of unique nouns.

Secondly, in this application curl command is used to perform all HTTP requests. It is easy to perform GET requests from the browser. Thus, Figure 11 and 12 shows simple GET requests from the browser. However, for POST, PUT and DELETE requests curl command is used from the Terminal. Therefore, curl libraries should be installed on the local machine to run all the commands in this report document. Moreover, this demonstration uses JSON format to send and receive resources over HTTP. However, it does not mean that REST is JSON dependent. Hence, the same resource could be transferred using XML script.

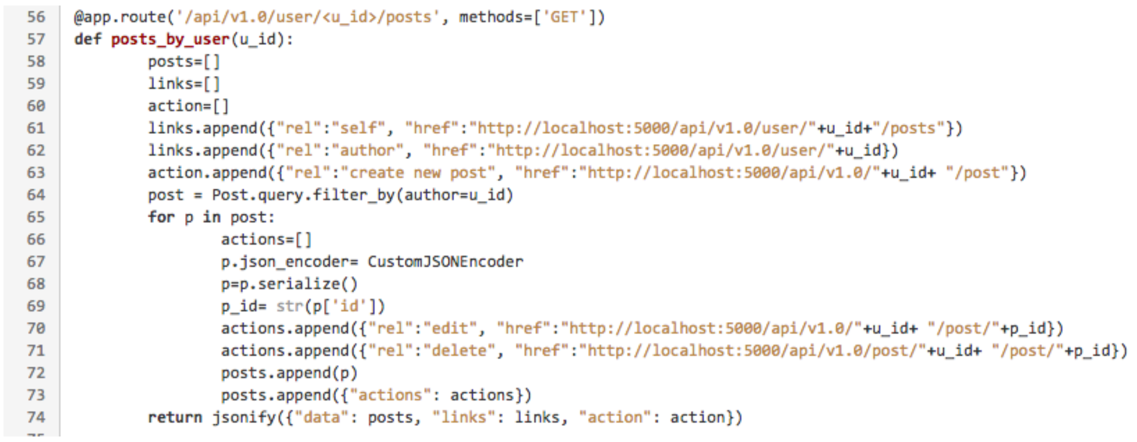
Finally, HATEOAS implementation is shown practically. As it was explained in the previous chapters, HATEOAS is the crucial part of REST API. It enables one resource to be self-describing. Thus, a resource that has identified by URI has different states. Therefore, REST API will have the data section that is the resource and link to show the possible state transitions. For instance, optional helper resources can also be implemented as in Figure 11, which shows all possible states in the API. Hence, the idea of implementing optional helper resource is controversial. Most of REST API developers prefer individual resources to be self-documenting, rather than adding an additional site map for the API. However, they can be useful in some other cases. For that reason, it is included to this demonstration and shown below.



```
{
  "Helpers": [
    {
      "href": "http://localhost:5000/api/v1.0/",
      "rel": "self"
    },
    {
      "href": "http://localhost:5000/api/v1.0/users",
      "rel": "all users"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/<u_id>",
      "rel": "get single user"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/<u_id>",
      "rel": "edit user"
    },
    {
      "href": "http://localhost:5000/api/v1.0/posts",
      "rel": "all posts"
    },
    {
      "href": "http://localhost:5000/api/v1.0/post/",
      "rel": "make a post"
    },
    {
      "href": "http://localhost:5000/api/v1.0/post/<p_id>",
      "rel": "get single post"
    },
    {
      "href": "http://localhost:5000/api/v1.0/post/<p_id>",
      "rel": "edit post"
    },
    {
      "href": "http://localhost:5000/api/v1.0/post/<p_id>",
      "rel": "delete post"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/<p_id>/posts",
      "rel": "User posts"
    }
  ]
}
```

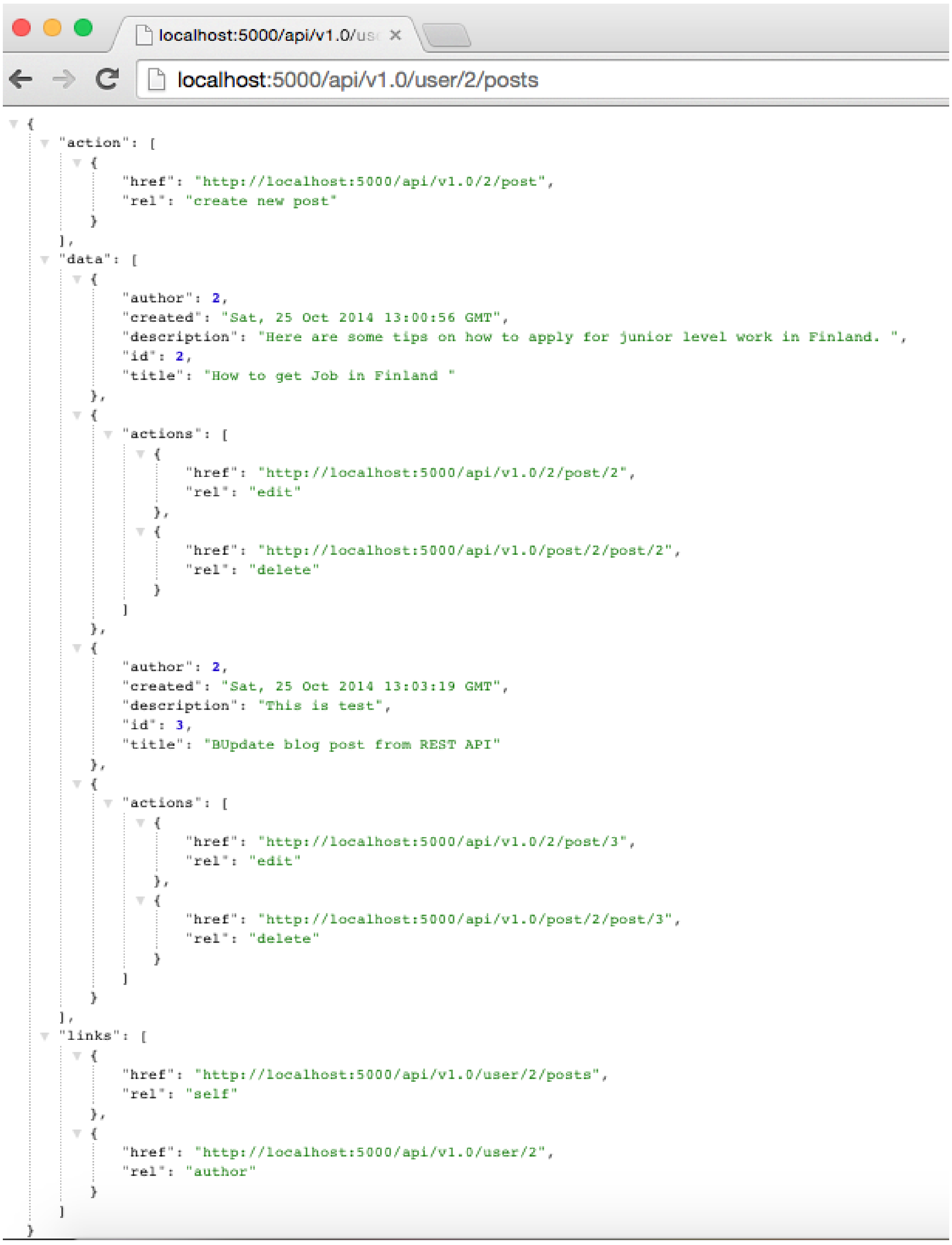
Figure 13. API helper

Here is a demonstration of how resources are represented in web application. These resources are serialized to JSON format from SQLAlchemy database object. Hence, this improves the API's performance by letting users to consume it easily. The First example shows how to represent list of post resources made by a given user identified by its id. Additionally, this representation also shows the possible links (states) that one could transfer from this node. Thus, a separate documentation should not be prepared as a user's guide. This could be done by sending GET request to the server using the URI http://localhost:5000/api/v1.0/user/2/posts.Here is the code snippet in app/views.py file.



```python
@app.route('/api/v1.0/user/<u_id>/posts', methods=['GET'])
def posts_by_user(u_id):
        posts=[]
        links=[]
        action=[]
        links.append({"rel":"self", "href":"http://localhost:5000/api/v1.0/user/"+u_id+"/posts"})
        links.append({"rel":"author", "href":"http://localhost:5000/api/v1.0/user/"+u_id})
        action.append({"rel":"create new post", "href":"http://localhost:5000/api/v1.0/"+u_id+ "/post"})
        post = Post.query.filter_by(author=u_id)
        for p in post:
                actions=[]
                p.json_encoder= CustomJSONEncoder
                p=p.serialize()
                p_id= str(p['id'])
                actions.append({"rel":"edit", "href":"http://localhost:5000/api/v1.0/"+u_id+ "/post/"+p_id})
                actions.append({"rel":"delete", "href":"http://localhost:5000/api/v1.0/post/"+u_id+ "/post/"+p_id})
                posts.append(p)
                posts.append({"actions": actions})
        return jsonify({"data": posts, "links": links, "action": action})
```

Furthermore, its corresponding response from the browser is depicted in Figure 12. All the information about the resource presented in the data section of the resource. Moreover, the HATEOAS section shows the list of states in the link section of the resource.

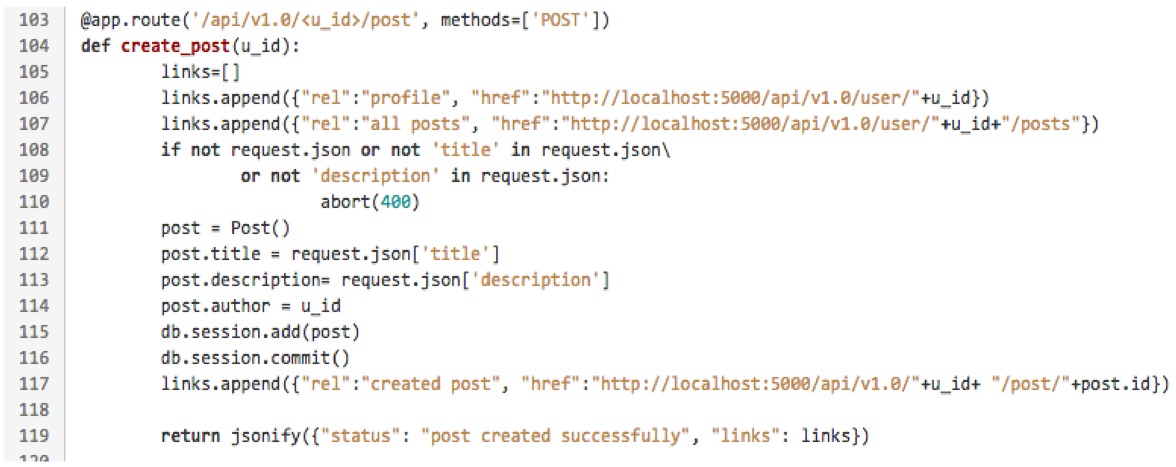


localhost:5000/api/v1.0/user/2/posts

```
{
    "action": [
        {
            "href": "http://localhost:5000/api/v1.0/2/post",
            "rel": "create new post"
        }
    ],
    "data": [
        {
            "author": 2,
            "created": "Sat, 25 Oct 2014 13:00:56 GMT",
            "description": "Here are some tips on how to apply for junior level work in Finland. ",
            "id": 2,
            "title": "How to get Job in Finland "
        },
        {
            "actions": [
                {
                    "href": "http://localhost:5000/api/v1.0/2/post/2",
                    "rel": "edit"
                },
                {
                    "href": "http://localhost:5000/api/v1.0/post/2/post/2",
                    "rel": "delete"
                }
            ]
        },
        {
            "author": 2,
            "created": "Sat, 25 Oct 2014 13:03:19 GMT",
            "description": "This is test",
            "id": 3,
            "title": "BUpdate blog post from REST API"
        },
        {
            "actions": [
                {
                    "href": "http://localhost:5000/api/v1.0/2/post/3",
                    "rel": "edit"
                },
                {
                    "href": "http://localhost:5000/api/v1.0/post/2/post/3",
                    "rel": "delete"
                }
            ]
        }
    ],
    "links": [
        {
            "href": "http://localhost:5000/api/v1.0/user/2/posts",
            "rel": "self"
        },
        {
            "href": "http://localhost:5000/api/v1.0/user/2",
            "rel": "author"
        }
    ]
}
```

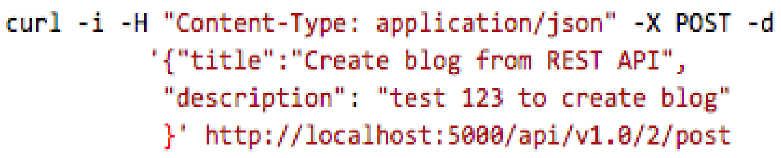
Figure 14. Posts made by user

Following this, demonstration of other HTTP requests is performed using curl command. It is possible to make a post from the API to be sent to the webserver. Usually, HTTP’s POST method handles this. Here is the function, which does this in app/views.py.



```
@app.route('/api/v1.0/<u_id>/post', methods=['POST'])
def create_post(u_id):
        links=[]
        links.append({"rel":"profile", "href":"http://localhost:5000/api/v1.0/user/"+u_id})
        links.append({"rel":"all posts", "href":"http://localhost:5000/api/v1.0/user/"+u_id+"/posts"})
        if not request.json or not 'title' in request.json\
                or not 'description' in request.json:
                        abort(400)
        post = Post()
        post.title = request.json['title']
        post.description= request.json['description']
        post.author = u_id
        db.session.add(post)
        db.session.commit()
        links.append({"rel":"created post", "href":"http://localhost:5000/api/v1.0/"+u_id+ "/post/"+post.id})

        return jsonify({"status": "post created successfully", "links": links})
```

Then this function could be run using curl command from the terminal. This command creates a new post in the database. Below is a sample command that performs this.



```
curl -i -H "Content-Type: application/json" -X POST -d
              '{"title":"Create blog from REST API",
               "description": "test 123 to create blog"
               }' http://localhost:5000/api/v1.0/2/post
```

This returns a message with HTTP status code. Thus, by this message a client could easily know what the server is responding. This message often uses HTTP status codes as a standard. For instance, server response below shows HTTP status 200 ok. Hence, it refers that the request is successful.

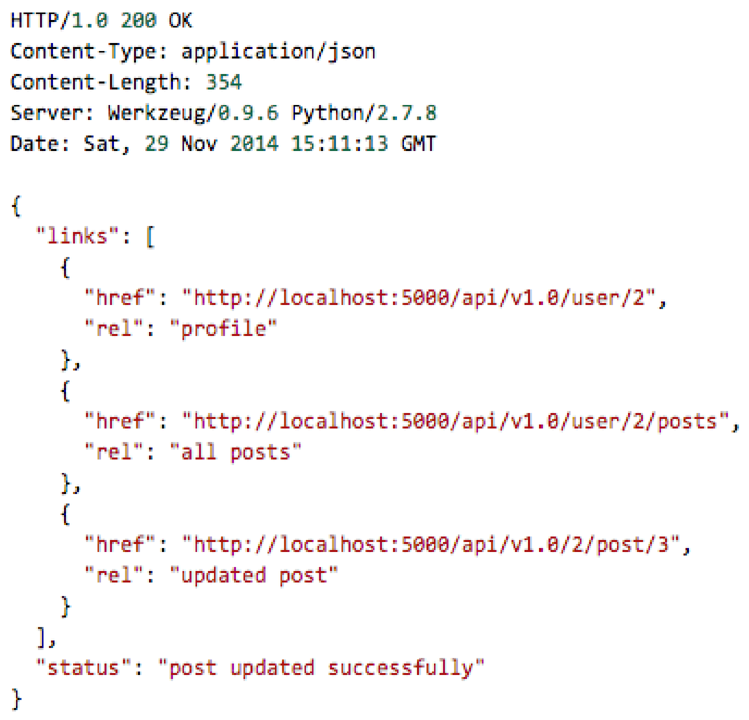
```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 354
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sat, 29 Nov 2014 15:26:28 GMT

{
  "links": [
    {
      "href": "http://localhost:5000/api/v1.0/user/2",
      "rel": "profile"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/2/posts",
      "rel": "all posts"
    },
    {
      "href": "http://localhost:5000/api/v1.0/2/post/2",
      "rel": "created post"
    }
  ],
  "status": "post created successfully"
}
```

Then, the post could be updated the same way. At this point the only change is that PUT method is used instead of POST. The code snippet below shows the update function in views.py file. Additionally, the curl command is presented to send PUT request to the server, which then updates the resource from the data source, in this case the database.

```
@app.route('/api/v1.0/<u_id>/post/<p_id>', methods=['PUT'])
def edit_post(u_id, p_id):
        links=[]
        links.append({"rel":"profile", "href":"http://localhost:5000/api/v1.0/user/"+u_id})
        links.append({"rel":"all posts", "href":"http://localhost:5000/api/v1.0/user/"+u_id+"/posts"})
        if not request.json or not 'title' in request.json\
                or not 'description' in request.json:
                        abort(400)
        post = Post.query.filter(Post.id == p_id).first()
        post.title = request.json['title']
        post.description= request.json['description']
        post.author = u_id
        db.session.add(post)
        db.session.commit()
        links.append({"rel":"updated post", "href":"http://localhost:5000/api/v1.0/"+u_id+ "/post/"+p_id})

        return jsonify({"status": "post updated successfully", "links": links})
```

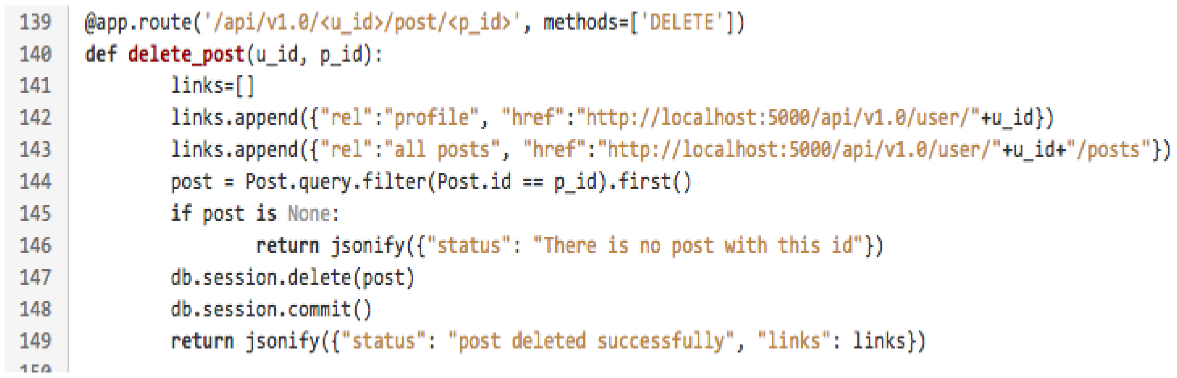
```
curl -i -H "Content-Type: application/json" -X PUT -d
          '{"title":"Update blog post from REST API",
           "description": "This is test update"
           }' http://localhost:5000/api/v1.0/2/post/3
```

Likewise, the server responds the same way as POST method. It includes HTTP status code for notifying the client. Furthermore, the developer could add more descriptive message. Hence, the requests are not always successful, more information are needed other than HTTP status code.

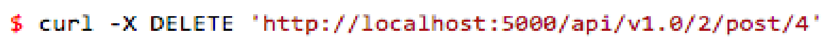


```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 354
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sat, 29 Nov 2014 15:11:13 GMT

{
  "links": [
    {
      "href": "http://localhost:5000/api/v1.0/user/2",
      "rel": "profile"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/2/posts",
      "rel": "all posts"
    },
    {
      "href": "http://localhost:5000/api/v1.0/2/post/3",
      "rel": "updated post"
    }
  ],
  "status": "post updated successfully"
}
```

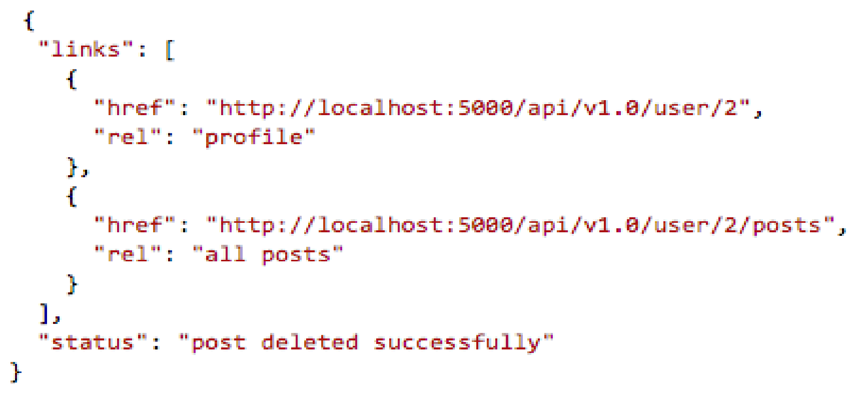
Finally, DELETE method is practiced easily the same way. It is possible to delete user’s post with a single line curl command. In the same way as above, python function is added for the app route to delete a record from the data server. Thus, the function, the curl command and its corresponding response are shown below.



```
@app.route('/api/v1.0/<u_id>/post/<p_id>', methods=['DELETE'])
def delete_post(u_id, p_id):
        links=[]
        links.append({"rel":"profile", "href":"http://localhost:5000/api/v1.0/user/"+u_id})
        links.append({"rel":"all posts", "href":"http://localhost:5000/api/v1.0/user/"+u_id+"/posts"})
        post = Post.query.filter(Post.id == p_id).first()
        if post is None:
                return jsonify({"status": "There is no post with this id"})
        db.session.delete(post)
        db.session.commit()
        return jsonify({"status": "post deleted successfully", "links": links})
```



```
$ curl -X DELETE 'http://localhost:5000/api/v1.0/2/post/4'
```



```
{
  "links": [
    {
      "href": "http://localhost:5000/api/v1.0/user/2",
      "rel": "profile"
    },
    {
      "href": "http://localhost:5000/api/v1.0/user/2/posts",
      "rel": "all posts"
    }
  ],
  "status": "post deleted successfully"
}
```

5 DISCUSSION AND CONCLUSION

Web services are used as a means of communication between various systems. This report document introduced REST starting from its fundamental architectural definition to its API level implementation. Majorly, it focused on describing how RESTful web services work both from theoretical and practical perspective. Moreover, it showed service-based communication, specifically SOAP and clearly demonstrated how SOAP messages are heavy and complicated. Hence, the author efforts to document a full package guide for REST API.

Furthermore, the above chapters clearly showed, REST gives rest for developers. It rediscovered the way systems communicate each other. For that reason, sharing resources through different technologies and platforms was being simple. Additionally, clients could easily know the transition of states from the entries of a resource. Thus, additional descriptive language layer is avoided from communicable resource. Moreover, RESTful web services perform well as the theoretical principles suggests. It uses HTTP status codes and JSON format data to transfer resources. As a result, it could easily be consumed by the client, which enables it to fulfill all the theoretical promises it made. Finally, this thesis report concludes by suggesting web developers to adopt RESTful web services as a standard for their systems' API.

REFERENCES

Alexander, C. Ishikawa, S. Silverstien, M. Jacobson, M. Fiksdahl-King, I. Angel, S. 1977. A pattern Language: Towns, Buildings, Construction. New York: Oxford University Press.

Bicking, I. 2014. Introduction. Referenced 17.7.2014. http://virtualenv.readthedocs.org/en/latest/virtualenv.html.

Bechara, G. 2009. Oracle Network Technology. Referenced 4.3.2014 http://www.oracle.com/technetwork/articles/bechara-reusable-service-087796.html.

Erl, T. 2009. Service-Oriented Architecture: Concepts, Technology, and Design. Indiana : Prentice Hall.

Fielding, R. T. 2000. Chapter 5: Architectural Styles and the Design of Network-based Software Architectures. University of California. Fielding's Doctoral Dissertation.

Fielding, R. T. 2008. REST APIs must be hypertext-driven. Referenced 24.3.2014 http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

Gamma, E. Helm, R. Johnson, R. Vlissides, J. 1994. Gang of Four. Design Patterns: Elements of Reusable Object-Oriented Software. 1st edition. New York: Addison-Wesley Professional.

Grinberg, M. 2012. The Flask Mega-Tutorial. Referenced 27.10.2014. http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world.

Gulzar, N. 2003. Practical J2EE Application Architecture. 1st edition. New York: McGraw-Hill Osborne Media.

Linthicum, D. 1999. Chapter 1: Service Oriented Architecture (SOA). Referenced 6.3.2014. http://msdn.microsoft.com/en-us/library/bb833022.aspx.

Mulloy, B. 2013a. API Design: Ruminating Over REST. Referenced 6.9.2014. https://blog.apigee.com/detail/api_design_ruminating_over_rest.

– 2013b. API Design: Harnessing HATEOAS, Part 2. Referenced 6.9.2014. https://blog.apigee.com/detail/api_design_harnessing_hateoas_part_2.

Oracle Corporation. 2001. Simple Object Access Protocol Overview. Referenced 10.3.2014. http://docs.oracle.com/cd/A97335_02/integrate.102/a90297/overview.htm.

Reese, G. 2012. The REST API Design Handbook. 1st edition. Amazon Kindle Edition.

Ronacher, A. 2011. Flask-SQLAlchemy. Referenced 14.10.2014. https://pythonhosted.org/Flask-SQLAlchemy/.

Rozanski, N. & Woods, E. 2011. Software Systems Architecture: Working with Stakeholders using viewpoints and perspectives. 2nd edition. Massachusetts: Addison Wesley.

Software Engineering Institute 2013. Referenced 4.3.2014 http://www.sei.cmu.edu/architecture/start/glossary/moderndefs.cfm.

SQLAlchemy. 2014. Object Relational Tutorial. Referenced 14.10.2014 http://docs.sqlalchemy.org/en/rel_0_9/orm/tutorial.html.

W3C Working Group Note. 2004. Web Services Glossary. Referenced 3.3.2014 http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice.