RESTful Web API Case Study

horizontal line

This document briefly discusses what a RESTful Web API is and some of the motivations for using them. Example APIs are described using the HTTP verbs GET, POST, PUT, DELETE. These APIs were written in C# using Visual Studio 2017, .NET and MVC.

What is a RESTful Web API? A quick search of the internet will show that there are numerous discussions, videos, tutorials, seminars and other media discussing this topic. After spending only a few hours researching this topic, it became abundantly clear that there is a large volume of information which is incorrect and unnecessarily complicated.

Discussions of REST and comparisions to other technologies such as SOAP have been raging since the Phd dissertation of Roy Thomas Fielding was published in year 2000. His dissertation defines a way to describe and evaluate network-based application software. Here is a link to his paper:

https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\_dissertation.pdf

In his dissertation, after an extensive description of the history of the internet, the worl wide web and various software architectures, he defined and proposed REST (Representational State Transfer) as an architecture which, quoting from his paper, emphasizes the following:

*REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.*

Returning to the question, “What is a RESTful Web API?”, clearly to profoundly understand this issue requires a great deal of study and experience. However, here I will share what I have learned after researching the issue for a pair of hours and then spending a day getting some APIs working and developing a test program written in python 3.5 using Jupyter Notebook.

Before jumping into describing the RESTful APIs, let’s discuss a few RESTful principles. Keep in mind that REST is an architecture, not a standard. Thus, comparing REST against other architectures can be tense. Its similar to comparing building architecture styles. Is it easy to accurately differentiate between homes which are Mediterranean versus Southwestern versus Spanish architectures?

To keep the discussion concrete, we will be discussing how to program APIs which allow us to implement an “address book” accessible on the web. It’s a very simple address book, but keep in mind that we want the solution to be scalable. For example, perhaps every creature on earth could be listed in the book and perhaps billions of users might access it frequently. I’m afraid that a single computer with a single network connection is not going to tbe able to handle the traffic. The solution must be ***scalable.***

So, let’s jump in. Here are some general principles of REST :

1. Everything is a URI.

Rather than treating documents, videos, audio, etc. differently, we should allow them to be accessed by a URI, eventhough internally they may be completely unrelated. For our address book, we will allow our address book to be accessed via a URI. Here is an example URI: http://www.mysite.com/address

1. Every Resource is identified by a unique identifier

As much as practical, we’d like to expose parts of our data via a URI. For our address book, we will allow every row in the address book to be accessed via a URI. If each row has a key, and if one row has key=14, then if the address book is located at http:://www.mysite.com/address, we can refer to this row as follows: URI = http:://www.mysite.com/address/14

1. Use simple/uniform interfaces

For our Address book project we will use simple HTTP 1.1 verbs GET/PUT/POST/DELETE according to the RESTful guidelines (more on this later).

1. Requests/Responses use “representations” of data (JSON or XML generally)

Rather than using proprietary, specialized, binary data structures, RESTful APIs use XML and more commonly, JSON data structures.

1. Be Stateless

Communication on the web is unreliable. Machines break, computers crash, messages are lost. Also, there are no garantees about order of arrival or rate of arrival of messages. Multiple users may be trying to do the same thing at the same time. In as much as possible each transaction should not depend on history. One must be aware that no matter how quickly two transactions are made, an infinite number of other users transactions can occur between them. The second transacation cannot depend on the first.

Let’s discuss how the APIs will permit access and changes to the address book. It is assumed that the reader understands the basics of HTTP with regard to the format of the HTTP verbs.

Here are the transactions that the APIs implement:

READ a single address row

READ entire address book (all rows)

CREATE a new address row

UPDATE a single address

DELETE a single address

The following table indicates how the verbs implement the address book transactions. “/api/address” represents the suffix of the URI of the address book.

| **API** | **Description** | **Request body** | **Response body** |
| --- | --- | --- | --- |
| GET /api/address | Get all addresses | None | List of Addresses |
| GET /api/address/<key> | Get an address by <key> | None | Single address |
| POST /api/address | Create a new address | Single address | Server creates <key> and returns address and <key> |
| PUT /api/address/<key> | Update an existing address | Single address | None |
| DELETE /api/address/<key> | Delete an address | None | None |

Let’s discuss some aspects of each transaction:

CREATE – The action starts here.

We do a POST to /api/address. The address row information is in our HTTP message. The server returns a unique <key> for this row. If we want to refer to this row again, we can refer to it directly using /api/address/<key>.

Once stored on the server, each row of our address book has a unique <key>; however, the <key> is assigned by the server, not by us. This is one of the RESTful principles – the server controls its resources. There are many reasons for this, but one important one is scalability. Allowing the server to assign the <key> and also making it visible in the URI allows the server(s) and intermediary computers, caches, etc to route and process the transaction in the manner that best optimizes system requirements. For example, the address book may grow so large that it needs to be spread among 20 servers. Or, there may be so much network traffic for the address book that servers, gateways, caches, intermediate computers, etc. may be reconfigured to optimize performance. This can be done without changes to the APIs or to client software because the <key> is visible in the URI and the server assigns it, not the client.

READ single address row – To read a row, we do a GET with a URI of /api/address/<key>. The row is returned in the HTTP response.

READ entire address book (all rows) – To read all the rows, we do a GET with a URI of /api/address. Again, the returned data is in the HTTP response.

UPDATE a single address –We do a PUT to /api/address/<key>. A PUT is a REPLACE of a row. Hence, if we do the same UPDATE many times, the result is the same. It is an ***idempotent*** operation.

DELETE a single address – We do a DELETE to /api/address/<key>.

The APIs to implement an address book and test programs testAddr.py (python), testAddr.ipynb (Jupyter Notebook) can be found at:

<https://williamgroves-hotmail.visualstudio.com/_git/AddrApi>

Playing with testAddr.ipynb is an easy way to learn about the APIs. You can also use POSTMAN, curl or many other ways to try them.

These APIs are to illustrate some RESTful principles. To implement a production system one needs to consider the following which are beyond the scope of my Sunday project!

1. Key – Each row has a unique key. The server decides the value of the key. It could be a hash of various fields in the row or it could be a unique number (phone, etc). It could also include geographical location to help with load balancing. This is beyond the scope of the APIs. Appropriate choice of the <key> can prevent redundant information from being entered.
2. Atomicity of the fields of the address rows. The PUT (update) causes a consistent set of fields to arrive at the server. It is the servers responsibility to atomically update the row such that the internal fields are consistent.
3. Validation of fields. The APIs do not do any validity checking of the data put into the rows of the address book.