Bryan Barnard
DePaul University
March 20, 2014

RESTful API Design and Implementation

Independent Study Winter Quarter 2014
DePaul University
Advising Faculty: Massimo DiPierro
Project Repository: https://github.com/bryanbarnard/ISWQ2014

Bryan Barnard
DePaul University
March 20, 2014

Table of Contents

Bryan Barnard
DePaul University
March 20, 2014

1. Introduction

APIs and more specifically Web APIs are everywhere these days; from the backend service your mobile device is talking to at this moment to check the weather or the status of your flight to the AJAX call your Web Browser is making to validate the address you entered on a checkout form when shopping online. If you are not interacting with a Web API explicitly as a developer you need only inspect the network traffic between any internet connected device and the internet to quickly realize how dependent our current systems are on Web APIs and at the same time how powerful Web APIs can be for building distributed systems. Browsing Netflix on your TV, you're (computer/device) using a Web API. Paying for your coffee odds are the cash register or your phone is making a call to a Web API to process the payment.

Web APIs by which I mean APIs that are accessible via either the public Internet or private networks and rely on the HTTP Protocol for communication of messages (request/response) are not a new concept. Developers have been using Web Services for quite some time to build distributed systems. Many networked systems have been built using technologies such as SOAP, RPC and Corba.

Recently another pattern has become popular for building networked systems. REST; which stands for Representational State Transfer is arguably being used as the default style for building Web based networked systems. What is REST and what is involved in designing and implementing a RESTful API? These are the overarching questions that have motivated me to pursue an Independent Study this quarter. During this quarter I have worked to answer these questions through study and research on the topic as well as by designing and implementing a RESTful API in multiple languages and Web frameworks.

1.2 What is REST?

REST stands for Representation State Transfer as defined in Roy Fielding's Ph.D. dissertation[1]. Fielding describes REST as an architectural style for building distributed hypermedia systems. In describing REST Fielding specifies a set of constraints that he refers to as the "architectural properties" of the Web. These constraints (listed below) roughly describe the architecture and properties that make up the Web that we are familiar with today. This is no coincidence given that Fielding worked on the definition and development of both version 1.0 and 1.1 of the HTTP Protocol.

[1] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation. University of California, Irvine, 2000.

Fielding Constraints

- Client – Server
- Stateless
- Cache
- Layered System
- Code on demand (optional)
- Uniform Interface

Briefly summarizing the constraints is helpful in gaining an understanding of the properties of the Web and how they are applied to Web APIs. The Client-Server constraint suggests a separation of concern between the client and server components and their interaction.

The Stateless constraint applies specifically to the interactions between the client and server and specifies that their interactions must be such each request/response contains all necessary information for that component to understand the message.

The Cache constraint requires that each request / response message indicates it's ability to be cached. This aids in efficiency by allowing some data to be cached and not need to be requested by the client repeatedly when the underlying data being requested has not changed.

The Layered System constraint builds upon the client server constraint suggesting that the various subcomponents should separate and limit their concern so that they can focus exclusively on the area of concern and in doing so increase scalability.

The optional Code-On-Demand Constraint allows clients to download or receive code at runtime. This is an optional constraint and can add complexity to an implementation mainly in the form of securing the sources of code.

The Uniform Interface constraint is described by Fielding as a "central feature that distinguishes the REST architectural style" from others. This constraint describes the way in which systems should interact with data and are broken down into the following:

Uniform Interface Constraints

- Identification of resources
- Manipulation of resources through representations
- Self-Descriptive messages
- Hypermedia as the engine of application state

Simply put the Identification of resources constraint states that a URI identifies a resource and one should be able to rely on a URI pointing to that resource even if it's state changes.

The manipulation of resources through representations constraint implies that actions taken on resources are done via representations. Representations represent a resource via a list of attributes rendered as a "sequence of bytes," that can be transferred over a network and understood by a client. This constraint also limits the actions that can be taken on a resource to those defined by the application protocol. This constraint also asserts that clients should use the representation of a resource as stand in for the actual representation. Applying this constraint to Web Services and APIs clients and servers manipulate resources by sending representations of those resources back and forth using HTTP methods.

The Self-Descriptive messages constraint specifies that messages should be self-descriptive, meaning that each message should contain enough information via message content, standard media types and protocol methods to convey semantics of the message and allow the recipient to understand it. This means that a message should in and of itself contain enough information for a client to interpret and act upon it without any additional documentation. This idea is a vast departure from other architectures mainly SOAP which relied on a WSDL or some other documentation for consumers to be able to parse and understand the contents of the messages.

The Hypermedia constraint as defined by Fielding is a bit less clear than the others however Mike Amundsen and Leonard Richardson simplify it in their book "Restful Web APIs"[2] as the following:

> 1. All application state is kept on the client side. Changes to the application state are the client's responsibility.
> 2. The client can only change its application state by making an HTTP request and processing the response.
> 3. A client knows which requests it can make by looking at the hypermedia controls in the representation it's received thus far.
> 4. Hypermedia controls are therefore the driving force behind changes to application state.

My review of Fielding's constraints is by no means exhaustive it is merely meant to provide background and context for REST and the concepts that underpin the application and implementation of RESTful APIs.

1.3 What makes an API RESTful?

That being said what makes an API RESTful? It is important to keep in mind that Fielding's definition of REST as an architectural style was not directed at API design but rather as guidelines for building networked systems that would scale. It is also no coincidence that a system built in adherence to these guidelines would look and feel very much like the Internet that we know and use today given that Fielding was heavily involved in the design and formalization of the HTTP 1.0 and 1.1 standards.

Having begun to understand this via my study it started to make sense to me more and more why it seemed like the characteristics of RESTful APIs seemed to align closely with the design of the Internet, as we know it. Back to the question what makes an API RESTful? Technically an API that adheres to Fielding's constraints is RESTful however I think it helps to summarize how this is often applied to API's and the best way to do that is to list some of the common characteristics of RESTful APIs.

- Resources are interacted with as representations. In application this means that it is understood that servers and clients are at any time only communicating via HTTP messages and these messages provide a snapshot of a resource at the moment the message is returned. There is not a reference

[2] Mike Amundsen and Leonard Richardson, *RESTful Web APIs,* O'Reilly Media Inc. *2013*

or pointer the client maintains to an object in memory on the server, and changes the client makes to the representation it received will have no effect on the resource maintained by the server. If a client wishes to read or modify a resource the server is managing the only way it can do so is via these representations and the interface the server provides.

- URL's point to resources and it is implied that making a HTTP GET request to a URL will return a representation of the resource or a valid HTTP Status code and potentially an error message informing the requester that the resource is not available, has been moved or there is a server error.

- Interactions with resources are mapped to valid HTTP methods that are most appropriate. For example the most common being GET for query/read, POST for creating a resource, PUT for update a resource, and DELETE for deleting a resource.

- Addressing interactions it is also important to adhere to the expected interaction behavior. GET should always be a non-mutating safe interaction. POST, PUT and DELETE are non-safe interactions as they result in possible mutation of the resource. PUT should be idempotent in that regardless of the number of requests made with the contents of PUT as long as the contents of the request do not change the result on the resource should be the same. DELETE. This is often referred to as protocol semantics of an API.

- Media-Type should contribute to making messages self-describing. Well-defined standardized Media-Types such as Collection+JSON[3], SIREN[4] and HAL[5] to name a few not only provide constraint and form to how the message will be rendered but also carry with them some constraints that effect application semantics. In doing so they also help convey information to the client regarding the semantics of the application and how they are mapped to the semantics of the protocol being used (HTTP).

- Messages are self-descriptive and carry with them not only a representation of the resource but additional information the client can use to understand the state of the resource, recognize potential actions that are available to them on that resource and understand the appropriate format (message format, endpoint, etc.). This is often addressed with the inclusion of Hypermedia controls in messages.

---

3 http://amundsen.com/media-types/collection/

4 http://stateless.co/hal_specification.html

5 https://github.com/kevinswiber/siren

- A goal of a RESTful API should be to allow the consumer to understand how to interact with the resources it provides an interface for without requiring additional documentation in the form of PDF's, Web Sites, Help Text found on other sites that is specific only to that specific API. It is not wrong to have this additional information available but it should not be required.

- Standard HTTP methods, headers and status codes are employed appropriately to convey information regarding resources.

- Caching by the client is supported by the use of etag and last modified headers to allow clients to make conditional requests and operate efficiently.

- Hypermedia controls are included in messages to bolster the self-descriptiveness of messages.

1.3 What is Hypermedia?

I found the term Hypermedia as it relates to APIs very difficult to understand and even harder to define. The best explanations I've found and those that most contributed the most to my current understanding are listed below. Both of these quotes are taken from 'RESTful Web APIs' by Mike Amundsen and Leonard Richardson.

"Hypermedia connects resources to each other, and describes their capabilities in machine-readable ways. Properly used, hypermedia can solve—or at least mitigate—the usability and stability problems found in today's web APIs."[6]

"Hypermedia is a way for the server to tell the client what HTTP requests the client might want to make in the future. It's a menu, provided by the server, from which the client is free to choose."[7]

Given this definition what are some examples of hypermedia in APIs? Hyperlinks or links are the most commonly used hypermedia control in APIs. Simple anchor tags (<a href="">) paired with a 'rel' that describes the relationship between the current resource representation and the resource residing at endpoint of that link for which you can get a representation by issuing a HTTP GET request provide a great example of Hypermedia. A link connects resources, is machine readable, and can describe a capability to return a representation of the resource addressed by its URL. Links are the most common hypermedia control in APIs their use in APIs and specifically representations go a long way to making messages self-descriptive.

[6] RESTful Web APIs, Leonard Richardson & Mike Amundsen, O'Reilly Media 2013, Version 1, pg. 45

[7] RESTful Web APIs, Leonard Richardson & Mike Amundsen, O'Reilly Media 2013, Version 1, pg. 45

Media-Types that support hypermedia often include multiple hypermedia controls in the form of links, forms, POST templates, etc. These controls will be specified by the media-type. Collection+JSON for example support links, write and query templates.

2. Plan

Having researched the REST architecture style as well as commentary by API designers regarding how to apply REST to the design of APIs I came up with a plan for designing and building a simple RESTful API. The goal was to formulate a plan that would allow me to work through the design process gaining experience designing and implementing an API that adhered to the REST constraints when applicable as well as providing me an opportunity to familiarize myself with the frameworks available for implementing an API in NodeJS as well as Python.

2.1 Designing an API

The first decision I had to make regarding my API design was choosing subject matter. This would define the content that the API would provide access to and have implications on the application semantics of my API. Much of the work I have done in the past on APIs and Web Services has dealt with collections of one sort or another and it's a common use case for an API to provide access to a collection of data. For my subject matter I wanted something that would conform to a collection and could also make since having only a limited number of attributes. My chosen subject matter was a movie collection. This would provide a simple subject matter with well-known attributes (i.e. Title, Genre) that I could design an API for that would allow for basic interaction.

Having chosen my subject and establishing that my API would provide basic collection interactions with this content I set out to define what operations would be necessary to support those interactions and how I would represent these Movie resources as items in a collection. The natural operations on a collection of items are create, read, update, delete and search. My API would need to provide these common operations to consumers of the API. The collection pattern also seems to lend itself to the most common implementations of REST APIs as well and allows for a mapping of HTTP methods to these operations.

For my API movies were the resources that would be manipulated via representations. The representations being HTTP messages passed between the API (server) and consumer (client) including appropriate HTTP methods, headers and entity's to provide the desired operations.

2.2 Media-Type

The REST self-descriptive constraint on messages specifies that standard method and media types should be used for communications between client and server. Adhering to this constraint requires that in designing an API the designer either choose an established media-type or create his or her own media-type. If the latter is chosen than necessary steps should be taken to publish this Media-Type so that it may be known easily referred to for clients to understand and utilize for requests and responses. It is often assumed that a RESTful API must use JSON or that any API that returns JSON is RESTful. This is not the case, and much to the contrary API's that return XML and other standard media-types may be implemented in a RESTful style.

For my API I wanted a media-type that represented collections well and provided a standard for implementing hypermedia controls. My preference was also for JSON as I find it to be human readable as well as easy to parse and interact with programmatically. It was also important to me that a media-type be well documented and provide clear specifications and examples. This would benefit me as an implementer as well as any potential consumer of my API as documentation, examples and potentially tooling would already exist to support this media-type.

For these reason I chose Collection+JSON[8] as the media type I would use to format messages for requests and responses to my API representing Movie resources. "Collection+JSON is a JSON-based read/write hypermedia-type designed to support management and querying of simple collections" [9]. In researching media types I considered both the HAL[10] and Siren[11] media types. Both are implemented on top of JSON and provide support for hypermedia types, however they seem to be a bit less standardized than Collection+JSON at this point and have less supporting documentation.

8 http://amundsen.com/media-types/collection/

9 Collection+JSON Media Type, Mike Amundsen, IANA.org, http://www.iana.org/assignments/media-types/application/vnd.collection+json

10 http://stateless.co/hal_specification.html

11 https://github.com/kevinswiber/siren

A Collection+JSON response in its most minimal representation:

```
{
  "collection": {
    "version" : "1.0",
    "href" : "http://example.com/movies",
    "items" : [],
    "links" : [],
    "queries" : [],
    "template" : {},
    "error" : {}
  }
}
```

2.3 Design

Having chosen the subject matter, type of operations I wanted to provide and a media-type that supported this I set about defining the application semantics of my API and mapping them to the corresponding protocol semantics of HTTP. Since I had chosen a media-type (arguably part of the design) that supported the collection pattern and was quite opinionated regarding its implementation I had a set of constraints within which to work in.

Movie records are the resources of my API and would be represented by messages formatted in accordance with the media-type. Common collection operations create, read, update, delete and query would be supported on Movie resources.

I chose a simple URL structure that I believe is common in providing APIs at endpoints within domains that also house other applications. For my API the base URL would be http://{host}/api/. This URL would return what is commonly referred to as a 'billboard' for the API, which included links to the movies collection, an ALPS[12] as well as a link where additional more human friendly documentation and sample requests could be found.

Movie resources are accessible via http://{host}/api/movies with collection operations being mapped to standard HTTP methods as follows.

HTTP Method	URL	Operation
GET	/api/movies	Get Movies Collection.

[12] http://alps.io

GET	/api/movies?{offset}&{limit}	Get Movies Specifying Paging.
GET	/api/movies?{name}	Search Movies Collection matching name provided.
GET	/api/movies/{id}	Get Movie Item by id.
POST	/api/movies	Create (append) a new Movie resource to the Collection.
PUT	/api/movies/{id}	Update an existing Movie Resource.
DELETE	/api/movies/{id}	Delete an existing Movie Resource.

Movie resources have the following attributes as listed in the table below. Subset of those defined at http://schema.org/Movie

Property Name	Friendly Name	Schema.org Type
name	Movie Name	http://schema.org/name
description	Movie Description	http://schema.org/description
datePublished	Date Published	http://schema.org/datePublished
about	Alt Description of Movie	http://schema.org/about
genre	Movie Genre	http://schema.org/genre
version	Version of this release	http://schema.org/version
timeRequired	Duration of Movie	http://schema.org/timeRequired
contentRating	Movie Rating	http://schema.org/contentRating
director	Movie Director	http://schema.org/director
created_on	DateTime Resource Created	n/a
updated_on	DateTime Resource Updated	n/a
sys_id	Movie Id	n/a

* These are also defined along with additional details of the API in xml as a ALPS profile at: https://github.com/bryanbarnard/ISWQ2014/blob/master/docs/movies.xml

3. Implementation

As part of my research into REST and designing RESTful APIs I also wanted to gain experience implementing them in various languages and frameworks. I chose NodeJS and Python for my implementations. My reasoning behind this decision is that both NodeJS and Python provide the ability to host a Web Service without requiring a separate Web Server in addition they're libraries provide access directly to the request and response objects. This allows the developer to setup a rather lightweight implementation without many dependencies.

An API application in its simplest form is comprised of a few basic components. An HTTP request handler that will listen on a port for an incoming request and upon receipt of that request pass the request to your code to inspect and take action on. A router that inspects these requests and subsequently routes the request to a function you have specified to process that request. Functions that you have specified to handle said request and which modify or build a response object based on the processing of the request. Both of my solutions implement these components. For this project I tried to keep my implementations as simple as possible and only implement necessary functionality.

3.1 NodeJS Implementation

My NodeJS implementation of the movies API uses only the standard 'HTTP' NodeJS libraries to access the request and response objects, allowing me to directly handle and inspect incoming requests. The application will listen for requests on a specified port and upon receipt of the request attempt to parse the components of the request (URL, method, headers, entity) and match the request against a specific route. Based on the requests URL matching various regex patterns the request will be dispatched to a callback function(s). The callback function will process the request applying any necessary application logic and build an appropriate response that will be returned to the client as a HTTP response.

I found implementing an API in NodeJS to be very straightforward. NodeJS provides direct access to both the request and response objects with standard libraries for common URL and path processing.

I chose not to use an existing Web Framework such as ExpressJS[13] or Restify[14] because I didn't see a need to add a layer by using a framework given the level of access NodeJS provides out of the box for handling HTTP requests and I wanted to

[13] http://expressjs.com/
[14] https://github.com/mcavage/node-restify

take this opportunity to familiarize myself with the standard libraries. In the limited time I spent researching these frameworks they both look to be interesting projects that include some nice routing functionality and helpers to make routing simpler. ExpressJS seems to be the most commonly used and popular for building Web Applications and Web APIs in NodeJS at this time.

Working with NodeJS and using a JSON based media-type allows you to NodeJS native JSON processing. This eliminates the need for extra code or libraries to parse and transform JSON strings to dictionaries or other object types.

3.2 Python Implementation

For my implementation of the Web API in Python I chose to use the Micro Web Framework Bottle[15]. I made this choice because I have limited experience working with Python and although I looked at writing my API directly on top of WSGI I was concerned it may add unnecessary time to my development. Instead I chose to use the most lightweight Python Web framework that would provide me with convenient helper functions for handling and routing requests, parsing URLs, and building responses.

Bottle provides enough of the common functionality needed to build an API without getting in the way or being overly opinionated. The only small complaint I have regarding working with Bottle is that it attempts to default the content-type when processing requests and response containing JSON to “application/json”, this can be turned off with configuration.

The structure of my application using Python with Bottle was very similar to my NodeJS implementation. Essentially the application receives a request and inspects a HTTP request comparing its URL and method to a set of defined routes that map routes to function(s). The handler dispatches the request to a function that will then process the request, apply application logic and build an appropriate response. Bottle provides a simple decorator syntax that allows you to map routes to functions simple by decorating the function. Through this decorator syntax you can align routes based on method, simple pattern matching or even custom RegEx. This is a helpful feature that is available in most all Web Frameworks now, and is included in both of the Web Frameworks I mentioned previously for NodeJS.

[15] http://bottlepy.org/docs/dev/index.html

3.3 Persistence

Both the NodeJS and Python implementations rely on a MongoDB[16] document database for persistence. I chose MongoDB for its ease of setup and use. The focus of this project was not on database technology and so I wanted something that was simple to setup and easy to access. Well-written and documented drivers exist for both Python and NodeJS to work with MongoDB as well as lightweight ODM's. In my NodeJS implementation I use the MongooseJS[17] ODM and in the Python implementation I use MongoEngine[18]. Both of these packages were lightweight, open source, and provide a similar API for interacting with MongoDB.

4. Conclusion

Through my work on this independent study I have vastly increased my understanding of REST and what it means for an API to be RESTful. My research has included reviewing formal materials on the subject such as published books and papers, as well as seeking out communities and forums (Google groups, blogs, presentations) on the Internet where many of the new design patterns for APIs are being actively discussed and formulated.

In addition to researching the topic having the opportunity to implement a RESTful API was key in furthering my understanding. Without working through an implementation (or two) I would have not gained as thorough and understanding of what hypermedia is and the important role it plays in RESTful APIs. I believe implementing this API in multiple languages / frameworks provided an enhanced perspective on the components that make that make up a Web Framework and the common patterns employed for handling request and response objects in Web Frameworks. I expect this perspective will beneficial for future development endeavors.

My regular meetings with my faculty advisor, Massimo DiPierro, were an important part of my independent study. These meetings gave me an opportunity to discuss and explain the concepts I was learning which I believe helped me to formalize my understanding. At the same time having frequent check ins regarding my work and progress on implementing my API helped to keep me on schedule and progressing at a regular pace.

4.2 What's next?

I've thoroughly enjoyed the time I've spend researching REST and implementing RESTful APIs as part of my independent study. I'm thankful for having the

[16] https://www.mongodb.org/
[17] http://mongoosejs.com/
[18] http://mongoengine.org/

opportunity to pursue this topic and am even more interested in pursuing further experience and knowledge than I was prior to this quarter. I have some further enhancements I plan to make on the APIs I've implemented for this class and would also like to implement the same API in a few other frameworks including but not limited to Ruby, Java and C#.

Appendix A – API Sample HTTP Request and Response Messages

Appendix B – References utilized during my independent study

Books

- RESTful Web APIs, By Leonard Richardson, Mike Amundsen, Sam Ruby. Publisher: O'Reilly Media Released: September 2013
- RESTful Web Services, By Leonard Richardson, Sam Ruby, Publisher: O'Reilly Media Released: May 2007
- Building Hypermedia APIs with HTML5 and Node By Mike Amundsen, Publisher: O'Reilly Media Released: November 2011
- Architectural Styles and the Design of Network-based Software Architectures Dissertation, by Roy Thomas Fielding, 2000
- RESTful Web Services Cookbook, By Subbu Allamaraju, Publisher: O'Reilly Media / Yahoo Press, Released: February 2010

Online Resources

- Mike Amundsen Blog, http://amundsen.com/blog/
- Google Groups - Collection JSON https://groups.google.com/forum/#!forum/collectionjson
- Google Groups - API Craft https://groups.google.com/forum/#!forum/api-craft