Andrew Havens *Ruby Developer*

Beginners guide to creating a REST API

13 SEPTEMBER 2012

If you're reading this, you've probably heard the terms **API** and **REST** thrown around and you're starting to wonder what the fuss is all about. Maybe you already know a little bit, but don't know how to get started. In this guide, I will explain the basics of REST and how to get started building an API (including authentication) for your application.

What is an API?

The term API stands for Application Programming Interface. The term can be used to describe the features of a library, or how to interact with it. Your favorite library may have "API Documentation" which documents which functions are available, how you call them, which arguments are required, etc.

However, these days, when people refer to an API they are most likely referring to an HTTP API, which can be a way of sharing application data over the internet. For example, Twitter has an API that allows you to request tweets in a format that makes it easy to import into your own application. This is the true power of HTTP APIs, being able to "mashup" data from multiple applications into your own hybrid application, or create an application which enhances the experience of using someone else's application.

For example, let's say we have an application that allows you to view, create, edit, and delete widgets. We could create an HTTP API that allows you to perform these functions:

```
 http://example.com/view_widgets
http://example.com/create_new_widget?name=Widgetizer
http://example.com/update_widget?id=123&name=Foo
http://example.com/delete_widget?id=123
```

A problem has started to arise when everyone starts implementing their own APIs. Without a standard way of naming URLs, you always have to refer to the documentation to understand how the API works. One API might have a URL like `/view_widgets` whereas another API might use `/widgets/all`.

Don't worry, REST comes to rescue us from this mess.

What is REST?

REST stands for **Re**presentational **S**tate **T**ransfer. This is a term invented by Roy Fielding to describe a standard way of creating HTTP APIs. He noticed that the four common actions (view, create, edit, and delete) map directly to HTTP verbs that are already implemented: `GET`, `POST`, `PUT`, `DELETE`.

If you're new to HTTP, you may not be familiar with some of these verbs. So let me give a brief rundown on HTTP methods.

HTTP methods

There are technically 8 different HTTP methods:

```
 GET
POST
PUT
DELETE
OPTIONS
HEAD
TRACE
CONNECT
```

Most of the time, when you're clicking around in your browser, you are only ever using the `GET` HTTP method. `GET` is used when you are "getting" a resource from the internet. When you submit a form, you are usually using the `POST` method to "post data" back to the website. As for the other methods, some browsers don't even implement them all. However, for our uses, that doesn't matter. What matters is that we have a bunch of "verbs" to choose from which help to describe the actions we are taking. We will be using client libraries which already know how to use the different HTTP methods.

Examples of REST

Let's look at a few examples of what makes an API "RESTful". Using our widgets example again...

If we wanted to view all widgets, the URL would look like this:

```
GET http://example.com/widgets
```

Create a new widget by posting the data:

```
POST http://example.com/widgets
Data:
    name = Foobar
```

To view a single widget we "get" it by specifying that widget's id:

```
GET http://example.com/widgets/123
```

Update that widget by "putting" the new data:

```
PUT http://example.com/widgets/123
Data:
    name = New name
    color = blue
```

Delete that widget:

```
DELETE http://example.com/widgets/123
```

Anatomy of a REST URL

You might have noticed from the previous examples that REST URLs use a consistent naming scheme. When you are interacting with an API, you are almost always manipulating some sort of object. In our examples, this is a `Widget`. In REST terminology, this is called a **Resource**. The first part of the URL is always the plural form of the resource:

```
/widgets
```

This is always used when referring to this collection of resources ("list all" and "add one" actions). When you are working with a specific resource, you add the ID to the URL.

```
/widgets/123
```

This URL is used when you want to "view", "edit", or "delete" the particular resource.

Nested Resources

Let's say our `widgets` have many users associated with them. What would this URL structure look like?

List all:

```
GET /widgets/123/users
```

Add one:

```
 POST /widgets/123/users
Data:
    name = Andrew
```

Nested resources are perfectly acceptable in URLs. However, it's not a best practice to go more than two levels deep. It's not necessary because you can simply refer to those nested resources by ID rather than nesting them within their parents. For example:

```
/widgets/123/users/456/sports/789
```

...can be referenced as:

```
/users/456/sports/789
```

...or even:

```
/sports/789
```

HTTP Status Codes

Another important part of REST is responding with the correct status code for the type of request that was made. If you're new to HTTP status codes, heres a quick summary. When you make an HTTP request, the server will respond with a code which corresponds to whether or not the request was successful and how the client should proceed. There are four different levels of codes:

- 2xx = Success
- 3xx = Redirect
- 4xx = User error

- 5xx = Server error

Here's a list of the most important status codes:

Success codes:

- 200 - OK (the default)
- 201 - Created
- 202 - Accepted (often used for delete requests)

User error codes:

- 400 - Bad Request (generic user error/bad data)
- 401 - Unauthorized (this area requires you to log in)
- 404 - Not Found (bad URL)
- 405 - Method Not Allowed (wrong HTTP method)
- 409 - Conflict (i.e. trying to create the same resource with a PUT request)

API response formats

When you make an HTTP request, you can request the format that you want to receive. For example, making a request for a webpage, you want the format to be in HTML, or if you are downloading an image, the format returned should be an image. However, it's the server's responsibility to respond in the format that was requested.

JSON has quickly become the format of choice for REST APIs. It has a lightweight, readable syntax that can be easily manipulated. So when a user of our API makes a request and specifies JSON as the format they would prefer:

```
GET /widgets
Accept: application/json
```

...our API will return an array of `widgets` formatted as JSON:

```
[
  {
    id: 123,
    name: 'Simple Widget'
  },
  {
    id: 456,
    name: 'My other widget'
  }
]
```

If the user requests a format that we haven't implemented, what do we do? You can throw some type of error, but I would recommend enforcing JSON as your standard response format. It's the format that your developers will want to use. No reason to support other formats unless you already have an API which needs to be supported.

Building a REST API

Actually building a REST API is mostly outside the scope of this tutorial since it is language specific, but I will give a brief example in Ruby using a library called Sinatra:

```
require 'sinatra'
require 'JSON'
require 'widget' # our imaginary widget model

# list all
get '/widgets' do
  Widget.all.to_json
end

# view one
get '/widgets/:id' do
  widget = Widget.find(params[:id])
  return status 404 if widget.nil?
  widget.to_json
end

# create
post '/widgets' do
  widget = Widget.new(params['widget'])
  widget.save
  status 201
end

# update
put '/widgets/:id' do
  widget = Widget.find(params[:id])
  return status 404 if widget.nil?
  widget.update(params[:widget])
  widget.save
  status 202
end

delete '/widgets/:id' do
  widget = Widget.find(params[:id])
  return status 404 if widget.nil?
  widget.delete
  status 202
end
```

API authentication

In normal web applications, handling authentication is usually handled by accepting a username and password, and saving the user ID in the session. The user's browser saves a cookie with ID of the session. When the user visits a page on the site that requires authentication, the browser sends the cookie, the app looks up the session by the ID (if it hasn't expired), and since the user ID was saved in the session, the user is allowed to view the page.

With an API, using sessions to keep track of users is not necessarily the best approach. Sometimes, your users may want to access the API directly, other times the user may way to authorize another application to access the API on their behalf.

The solution to this is to use token based authentication. The user logs in with their username and password and the application responds with a unique token that the user can use for future requests. This token can be passed onto the application so that the user can revoke that token later if they choose to deny that application further access.

There is a standard way of doing this that has become very popular. It's called OAuth. Specifically, version 2 of the OAuth standard. There are a lot of great resources online for implementing OAuth so I would say that is outside the scope of this tutorial. If you are using Ruby, there are some great libraries that handle most of the work for you, like OmniAuth.

Hopefully, I've filled in enough blanks for you to get started. If you still have questions, you may find this tutorial helpful.

Feel free to post any questions or criticisms in the comments.

51 Comments **Andrew Havens, Ruby Developer** 1 **Login**

Recommend 90 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Anna Maria Mazzarisi • 6 months ago

OmniAuth link is broken. The new link is: https://github.com/omniauth...

10 • Reply • Share ›

Julien • 3 years ago

Thanks, best explanation of REST! I've spent the last 4 days reading dozen of articles and trying to understand it.

9 • Reply • Share ›

Thijs • 3 years ago

Very insightfull for a beginner, thanks!

6 • Reply • Share ›

Srinivasan Rajappa • a year ago

As I upvoted a comment and wrote this comment, all that you showed here in this tutorial was replayed. It was so convincing that I was comfortably justifying each button click, each change on this page. Thanks Author Andrew :)

3 • Reply • Share ›

Dariel Javier • 5 years ago

In Addition to the http status codes you have already set, i also like to add: status 500 unless widget.save
this way if something is wrong trying to reach the database with the require operation,an internal server error is returned.

3 • Reply • Share ›

Nick Restrepo • 4 years ago

Badass Tut. Thanks!

GitHub • Twitter • LinkedIn • Email