- ⏱️ 13 minutes
- ✅ Completed

# Getting Started With RESTful Endpoints

ReST stands for REpresentational State Transfer. The acronym doesn't fit perfectly, but we're developers and we can come up with whatever cool acronyms work for us! This may sound like a complex concept, but don't let it scare you too much.

In this reading, you will learn the definition of ReST and how to apply its design principles. You will learn about how it replaces another kind of interaction called "remote procedure calls". Finally, you'll get to a very practical way to implement ReST using the RESTful endpoint design convention.

## Rules of ReST

ReST (Representational State Transfer) is an architecture style for designing networked applications. To be clear, ReST is not an official standard. Instead, it's a set of rules/constraints. Though ReST is commonly associated with APIs, not every API actually follows RESTful conventions.

ReST defines six architectural constraints, and in this reading, we'll focus on three of them:

1. **Decoupled client-server**: The client and the server should be decoupled so that they can evolve separately without any dependence on one another.
2. **Stateless**: This means that there is no necessary session between the client and the server. Data received from the server can be used by the client independently. This allows you to have short discrete operations. Luckily, this is a natural fit for HTTP operations in which requests are intended to be independent and short-lived.
3. **Uniform interface**: RESTful APIs are meant to be self-describing and uniform in their definition. Each operation is intended to be separated by a separate endpoint or URL. In practical real world terms, most RESTful APIs implement the classic CRUD (Create, Read, Update, Delete) operations against a resource that just happens to be in your data model. This uniformity allows developers to easily learn the usage pattern of each API.

# What does a RESTful API look like?

Because RESTful APIs are meant to be representational, you can start with the data model that the API is meant to represent. For example, if you're building a Twitter clone application, you will most likely want to define API endpoints that manage the operations of your users "tweets", such as "create a tweet" and "like a tweet."

## URLs

In RESTful APIs, you generally have two kinds of URLs, ones that points at *collections of resources* and ones that point at *single resources*. Using the Twitter application as an example, a path like **/my/tweets** would point to a collection of tweets made by you. A path like **/my/tweets/17** would point to a tweet made by you with the id of 17. Usually, those ids are the primary keys of rows in your database for the record that contains the information for that specific tweet.
The naming scheme, again, is quite simple. A path that ends in a plural noun represents a collection of resources that your API provides for developers to interact with. The following examples are just naming schemes that *you would decide* as the person creating the paths that your application will handle.

- **/invoices** would represent a collection of invoices that you're allowed to see
- **/people** would represent the people in the application that you're allowed to see
- **/houses** would represent a collection of houses

A path that combines a plural noun and a specific identifier represents a single resource in your application. Often, the identifier is a primary key from the database. However, if you have a unique column that also identifies a specific record, you could use that instead.

- **/invoices/PK-200201** would represent the single invoice that has the the invoice number "PK-200201"
- **/people/10103** would represent the single person with id 10103
- **/houses/bdfa5ef9-0c86-4810-bc13-10415250af09** would represent the house with the specific globally unique identifier "bdfa5ef9-0c86-4810-bc13-10415250af09"

## AJAX URLs and HTTP verbs

For collections of resources URLs, the following table describes what each HTTP verb means for interacting with that URL. The responses will almost invariably be formatted as JSON.

| HTTP Verb | Meaning | With respect to /my/tweets |
|---|---|---|
| GET | Get "all" of the specified resources | Get all of your tweets |
| POST | Create a new resource | Create a new tweet |
| PUT | n/a | n/a |
| PATCH | n/a | n/a |
| DELETE | Delete all of the resources | Delete all of your tweets |

For single resource URLs, the following table describes what each HTTP verb means for interacting with that URL. Again, the responses will invariably be formatted as JSON.

| HTTP Verb | Meaning | With respect to /my/tweets/17 |
|---|---|---|
| GET | Get the details of the resource | Get that specific tweet with id 17 |
| POST | n/a | n/a |
| PUT | Replace the resource | Replace all of the tweet details with the provided data |
| PATCH | Update the resource | Update specific properties of the tweet |
| DELETE | Delete the specified resource | Delete that specific tweet |

When dealing with these RESTful endpoints, the actions don't necessarily mean that records *will* get created or destroyed in your database, or only one record will be affected.
For example, you may decide to perform "soft deletes" on some of your data. This means that, instead of removing a record from a table, you mark the record "deleted". (You can enable this with the paranoid configuration setting in your Sequelize models.) This means that the HTTP DELETE request would cause a SQL UPDATE rather than DELETE. What is important is that the *concept* of a delete has been performed.
In another example, consider the path that reads **/weather/current**. That doesn't point to any static single record in the weather database. Instead, it would return the *most recent* record of weather in the database. The id of "current" would be treated special and initiate a lookup of the most recent record rather than a specific record like **/weather/10392**.

# HTML URLs and HTTP verbs

RESTful APIs don't need to be just data APIs returning JSON. The URLs that return HTML can follow a RESTful concept, as well. What it means, though, is that you will be limited to just the verbs GET and POST. That's all HTML-based views can generate. The following tables show the paths and HTTP verbs used to interact with HTML-based versions of a RESTful application.

| Path | HTTP Verb | Meaning |
|---|---|---|
| **/my/tweets** | GET | Get an HTML-based list of your tweets |
| **/my/tweets/new** | GET | Show a form to create a new tweet |
| **/my/tweets** | POST | Create a new tweet |
| **/my/tweets/17** | GET | See the details of your tweet with the id of 17 |
| **/my/tweets/17/edit** | GET | Show the edit form for your tweet with the id of 17 |
| **/my/tweets/17** | POST | Update the tweet with the submitted details |
| **/my/tweets/17/delete** | POST | Delete your tweet with the id of 17 |

All of the GET requests get HTML responses for the browser to show. All of the POST requests usually end in a redirect to another page that makes sense:

- After creating a resource, redirect to its detail page
- After editing a resource, redirect to its detail page
- After deleting a resource, redirect to the list page

# Designing your API

The simplest way to figure out what paths you need in your application is to figure out what resources you need in your application. For many developers, this means looking at the models in your application (and the database tables that power the models).

If you're creating a pet shelter management application, you would need to know about animals, animal types, what kennel they're in, who works there, and more. So, you would have some models like

- `Animal`
- `AnimalType`
- `HealthRecord`
- `Kennel`
- and more...

For each of those, you could create paths in your RESTful API to allow the Web application to interact with those resources:

- **/animals** would be the collection of animals
- **/animal-types** would be the collection of animal types
- **/animals/137/health-records** would be the health records for the specific animal whose id is 137
- **/kennels** would be the collection of kennels available at the shelter in which you could put the animals
- and more...

It all depends on what you need to interact with to make your application work like you want it to.

# GitHub API example

GitHub has a much vaunted RESTful API that many people use as a model for how to do *good API design*. This section checks out some of its features.
First, check out this GitHub REST API endpoint for a GET request to the `app-academy` user and the endpoint's response below. Notice how the information is formatted as JSON. JSON is the preferred format over other formats like XML. Feel free to take a look at the GitHub API documentation for sending a GET request for a single user.
If your browser is rendering the JSON below without the quotes around the property names, you're likely using the extension JSONView to *prettify* your JSON in JavaScript. The JSONView extension *prettifies* JSON by parsing the JSON text data into a JavaScript object. Note that you can also parse JSON into JavaScript by using the `JSON.parse()` method. Another extension, JSON Viewer, also *prettifies* JSON but it correctly preserves the quotes around the property names.
If you opened https://api.github.com/users/app-academy endpoint in your browser, you should see the JSON formatted data below. Feel free to navigate to the links in the JSON.


```
{
  "login": "app-academy",
  "id": 3155975,
  "node_id": "MDQ6VXNlcjMxNTU5NzU=",
  "avatar_url":
"https://avatars0.githubusercontent.com/u/3155975?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/app-academy",
  "html_url": "https://github.com/app-academy",
  "followers_url": "https://api.github.com/users/app-
academy/followers",
  "following_url": "https://api.github.com/users/app-
academy/following{/other_user}",
```

```
  "gists_url": "https://api.github.com/users/app-
academy/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/app-
academy/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/app-
academy/subscriptions",
  "organizations_url": "https://api.github.com/users/app-
academy/orgs",
  "repos_url": "https://api.github.com/users/app-
academy/repos",
  "events_url": "https://api.github.com/users/app-
academy/events{/privacy}",
  "received_events_url": "https://api.github.com/users/app-
academy/received_events",
  "type": "User",
  "site_admin": false,
  "name": null,
  "company": null,
  "blog": "",
  "location": null,
  "email": null,
  "hireable": null,
  "bio": null,
  "public_repos": 2,
  "public_gists": 1,
  "followers": 3,
  "following": 0,
  "created_at": "2012-12-31T00:08:43Z",
  "updated_at": "2016-02-27T05:27:57Z"
}
```


Navigate to the JSON's followers_url and repos_url. Notice how the url path of those endpoints connect to the endpoint that fetches all the data connected to the `app-academy` user.

Now open the followers_url endpoint in your browser. You are now are sending a `GET` request to GitHub's server to receive an array of followers associated to the `app-academy` user. Keep in mind that there are many Public APIs available as you plan your first full-stack project next week.

# RESTful vs remote procedure calls (RPC)

Defining HTTP APIs can come in many shapes and styles. The style you adopt may flow out of the problem that you are attempting to solve or it could just be your personal preference. In programming, there is usually more than one valid approach, and making HTTP based APIs is no exception. And while RESTful APIs are certainly one of the most popular styles for creating an API, it is not the only way.

## A remote procedure call

Remote procedure calls are like methods on objects rather than database operations. Clients make requests centered around specific operations. For example, with a RESTful API, you would see a GET request to this path to retrieve a specific tweet: `http://localhost/tweets/12`. In an RPC-based API, you could see a path similar to this `http://localhost/getTweetById?id=12`.

Your route would then have an action to handle that specific "call".

RPC style endpoints are notorious for specifying the method name in the URL.

This can be convenient, because by looking up the URL you can immediately understand what it does and what you get in return, and it can offer a way to get around some of the more complicated nested paths mentioned earlier.

The drawback is that you need comprehensive documentation to know all of the different methods available by an RPC-designed API. Contrast that to RESTful APIs where you need know only the resources; once you know the resources, you immediately know how to interact with them using the HTTP verbs.

## Which is better? ReST or RPC?

"Better" is very subjective. RESTful APIs are definitely more popular nowadays. Although Express is an un-opinionated framework, a typical Express setup is more conducive to following RESTful conventions. For example, routers and routes are organized in a way to match RESTful URLs, and routes are defined around the core HTTP verbs.

Generally speaking, in this course, you'll want to use RESTful APIs, as it offers a predictable and systematic way of organizing your API.

# What you learned

In this article, you covered quite a bit. You learned that

1. ReST is an acronym for Representational State Transfer.
2. RESTful services differ from traditional remote-procedure call styled services.
3. ReST is not an official standard.

4. RESTful APIs utilize a client/server architecture.
5. Data from your APIs are represented by resources and are accessed using Uniform Resource Locators (URLs).
6. Endpoints are a combination of a resource and an operation.
7. CRUD operations are represented by HTTP methods (POST, GET, PUT, DELETE, and sometimes PATCH) and a URL.
8. RESTful APIs are stateless.