Lesson 13

* REST
  + Up to this point, we’ve learned how to write a client that connects to a database, but we haven’t learned how to make the data of an application accessible outside of the console. In a full stack architecture, the front end, or what the user sees and interacts with, sends requests to the back end (which is what we’ve been building so far) and then the back end sends a response back to the front end to be rendered for display.
  + To do this, we have different protocols that we can choose from. The common choice in today’s development is Hypertext Transfer Protocol, or HTTP. While most new development is done with HTTP, HTTP doesn’t define how an application should be architected. That can cause HTTP to be used, and applications to be created, in many different, inconsistent ways across different companies and applications.
  + In order to mitigate this, Roy Fielding wrote a 150+ page dissertation on an architectural style that should be followed in order to develop better software. He named the architectural style Representational State Transfer, or REST. REST is not a protocol, it is not HTTP, but it can be thought of, in part, as the way we should use HTTP to create web services and applications.
  + As the name implies, Representational State Transfer focusses heavily on transferring the state of a resource from one machine to another, so an example would be from a front end application to the back end, or vice versa. A resource is a noun, such as customer, order, user, or anything else that could represent an entity. When we transfer the state of a resource, we have multiple choices to represent that state, it could be plain text, xml, json, or something else entirely. JSON, or JavaScript Object Notation, is the preferred way for most developers.
  + An example of the state of a resource is typically conveyed in key value pairs, where the properties or fields of a resource have values assigned that represent the state of that entity at that given point in time. For example, you could have a bank account resource and the state could have an account number, a balance, and an account type. There could be more properties as well, but we’ll stick with these for now. In JSON, this could be represented as:
    - {accountNumber: “9837298”, balance: 670.43, accountType: “checking”}
  + The state of this bank account resource can change, and as it changes, the front and back end can send the new state to each other so the user and the database are aware of the new, current state.
  + REST is more than just this, but this concept is one of the most noticeable and recognizable.
* HTTP Verb Methods
  + We’ve talked about REST and how it means to transfer the state of a resource, but we haven’t talked about how that would look. In this class we are building back end, web APIs. That means that a front end client will send requests to our API via HTTP, and we will process the request and send a response back. An example of this would be if we build a restaurant finding API and someone sends a request to find all restaurants within 5 miles of Phoenix, our API would query the database for the proper restaurant data, map it to a List of entities, and then send the response back to the front end client that made the request so it can render the data for the user to then see and interact with further.
  + In order to build an API that processes requests, we need two things. First, we need to have endpoints; an endpoint is like an address, or a dock, where requests can be sent and processed for a specific resource. For example, in our restaurant API we might have an endpoint for Restaurant, User, and Reservation. These endpoints are also known as Uniform Resource Identifiers, or URIs. Any request sent to the Reservation URI, or endpoint, will affect our Reservation data, or in other words will transfer the state of one or more Reservations from the back end to the front end or vice versa. These endpoints would look something like this:
    - [www.myapi.com/v1/restaurant](http://www.myapi.com/v1/restaurant)
    - [www.myapi.com/v1/user](http://www.myapi.com/v1/user)
    - [www.myapi.com/v1/reservation](http://www.myapi.com/v1/reservation)
    - The beginning of the endpoint can be just about anything, but the important part is the last segment, slash resource name
  + Now that we know what an endpoint looks like, the second thing we need to process requests from an API is to know *what* we are supposed to do with a resource sent to a specific endpoint. As we learned in our database course, there are four types of operations that can be performed on data, and those operations are represented by our CRUD acronym. Create, Read, Update, and Delete. So, when we receive a request at one of our endpoints, we need a way to tell what CRUD operation we are performing. To specify which CRUD operation the client wants our API to perform on the specified resource, the client will send different types of HTTP requests.
  + HTTP has different verb methods that we can make use of, and the four most commonly used HTTP methods can be mapped directly to our CRUD operations. These four methods are GET, POST, PUT, and DELETE, and are mapped like this:
    - Create = POST
    - Read = GET
    - Update = PUT
    - Delete = DELETE
  + What this means is that if a front end client sends an HTTP GET request to our reservation endpoint, we know that they want to read the data that represents one or more reservations, and if they send an HTTP POST request to the restaurant endpoint, they want to create a new restaurant with the data payload included in their request.
  + Now that we know an endpoint, or URI, identifies a specific resource, and the type of HTTP verb method request determines what CRUD operation to perform on that resource, let’s look at a standard way to determine which specific instance the operation should be performed on. To do this, we just need to add some parameters to a few of our endpoints.
    - //use excel to document what it will look like

|  |  |  |
| --- | --- | --- |
| * + - HTTP VERB | * + - URI | * + - What it Does |
| * + - GET | * + - /restaurant | * + - Retrieve all restaurants |
| * + - GET | * + - /restaurant/{id} | * + - Retrieve a specific restaurant by id |
| * + - POST | * + - /restaurant | * + - Create a new restaurant |
| * + - PUT | * + - /restaurant/{id} | * + - Update a specific restaurant by id |
| * + - DELETE | * + - /restaurant/{id} | * + - Delete a specific restaurant by id |

* + As you can see by these examples, we can append more information to the end of our endpoint to make the request more specific, such as identifying an instance by an id parameter. This could be taken even further by adding more. Let’s pretend we have a social media API that has users, posts, and comments. We could have an endpoint like: /user/34/posts/12/comments
    - Sending an HTTP GET request to this URI should return all the comments for the 12th post made by the user with the userId of 34. I say *should*  because we choose how to implement our APIs when building them, but this is the standard implementation that has been accepted. If you sent a POST request to this same URI, it would be to create a new comment on the 12th post of user 34.
* Postman
  + When building a RESTful Web API, it’s good to test your API as you build it and add new endpoints and features. With that being said, it rarely makes sense to build a front end to test your API because of the extreme overhead it adds to build the whole other side of the picture. In addition, sometimes APIs don’t have or need a front end counterpart, or they are created for other APIs or services to consume. Since we can’t always build a front end application to test our back end API, we need a client tool that can send requests for us to test. For this, we can use Postman.
  + Postman is “a powerful HTTP client for testing web services” that allows us to send different HTTP requests to API endpoints. To install Postman, google postman and then click on the download link and find the download option that matches your operating system. Go ahead and download and install Postman.
  + Once it is installed we can open it up and see what it looks like. We have a bar at the top here that allows us to enter the endpoint and determine the HTTP verb method to be sent. To test it out, we can use reqres.in (<https://reqres.in/>), which is a quote unquote fake API we can use to test Postman. The data we create and delete won’t persist when using reqres.in, but it will enable us to practice using Postman.
  + To start, let’s try out a GET request on the reqres.in user endpoint.
    - Reqres.in/api/user
    - We can add something called a query parameter to the end of our request endpoint using a question mark and a key value pair. The key we will use is page and we can set it equal to whatever page of user data we want to see.
    - Note that keys are defined by the developer of the API and can be anything they choose, they are not standard keys that exist for every API.
  + Next, let’s retrieve a specific user
  + Now we can test a delete on a specific user
  + Next, let’s create a new user by sending a POST request. This request is different than the previous two because it contains a payload with the data we want to send to the API to create a user. In order to send the payload, we have to define what type of data the payload will be. This will have to match the type of data that the API we are sending it to accepts. With that being said, this API accepts application/json, which is pretty standard for modern RESTful APIs. To set this, click on body, then select raw, and change the dropdown to JSON(application/json)
    - {
    - "name": "morpheus",
    - "job": "leader"
    - }
  + We can also test an update with a POST request
  + As I mentioned earlier, any request we send that would change the data, such as POST, PUT, and DELETE requests, will not actually persist in this test API, but it is a good way to practice seeing how requests are sent, and what a response can look like. When we start building our own RESTful APIs, we will implement endpoints that do persist the data through to the database.
* API Documentation
  + In order to know what requests you can send to an API and what those requests will result in, API developers should provide documentation on the functionality of the API. In addition to documentation being useful for consumers of an API, it is also beneficial to document what your API will do before you start building it so that the documentation can be used as a blueprint, or map, of how you want to build. It can be like a checklist of what endpoints and HTTP methods you have left to implement.
  + There are many different ways to document APIs. Let’s look at a couple examples of API documentation:
    - <https://developer.spotify.com/documentation/web-api/reference/>
    - <https://developer.github.com/v3/users/followers/>
  + To start creating documentation to use as a blueprint, we don’t have to have all the details. We can list the endpoints, the HTTP verbs we want each endpoint to accept, and a simple description of what each endpoint is used for. We can use any text editor or spreadsheet editor to do this. I will give an example using Microsoft Word.