INTRODUCTION TO WEB, PART 2

LESSON

WHAT IS A REST API?

REST API

- One of the most popular types of API is REST or, as they're sometimes known, RESTful APIs. REST or RESTful APIs were designed to take advantage of existing protocols. While REST - or Representational State Transfer - can be used over nearly any protocol, when used for web APIs it typically takes advantage of HTTP. This means that developers have no need to install additional software or libraries when creating a REST API.

Client-Server

- The client-server constraint works on the concept that the client and the server should be separate from each other and allowed to evolve individually and independently. In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client. This creates a separation of concerns, letting each application grow and scale independently of the other and allowing your organization to grow quickly and efficiently.

Stateless

- REST APIs are stateless, meaning that calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully. A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself. Identifying information is not being stored on the server when making calls. Instead, each call has the necessary data in itself, such as the API key, access token, user ID, etc. This also helps increase the API's reliability by having all of the data necessary to make the call, instead of relying on a series of calls with server state to create an object, which may result in partial fails. Instead, in order to reduce memory requirements and keep your application as scalable as possible, a RESTful API requires that any state is stored on the client—not on the server.

Cache

- Because a stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data. This means that when data is cacheable, the response should indicate that the data can be stored up to a certain time (expires-at), or in cases where data needs to be real-time, that the response should not be cached by the client. By enabling this critical constraint, you will not only greatly reduce the number of interactions with your API, reducing internal server usage, but also provide your API users with the tools necessary to provide the fastest and most efficient apps possible. Keep in mind that caching is done on the client side. While you may be able to cache some data within your architecture to perform overall performance, the intent is to instruct the client on how it should proceed and whether or not the client can store the data temporarily.

Uniform Interface

- The key to the decoupling client from server is having a uniform interface that allows independent evolution of the application without having the application's services, models, or actions tightly coupled to the API layer itself. The uniform interface lets the client talk to the server in a single language, independent of the architectural backend of either. This interface should provide an unchanging, standardized means of communicating between the client and the server, such as using HTTP with URI resources, CRUD (Create, Read, Update, Delete), and JSON.

Layered System

- As the name implies, a layered system is a system comprised of layers, with each layer having a specific functionality and responsibility. If we think of a Model View Controller framework, each layer has its own responsibilities, with the models comprising how the data should be formed, the controller focusing on the incoming actions and the view focusing on the output. Each layer is separate but also interacts with the other. In REST API design, the same principle holds true, with different layers of the architecture working together to build a hierarchy that helps create a more scalable and modular application.

RESTFUL API DESIGN TIPS

USE API VERSIONING

- If you're going to develop an API for any client service, you're going to want to prepare yourself for eventual change. This is best realised by providing a "version-namespace" for your RESTful API.

- We do this with simply adding the version as a prefix to all URLs.

```
GET www.myservice.com/api/v1/posts
```

Avoid using verbs in URIs

- If you've understood the basics, you'll now know it is not RESTful to put verbs in the URI.
- This is because HTTP verbs should be sufficient to describe the action being performed on the resource.
- Let's say you want to provide an endpoint to generate and retrieve a banner image for an article. I will note **:param** a placeholder for an URI parameter (like an ID or a slug). You might be tempted to create this endpoint:

```
GET: /articles/:slug/generateBanner/
```

- But the GET method is semantically sufficient here to say that we're retrieving ("GETting") a banner. So, let's just use:

```
GET: /articles/:slug/banner/
```

- Similarly, for an endpoint that creates a new article:

```
# Don't
POST: /articles/createNewArticle/
# Do
POST: /articles/
```

Use plural resource nouns

- It may be hard to decide whether or not you should use plural or singular form for resource nouns.
- Should you use
  - **/article/:id/** (singular)
  - or **articles/:id/** (plural)?

- **GET /article/2/** is fine, but what about **GET /article/**? Are we getting the one and only article in the system, or all of them?

```
GET: /articles/2/
POST: /articles/
...
```

Return error details in the response body

- When an API server handles an error, it is convenient (and recommended!) to return **error details** in the JSON body to **help users with debugging**. Special kudos if you include which fields were affected by the error!

```
{
    "error": "Invalid payoad.",
    "detail": {
        "surname": "This field is required."
    }
}
```

Pay attention to status codes

- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Use status codes consistently

```
GET: 200 OK
POST: 201 Created
PUT: 200 OK
PATCH: 200 OK
DELETE: 204 No Content
```

Make use of the query string for filtering and pagination

- A lot of times, a simple endpoint is not enough to satisfy complex business cases.
- Your users may want to retrieve items that fulfill a specific condition, or retrieve them in small amounts at a time to improve performance.
- This is exactly what filtering and pagination are made for.
- With filtering, users can specify properties that the returned items should have.
- Pagination allows users to retrieve fractions of a data set. The simplest kind of pagination is page number pagination, which is determined by a page and a page_size.

```
GET: /articles/?published=true&page=2&page_size=20
```

REFERENCES

- https://www.mulesoft.com/resources/api/what-is-rest-api-design
- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- https://blog.florimondmanca.com/restful-api-design-13-best-practices-to-make-your-users-happy
- https://medium.com/studioarmix/learn-restful-api-design-ideals-c5ec915a430f