Notes:

This document contains all the links and personal notes pertinent to the development of the various modules of this project:

1. REST APIs and HTTP

**Recommended books and Resources on REST according to Quora Answers**

<https://www.quora.com/Which-book-is-the-best-for-learning-about-Restful-Web-APIs-in-Java-Python-C>

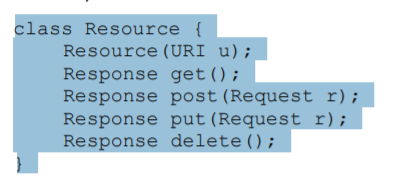
**The best introduction to REST APIs. This is considered the REST Bible**

<https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>

Architectural constraints imposed by REST:

1. *Client-server*: Separation of concerns between the client and server-side parts of the software (user interface and data storage, respectively).
2. *Stateless:* A request from the client to server must contain all the necessary information necessary to understand the request. In other words, the client cannot use any stored context on the server to process information. To be specific, only the *server is stateless.* State about the session must be kept entirely on the client side. (*client-stateless-server)*
3. *Cache:* Cache constrainsts require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. (*client-cache-stateless server)*.
4. *Uniform Interface:* REST emphasizes the implementation of a uniform interface between components/resources. **Rest Magazines** helps clarify that this specifically refers to the main operations of each resource namely **GET, POST, PUT, DELETE**
5. *Layered System:* Each component cannot see the layer above or below it (abstraction).
6. *Code-*On-Demand: REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

Another part of REST stands out to me:  
It states that **every** **resource** must support the same interface. In some Java-like syntax, this interface would look like this



**TODO: Study up on the definition of these methods according to the HTTP specification**

**Some properties (contracts if you will) of each method:**

1. **GET –**

* **GET supports efficient caching**, so it does not always have to send a request to the server
* **GET is idempotent**. It is impossible to tell if the request reached the destination when no response is received. The solution to this is usually requesting it again.
* **Safe,** because you have no obligation when you call it.

1. **PUT**

* This is idempotent as well
* Means: Update this resource with this data or create it at this URI if it’s not already there

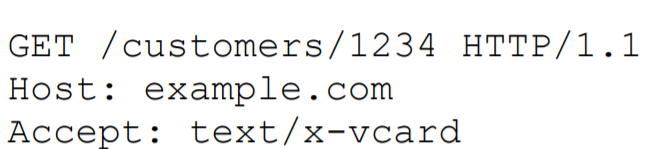
1. **DELETE**

* **Means:** You can try again and again until you get a result, delete something that’s not there is not a problem

1. **POST**

* Means: Creates a new resource.
* Used to invoke arbitrary processing and this is neither safe nor idempotent

**Yet another interesting aspect of REST.** Resources that can be expressed in some standard format should be represented in that format.



The Accept section represents the format of the resource

**A clarification on statelessness.** An application does not always have to store its state on the client-side. The state itself can also be represented as a **resource state** as well.

**Well, this was a simplication all along.**

I am currently reading on JAX-RS, a Java-based API for REST. I am also reading RESTFul Web Services, a well-known book on the theory of REST. Both books are currently recorded on Toggl, so there is no need to place them here.

**JAX-RS RESTEasy Installation for IntelliJ:**

<https://medium.com/@nicolaifsf78/intellij-idea-maven-resteasy-tomcat-f95bb41e6362>