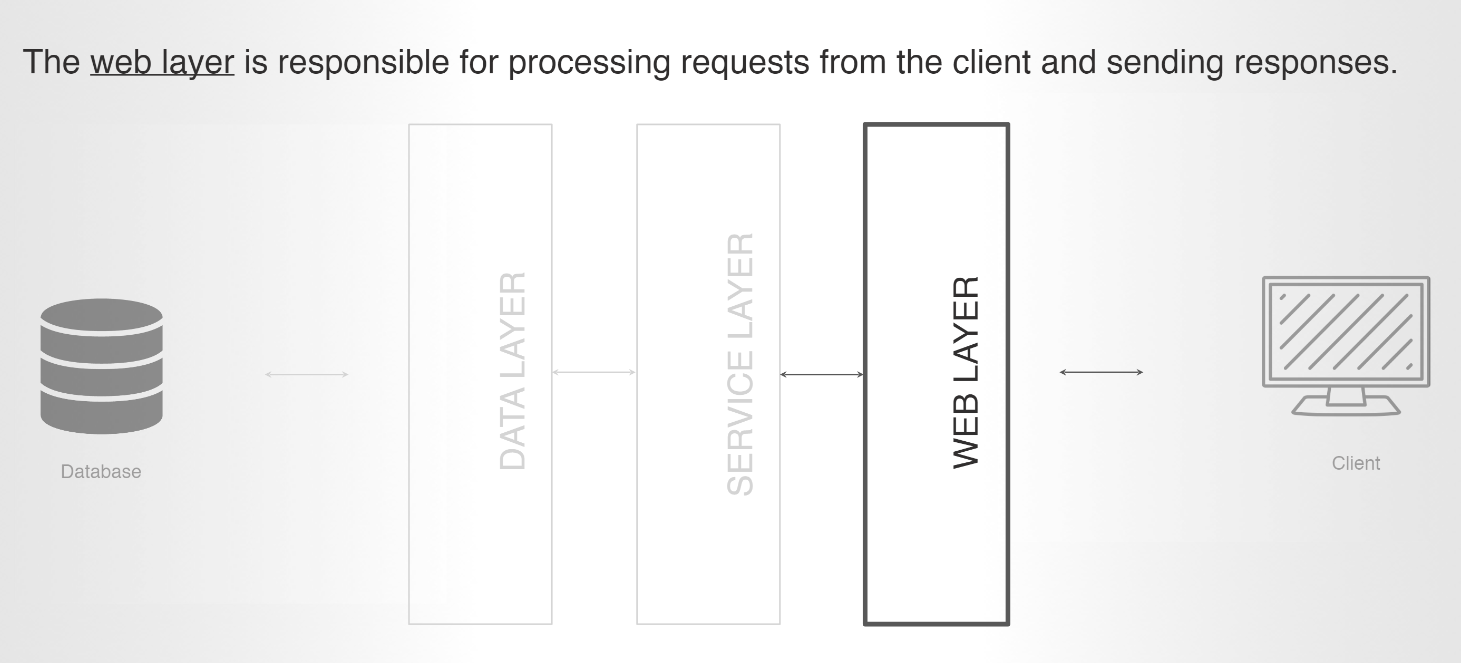
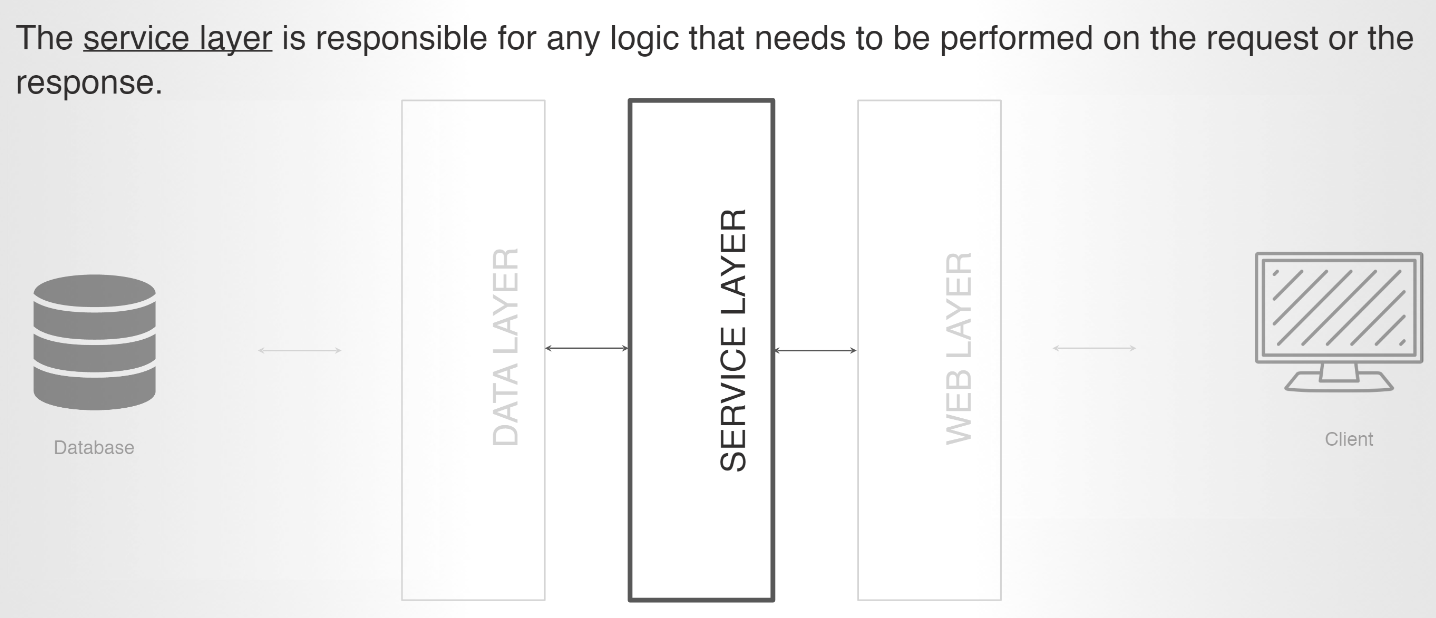
## In spring Initializer:

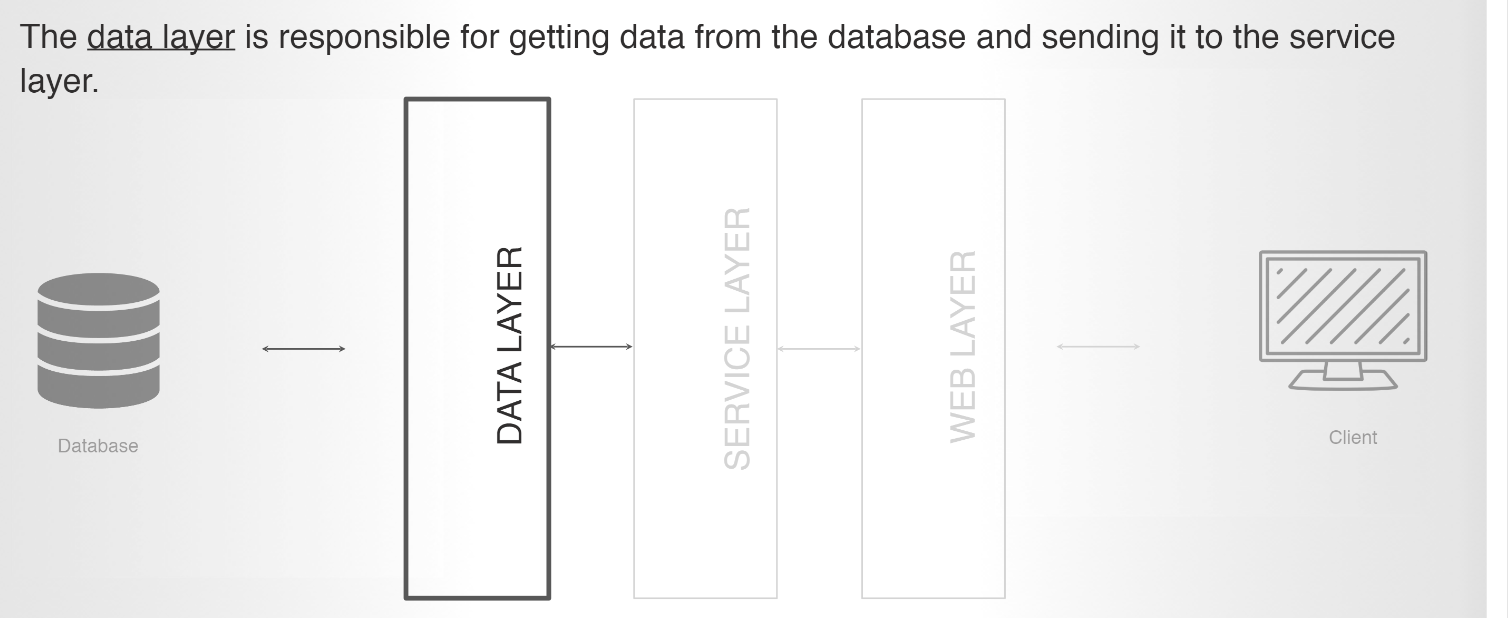
<https://start.spring.io/> =🡺 to generate code

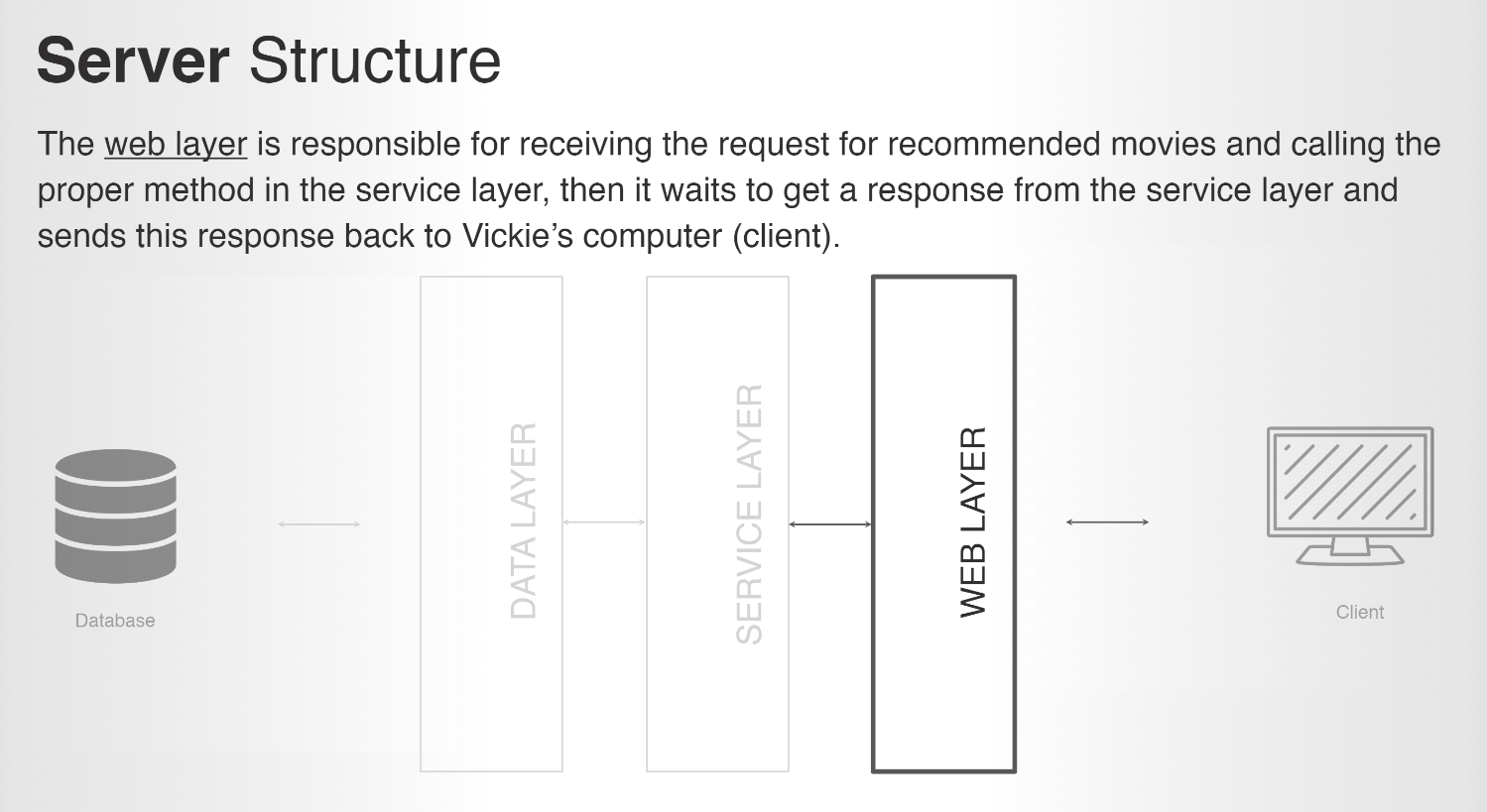
## 

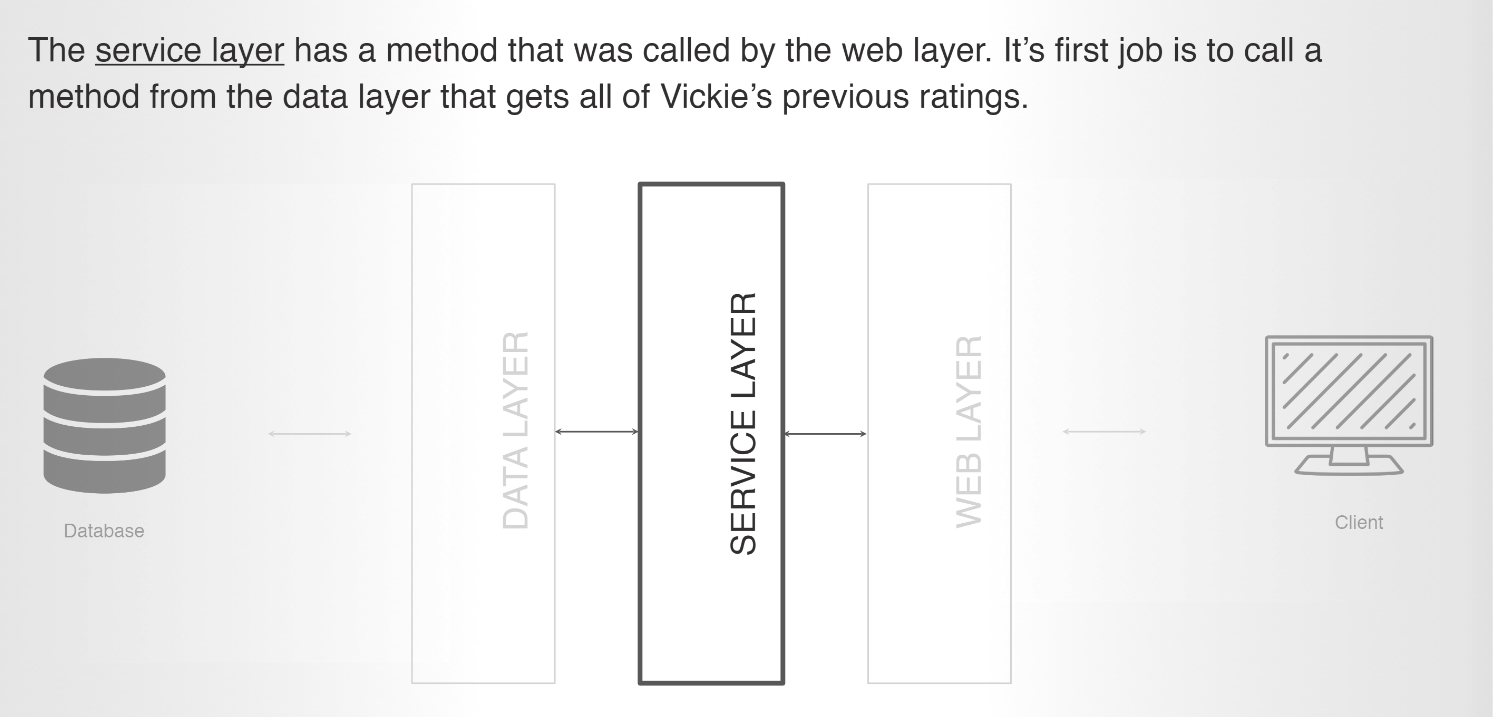
## 

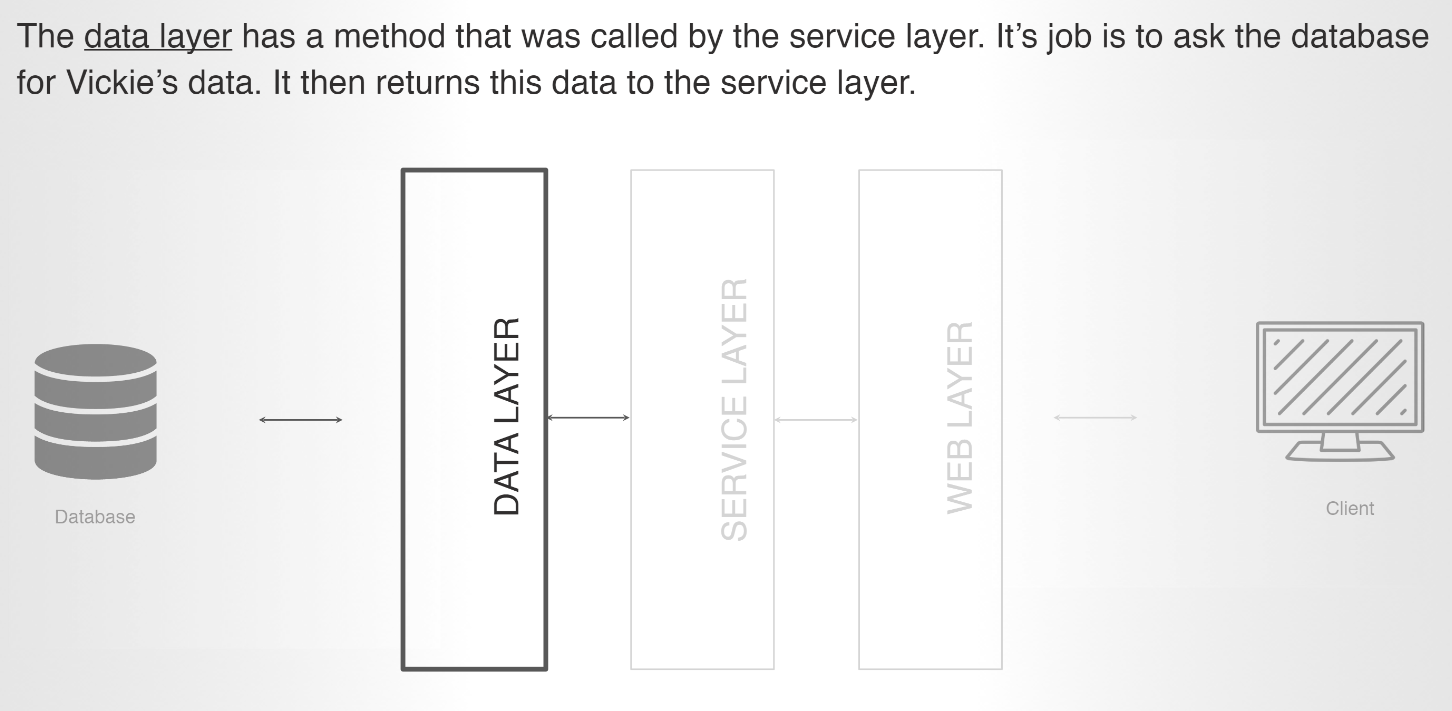


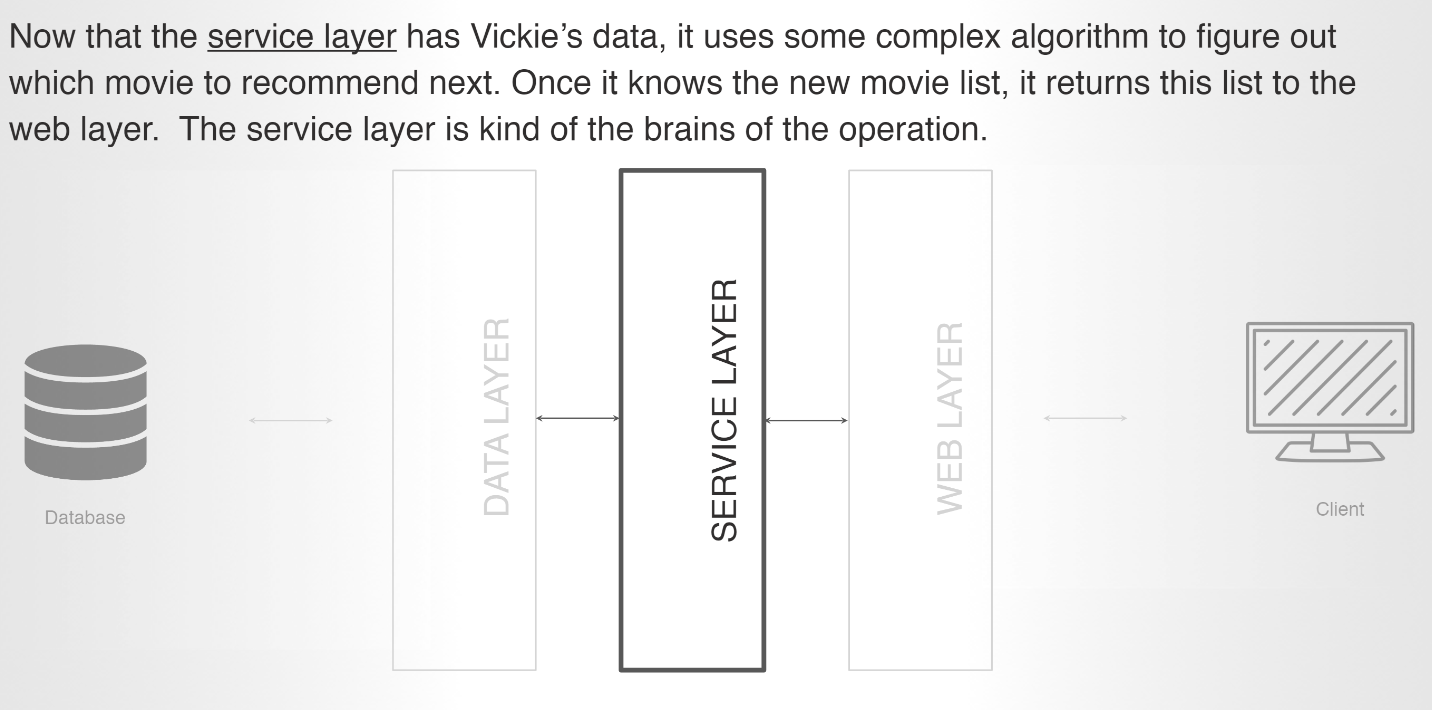








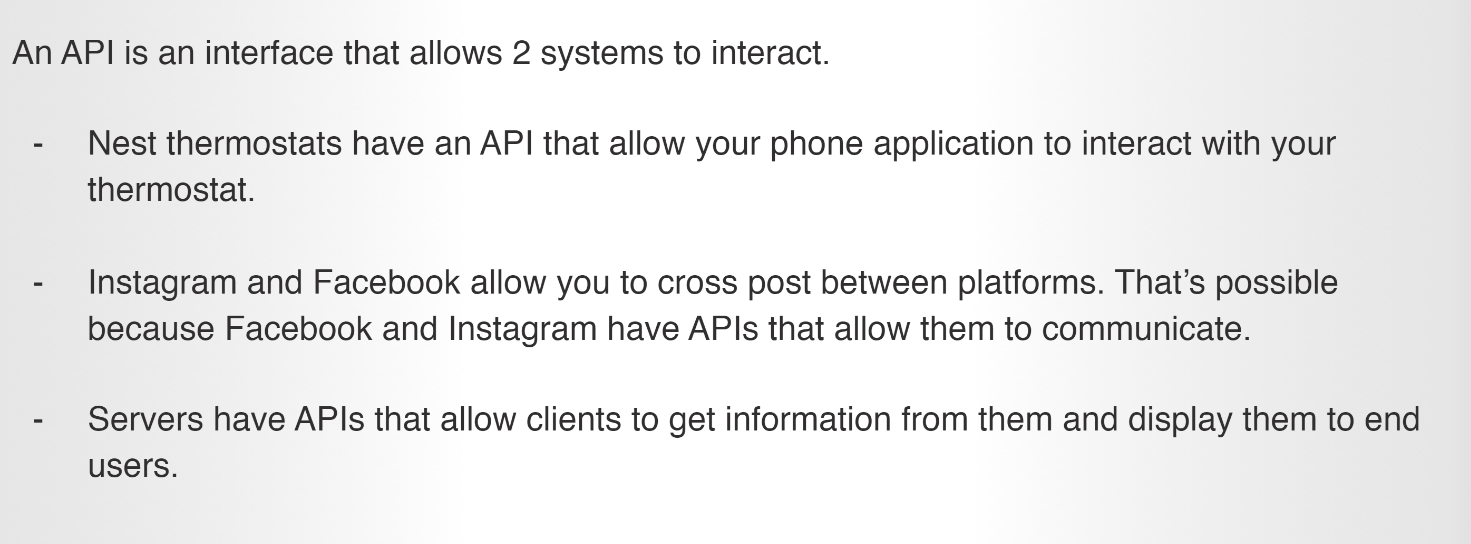


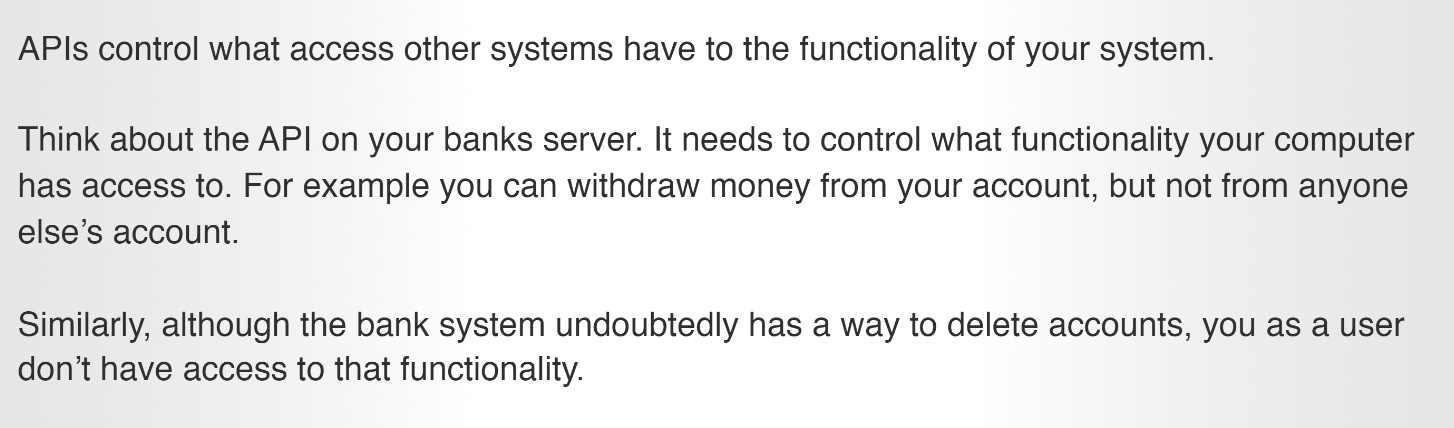


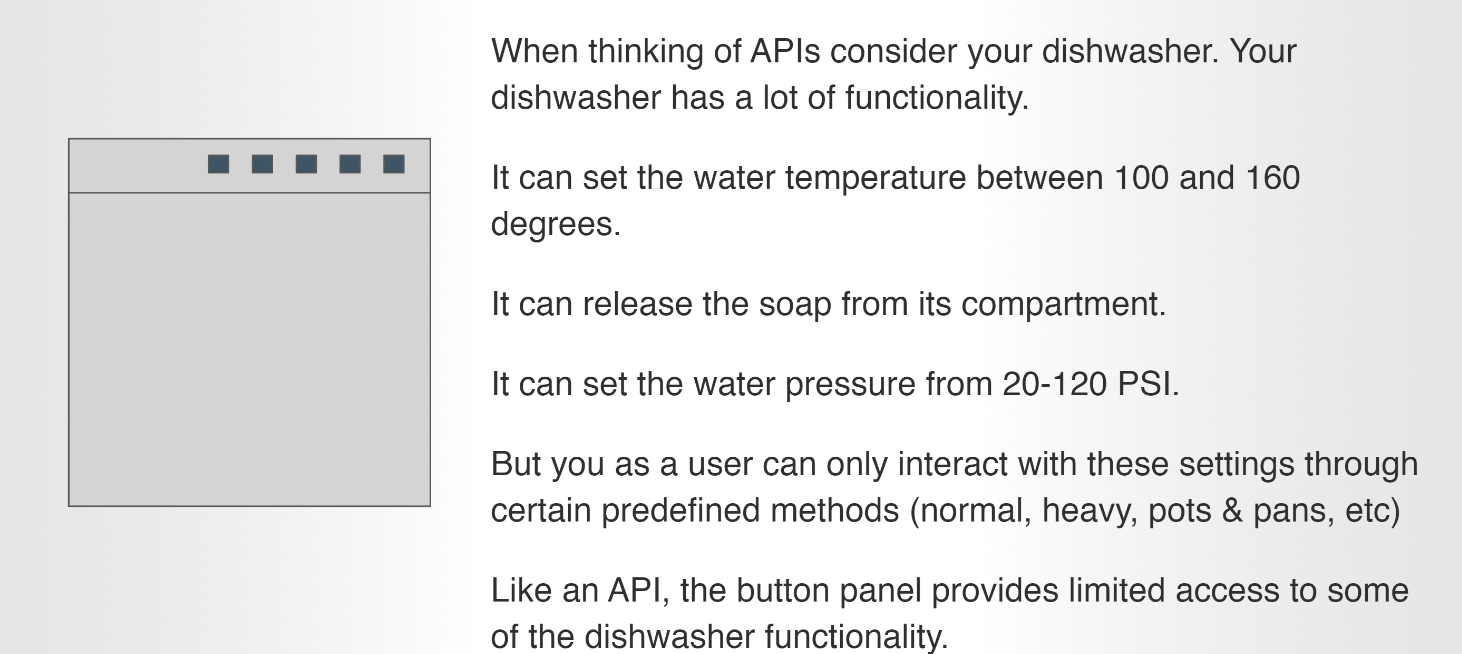


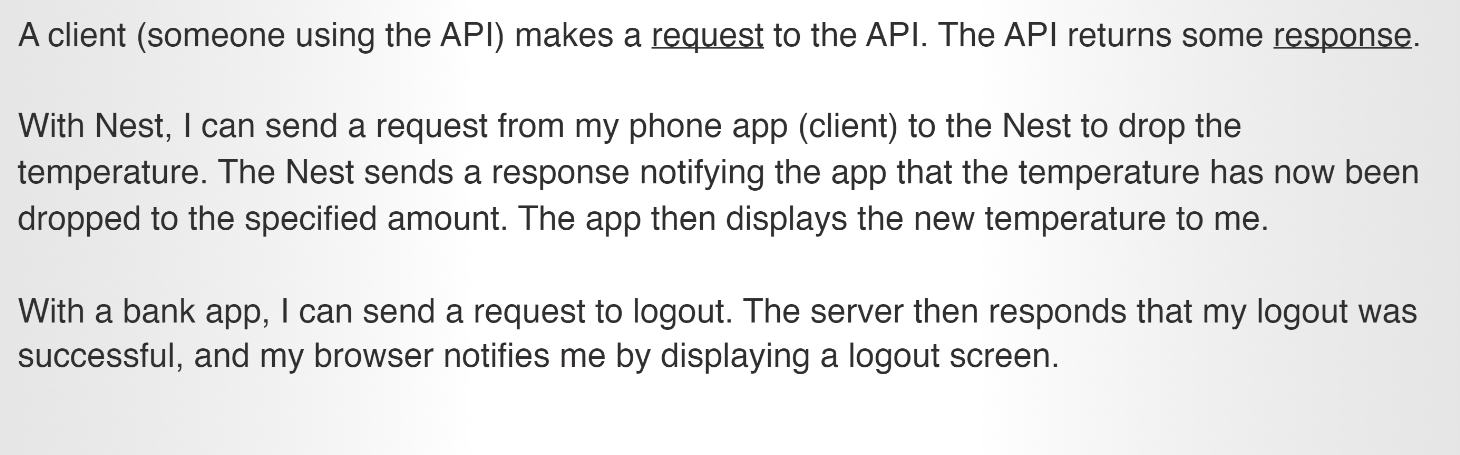
Representational State Transfer

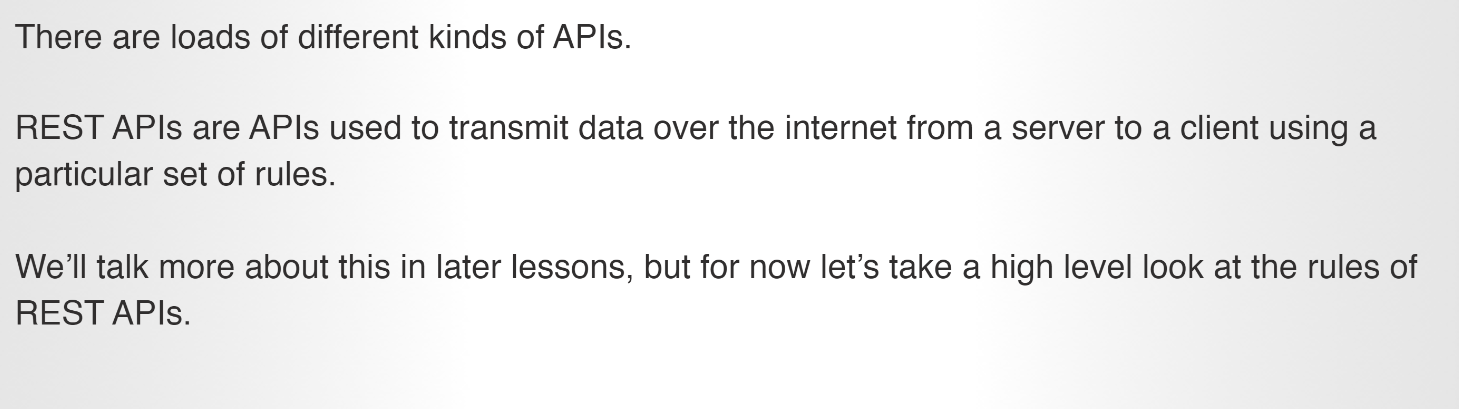
We use HTTP verb to dictate what operation we want to do

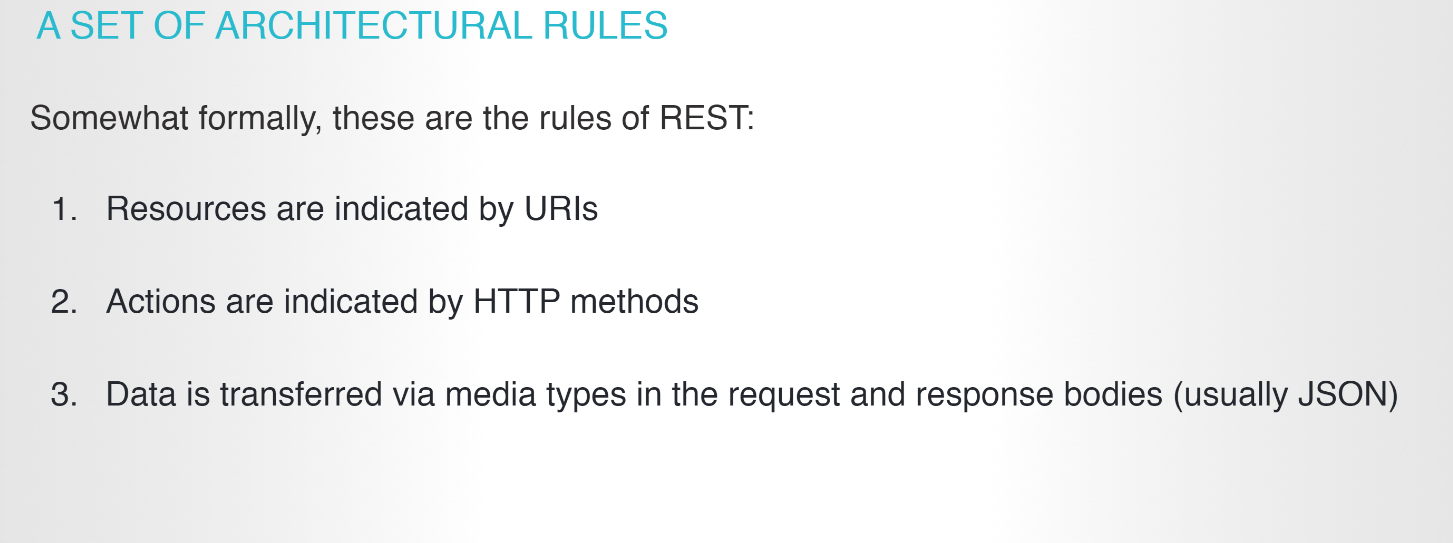












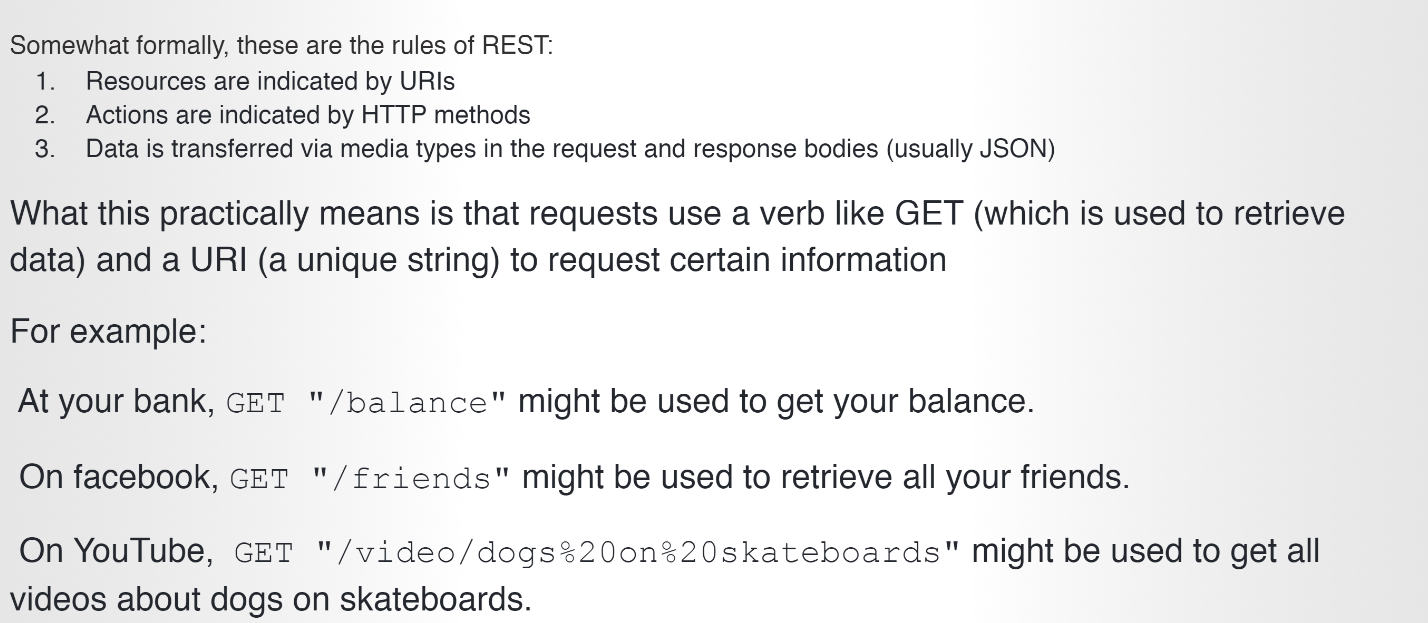
In REST, primary data representation is called **Resource**.  Having a strong and consistent REST resource naming strategy – will definitely prove your one of the best design decisions in long term.

*The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. “today’s weather in Los Angeles”), a collection of other resources, a non-virtual object (e.g. a person), and so on. In ot*

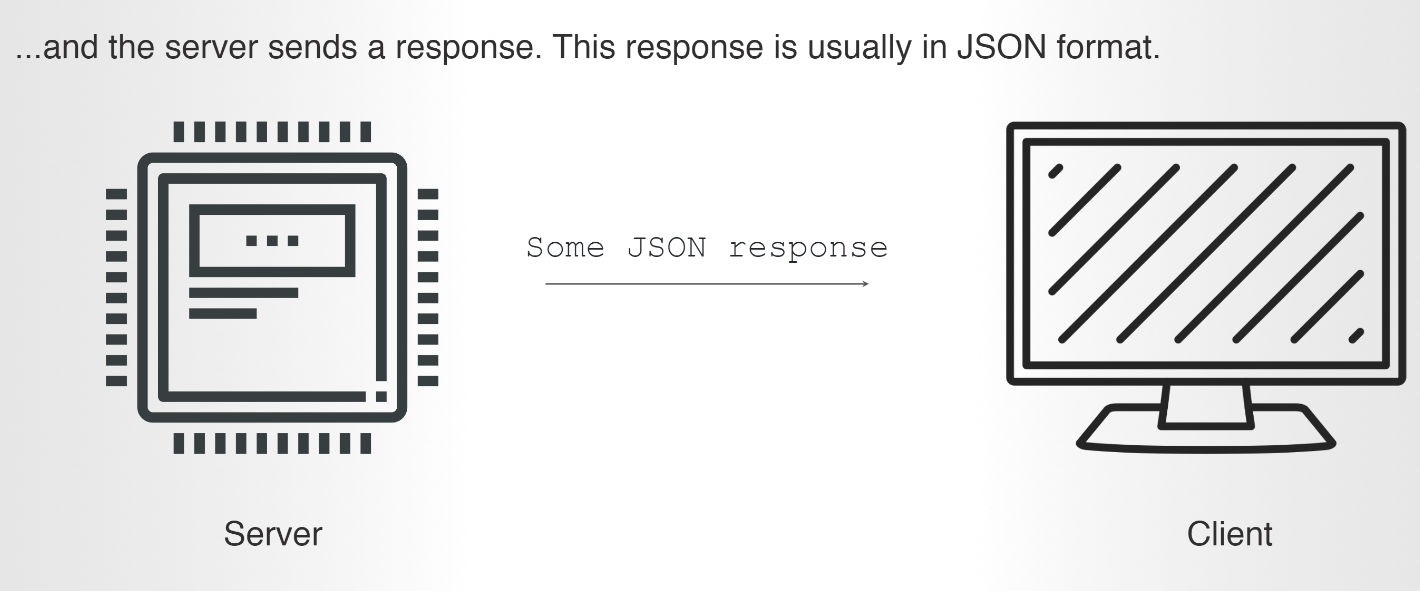
*her words, any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.*[Roy Fielding’s dissertation](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2_1_1)

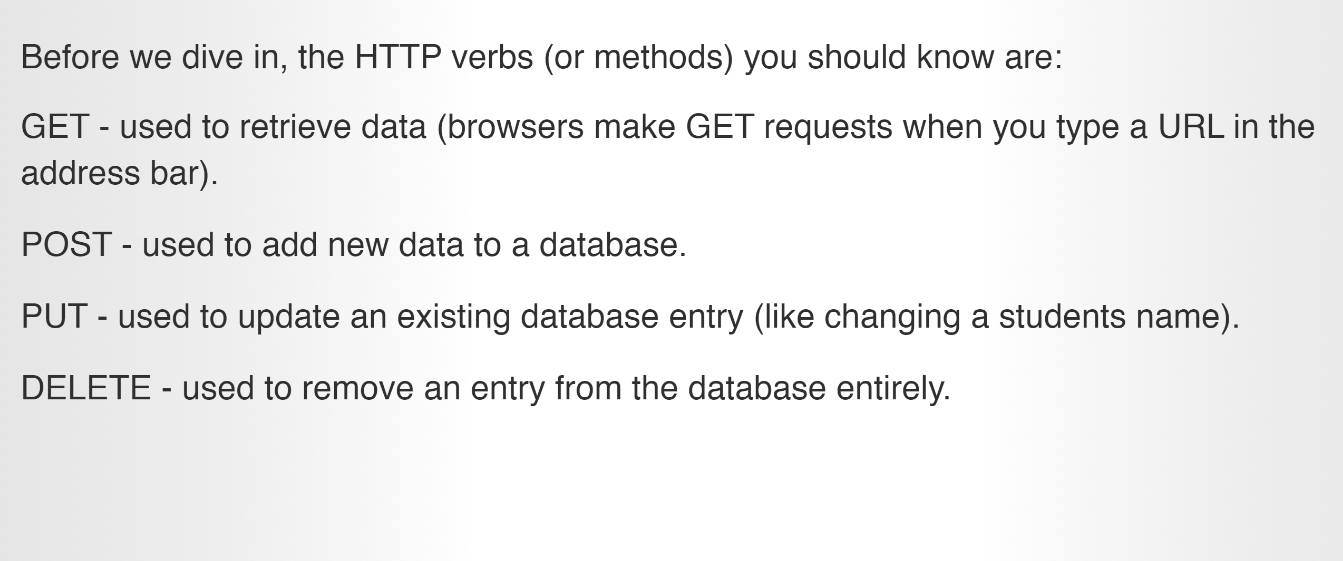
A **resource can be a singleton or a collection**. For example, “customers” is a collection resource and “customer” is a singleton resource (in a banking domain). We can identify “customers” collection resource using the URI “/customers”. We can identify a single “customer” resource using the URI “/customers/{customerId}”.

## <https://restfulapi.net/resource-naming/>



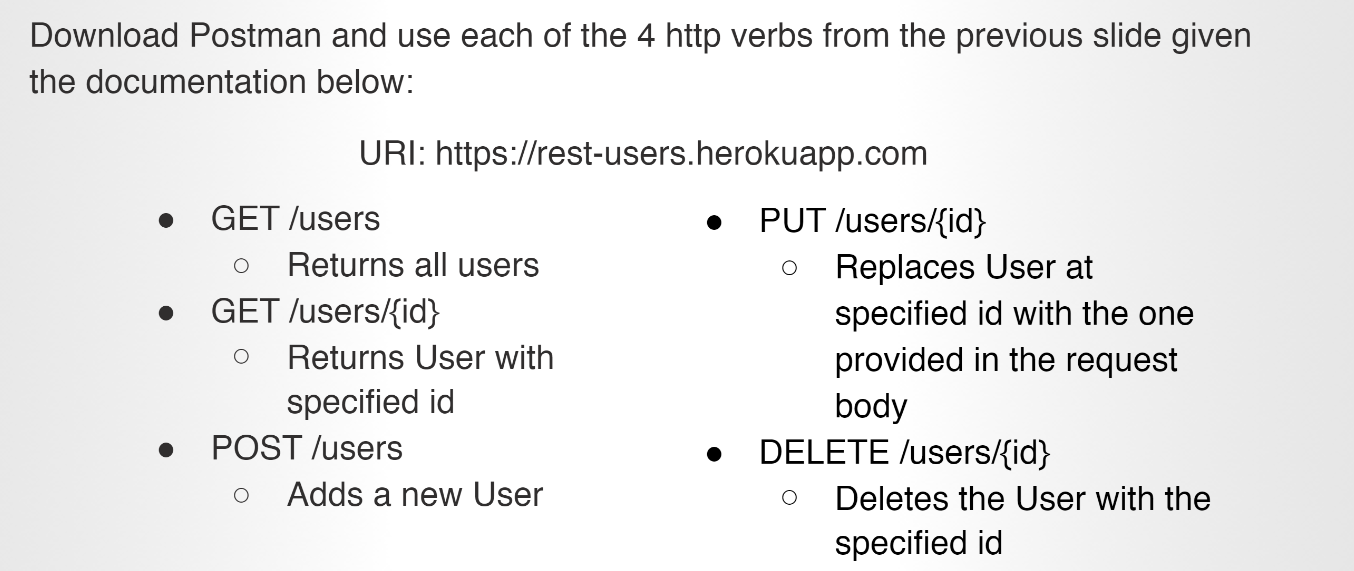






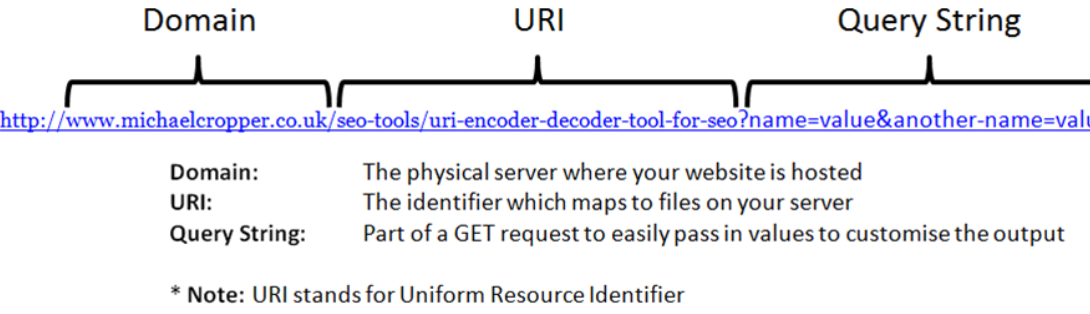
Goto Postman application in windows, use

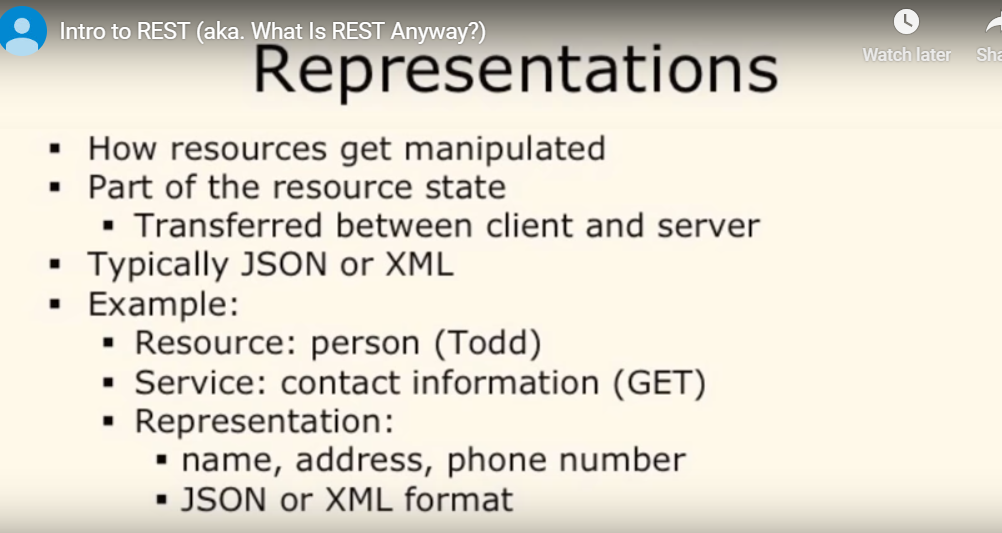
Get to list all user using the api <https://rest-users.herokuapp.com/users/> It will lists all the user in the server



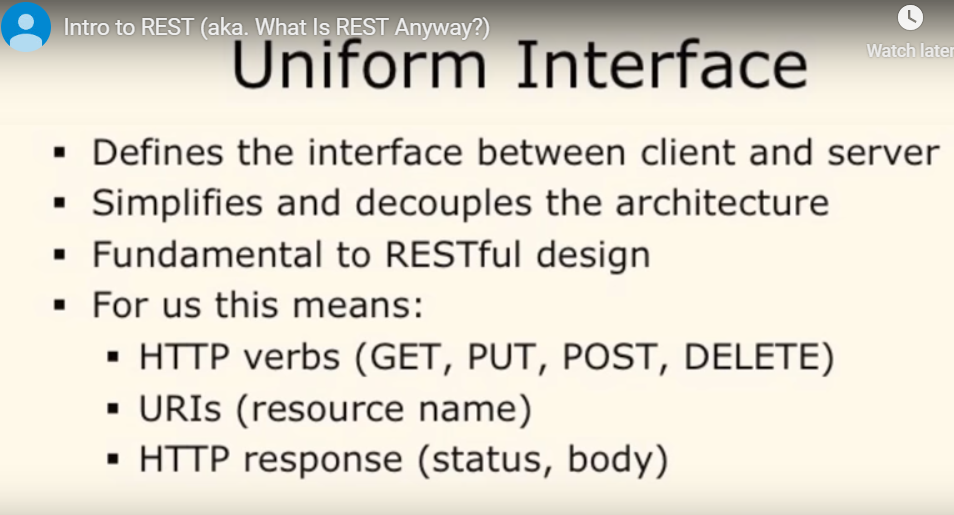
Spring Initializer:

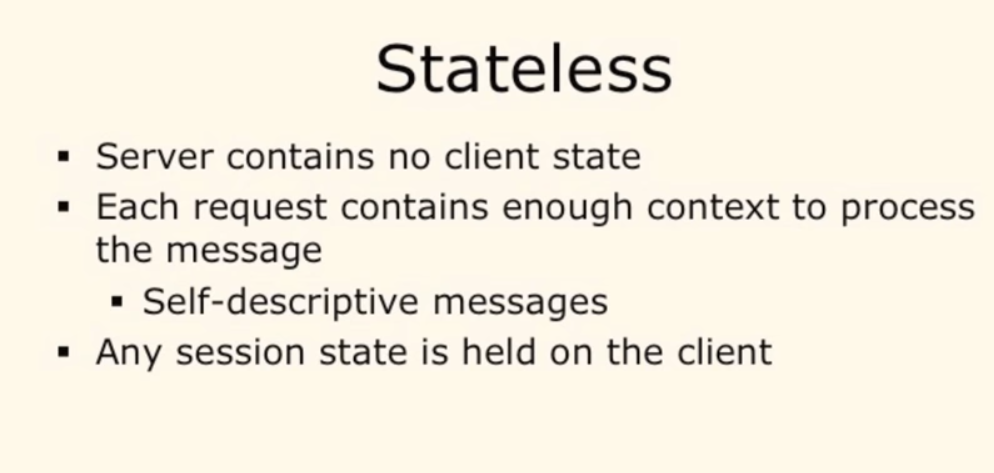
## 

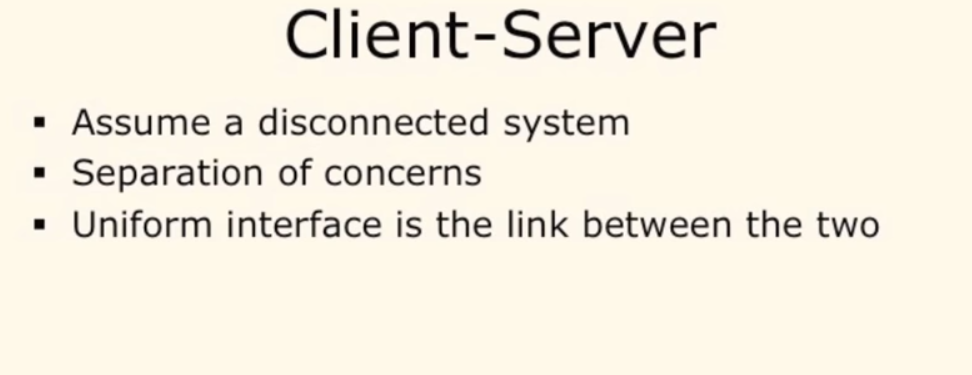




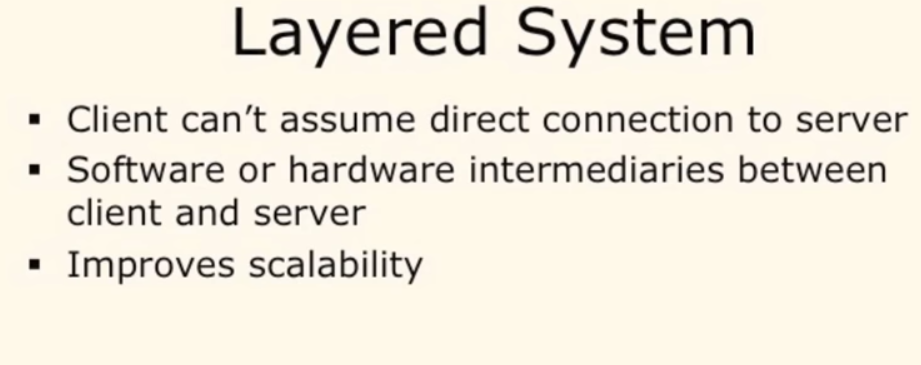
<https://www.restapitutorial.com/lessons/whatisrest.html>

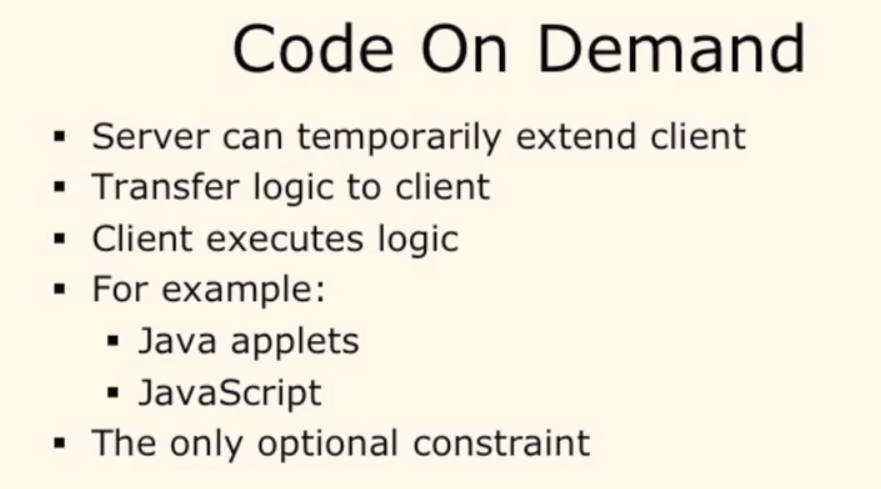


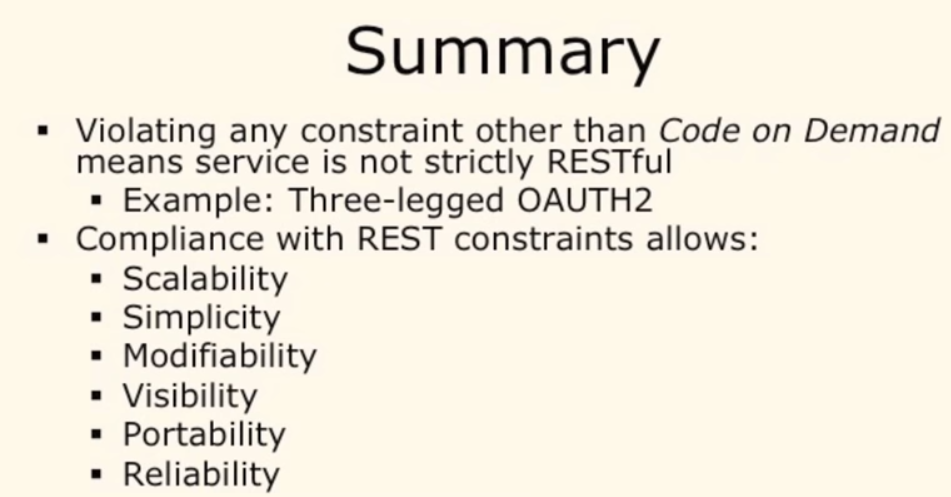












## Use HTTP Verbs to Make Your Requests Mean Something

API consumers are capable of sending GET, POST, PUT, and DELETE verbs, which greatly enhance the clarity of a given request.

Generally, the four primary HTTP verbs are used as follows:

**GET**

Read a specific resource (by an identifier) or a collection of resources.

**PUT**

Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is known before-hand.

**DELETE**

Remove/delete a specific resource by an identifier.

**POST**

Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.

### Note

GET requests must not change any underlying resource data. Measurements and tracking which update data may still occur, but the resource data identified by the URI should not change.

## Provide Sensible Resource Names

Producing a great API is 80% art and 20% science. Creating a URL hierarchy representing sensible resources is the art part. Having sensible resource names (which are just URL paths, such as /customers/12345/orders) improves the clarity of what a given request does.

Appropriate resource names provide context for a service request, increasing understandability of the API. Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications.

Here are some quick-hit rules for URL path (resource name) design:

* Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names.
  + **Good:** /users/12345
  + **Poor:** /api?type=user&id=23
* Leverage the hierarchical nature of the URL to imply structure.
* Design for your clients, not for your data.
* Resource names should be nouns. Avoid verbs as resource names, to improve clarity. Use the HTTP methods to specify the verb portion of the request.
* Use plurals in URL segments to keep your API URIs consistent across all HTTP methods, using the collection metaphor.
  + **Recommended:** /customers/33245/orders/8769/lineitems/1
  + **Not:** /customer/33245/order/8769/lineitem/1
* Avoid using collection verbiage in URLs. For example 'customer\_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer\_list).
* Use lower-case in URL segments, separating words with underscores ('\_') or hyphens ('-'). Some servers ignore case so it's best to be clear.
* Keep URLs as short as possible, with as few segments as makes sense.

## Use HTTP Response Codes to Indicate Status

Response status codes are part of the HTTP specification. There are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes. For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201. A list of valid [HTTP status codes](https://www.restapitutorial.com/httpstatuscodes.html) is available [here](https://www.restapitutorial.com/httpstatuscodes.html) which lists detailed descriptions of each.

Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:

**200 OK**

General success status code. This is the most common code. Used to indicate success.

**201 CREATED**

Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.

**204 NO CONTENT**

Indicates success but nothing is in the response body, often used for DELETE and PUT operations.

**400 BAD REQUEST**

General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

**401 UNAUTHORIZED**

Error code response for missing or invalid authentication token.

**403 FORBIDDEN**

Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).

**404 NOT FOUND**

Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

**405 METHOD NOT ALLOWED**

Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST /users/12345 where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"

**409 CONFLICT**

Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.

**500 INTERNAL SERVER ERROR**

Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

## Offer Both JSON and XML

Favor JSON support unless you're in a highly-standardized and regulated industry that requires XML, schema validation and namespaces, and offer both JSON and XML unless the costs are staggering. Ideally, let consumers switch between formats using the HTTP Accept header, or by just changing an extension from .xml to .json on the URL.

Be aware that as soon as we start talking about XML support, we start talking about schemas for validation, namespaces, etc. Unless required by your industry, avoid supporting all that complexity initially, if ever. JSON is designed to be simple, terse and functional. Make your XML look like that if you can.

In other words, make the XML that is returned more JSON-like — simple and easy to read, without the schema and namespace details present, just data and links. If it ends up being more complex than this, the cost of XML will be staggering. In my experience no one has used XML responses anyway for the last several years, it's just too expensive to consume.

Note that [JSON-Schema](https://json-schema.org/) offers schema-style validation capabilities, if you need that sort of thing.

<https://www.restapitutorial.com/lessons/restquicktips.html>

Spring:

<https://dzone.com/articles/what-is-spring-boot>

## What is REST?

REST stands for [Representational state transfer](http://en.wikipedia.org/wiki/Representational_state_transfer) which essentially refers to a style of web architecture that has many underlying characteristics and governs the behavior of clients and servers.

-> Restricting access

-> Request & response

-> Request: Get/balance - URI

-> respose will be in format of json

Get

Post

Put

Delete

Status: 2 – success

Spring

<https://start.spring.io/>

Using spring, api are written

@RestController  
**public class** FirstLastName {  
  
 @RequestMapping (value=**"/firstLast/{first}/{last}"**, method= RequestMethod.***GET***)  
 **public** String getFirstLast(@PathVariable String first, @PathVariable String last) {  
 **return** first+ **" "**+ last;  
 }  
  
}

Link: [http://localhost:8080/**firstLast/**kk/tp](http://localhost:8080/firstLast/kk/tp)

Here firstLast in link and value in @ RequestMapping should match. This tells the server should invoke this value. When that is invoked, the corresponding function gets invoked which is the getFirstLast is executed –

Later the query parameters that is in the link after

**Controller:Kind of a server listening**

**package** com.company.hellocontroller;  
  
**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
**public class** HelloControllerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(HelloControllerApplication.**class**, args);  
 }  
  
}

URI:

**package** com.company.hellocontroller;  
  
**import** org.springframework.web.bind.annotation.PathVariable;  
**import** org.springframework.web.bind.annotation.RestController;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RequestMethod;  
  
  
  
 @RestController  
 **public class** HelloController {  
  
 @RequestMapping(value=**"/"**, method=RequestMethod.***GET***)  
 **public** String getHello() {  
 **return "hello"**;  
 }  
  
 @RequestMapping(value=**"/name"**, method=RequestMethod.***GET***)  
 **public** String getName() {  
 **return "KK"**;  
 }  
  
 @RequestMapping(value = **"/echo/{input}"**, method = RequestMethod.***GET***)  
 **public** String echoName(@PathVariable String input) {  
 **return** input;  
 }  
 }

* Include getter, setter & constructor –default for sure

@RequestMapping(value = **"/firstLastName/{id}"**, method = RequestMethod.***GET***)

**{id} -🡪 placeholder**

@RequestMapping(value = **"/motorcycle/{id}"**, method = RequestMethod.***GET***)  
**public** String getMotorcycleById(**@PathVariable** **int** id) {

**@pathvariable is important**

**API owner gets to decide what user can see**

* **Pizza owner cannot disclose where he buys the dough. Control access of data**

**HTTP methods – GET POST PUT DELETE – what interaction we want to hava**

**API endpoints: URI: dominos.api.com/order**

**Dominos.api.com/listToppings**

**API- communication between 2 interfaces/entities**

* **How 2 systems interact**



**Postman is used for testing if api is working or not. It is replacing the frontend since we haven’t learnt it**

**When we are giving data to the api in postman, make the body to json format**

**Path - /name/{name}**

**Queryparameters - ?name=”a”&id=1**

* **Cross communication between companies use apis**

@ResponseStatus(value = HttpStatus.***CREATED***) --🡪 will send only success

**Post:**

@RequestMapping(value = **"/listPuppies"**, method = RequestMethod.***POST***)  
@ResponseStatus(value = HttpStatus.***CREATED***)  
**public** PuppyClass[] addPuppy(@RequestBody PuppyClass[] puppyClassArr){  
  
  
 **for** (**int** i =0; i< puppyClassArr.**length**; i++) {  
 puppyClassArr[i].setId();  
 **puppyClassList**.add(puppyClassArr[i]);  
 }  
 **return** puppyClassArr;  
}

**DELETE**

Exception handling:

2 path variables in postman:

[url:8080/color/make/{color}/{make}](url:8080/color/make/%7bcolor%7d/%7bmake%7d)

@Valid

@NotNull – Integer

@NotBlank – String

@NotEmpty

400+ - Client error

Error 422 - unprocessable entity

500+ - Server error

Vnd

@RestControllerAdvice

Table creation in class with camelcase

When table is created in the databse, camel case changes to \_

**spring.jpa.hibernate.ddl-auto**=**create**

Whenever postman run, table is created…and gets deleted & created when it is run again…

So change it to none when it run for the second time

All layers:

