# Week 3/4 Review

## Day 11: Web Addresses

**https://127.0.0.1:3000**

- **protocol:** others - http, ftp
- **ip address:** This is the unique address of a machine on a network.
- **port**:  Number allocated for a specific type of service.

**https://skynet.wecomeinpeace.com**

- **host name**: A physical name assigned to your machine.
- domain name: Defines a specific “region of control” on the internet, also, .com is referred to as the top-level domain name.
- The above URL is an example of a **fully qualified domain name**.

## Day 11: DNS

- **DNS** is an acronym for Domain Name System.

- A **DNS server** is responsible for converting a URL containing human readable domain names to one containing an IP address.

  https://skynet.wecomeinpeace.com → https://127.0.0.1:3000

## Day 11: What is an API ?

- **Application Programming Interface (API)**
- A set of functions and/or procedures designed to interact with an external system.
- Modern cloud architecture relies heavily on API’s.
- **Consuming an API** means interacting with an API’s code to produce a desired result.
- Most modern API's use the **Representational state transfer (REST)** model implemented via the **HTTP Protocol** (more on this next)
    - API's or services that use the REST model are said to be **RESTful Web Services**.

## Day 11: HTTP Protocol

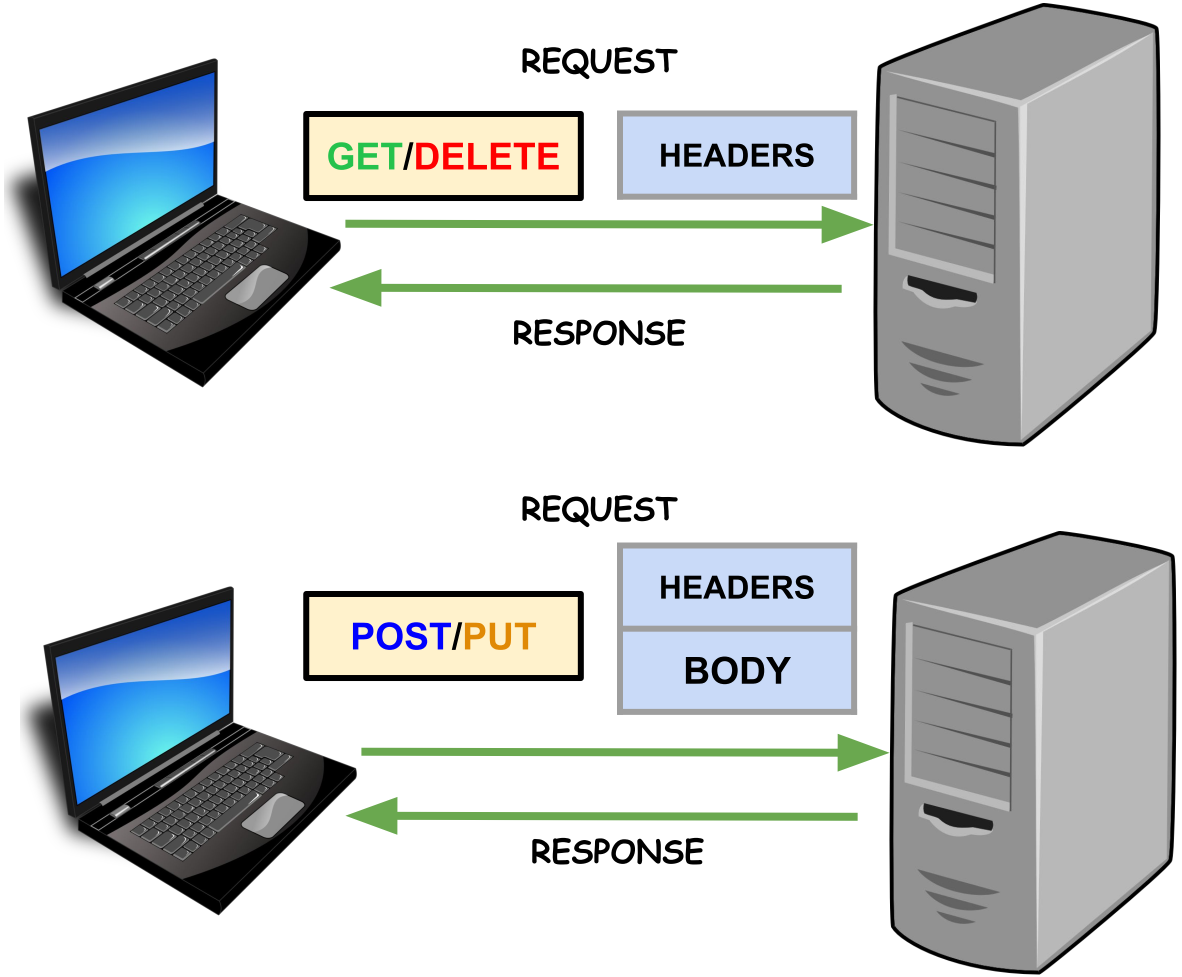
- Request/response protocol
  - Uses request and response messages
- Stateless
- Provides ability to implement authentication
- Resources access via **Universal Resource Locators (URLs)**
  - Implemented using the Uniform Resource Identifiers (URI's) schemes http and https.
- Defines methods to indicate the desired action to be performed on the identified resource.
  - We will use:
    - GET
    - POST
    - PUT
    - DELETE

GET:
Retrieve Resource

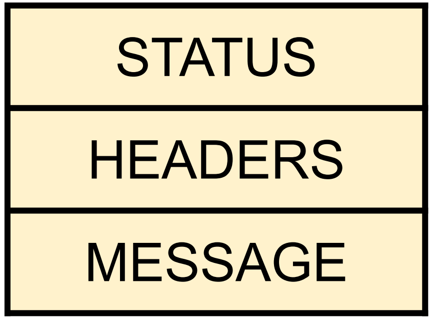
POST:
Create Resource

PUT:
Update Resource

DELETE:
Remove Resource



## HTTP Response



```
HTTP/1.1 200 OK
```

```
Content-Type: application/json; charset=utf-8
```

```
[
    {
        "hotelID": 1,
        "title": "What a great hotel!",
        "review": "It was great!",
        "author": "John Smith",
        "stars": 4
    }
}
```

## Day 11: Possible Responses from HTTP Request

Once a request is made, the REST server can respond with specific status codes:

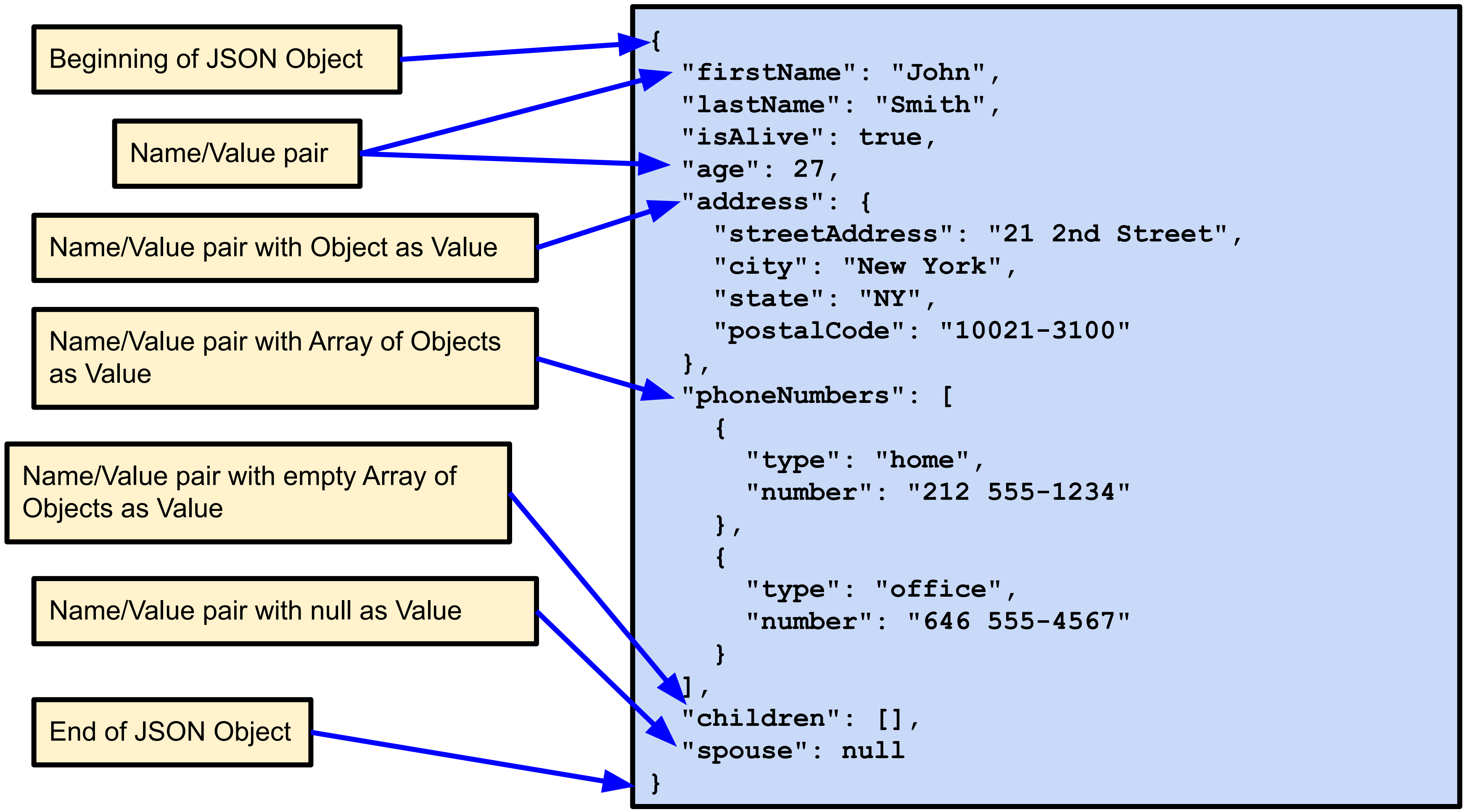
- **2xx**: All’s well, the request was successful.
- **4XX**: The client (you or your application) has not structured the request correctly. Common examples of these are 400 Bad Request and 401 Unauthorized Request.
- **5XX**: The server has encountered some kind of error. The most common of these is the 500 Internal Server Error message.

# Day 11: JavaScript Object Notation (JSON)

- Lightweight data-interchange format
- Easy for humans to read & write
- Easy to parse in code
- Text Only/Language Independent
- Often leveraged for passing data around the internet
- Made up of name/value pairs
- Name is always a quoted string (i.e. "address")
- Name/value separated by :
- Value can be one of several types

# Day 11: JSON Data Types

- **Number**
    - Any number (integer or decimal)
- **String**
    - A sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
- **Boolean**
    - Either of the values **`true`** or **`false`**
- **Array**
    - An ordered list of zero or more values, each of which may be of any type. Arrays use **`[]`** square bracket notation with comma-separated elements.
- **Object**
    - A collection of name–value pairs where the names (also called keys) are strings. Objects are delimited with **`{}`** curly braces and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.
- **`null`**


Beginning of JSON Object
Name/Value pair
Name/Value pair with Object as Value
Name/Value pair with Array of Objects as Value
Name/Value pair with empty Array of Objects as Value
Name/Value pair with null as Value
End of JSON Object

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

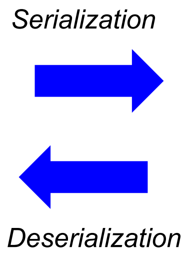
# Day 11: Serialization & Deserialization

Java Objects can easily be converted to JSON data and vice versa, Mapping from an Object to JSON is known as **serialization** of an Object. Mapping from JSON to an Object is known as **deserialization**.

Java

```
Review[];

public class Review {
    private int hotelID;
    private String title;
    private String review;
    private String author;
    private int stars;
}
```



JSON

```
[
    {
        "hotelID": 1,
        "title": "What a great hotel!",
        "review": "Great hotel,
        "author": "John Smith",
        "stars": 4
    },
    {
        "hotelID": 1,
        "title": "Peaceful night sleep",
        "review": "Would stay again",
        "author": "Kerry Gold",
        "stars": 3
    }
]
```

## Day 11: Making a GET Request using Java

The **RestTemplate** class provides the means with which we can make a request to an API. Here is an example call:

```
RestTemplate restTemplate = new RestTemplate(); // Create a new client
ResponseEntity response = restTemplate.getForEntity(
                "https://api.exchangerate-api.com/v4/latest/USD",
                String.class); // Make GET request using Client
System.out.println(response.getBody()); // your return data returned from .getBody()
```

**`response.getBody()`** returns a **`String`** representation of the JSON, just like if you saw the API response in your browser. In the next slide, you'll see how RestTemplate can automatically convert the response data into a Java Object.

## Day 11: Making a GET Request with the Result Mapped to a Java Object

The RestTemplate class provides a way to make a request to an API and treat the result as a Java Object. The JSON return by the API call will be mapped to the specified Java class. Here is an example call:

```
private static final String API_BASE_URL = "http://helpful-site/v1/api/data";
private static RestTemplate restTemplate = new RestTemplate();
MyObj myobj = restTemplate.getForObject(API_BASE_URL, MyObj.class);
```

Note that we can specify the return type of the API call with the second parameter (MyObj.class). Alternatively, if you are getting an array of objects back, we can write the following:

```
MyObj [ ] myobj = restTemplate.getForObject(API_BASE_URL, MyObj[ ].class);
```

## Day 12: More Request Types

In the last lecture we saw GET’s, which simply read the data. Today we will deal with request types that might potentially change the application’s data permanently:

- **POST**: Ideally suited for inserting new data into the data source.
- **PUT**: Ideally suited for updating an existing record within a data source.
- **DELETE**: Ideally suited for removing an existing record from the data source.

For the POST & PUT requests we are converting an object to data

# Day 12: Implementing POST Requests

***POST: http://localhost:3000/hotels/{id}/reservations***

Create HTTP Headers for POST

Set the **`content-type`** for JSON

Create an **`HttpEntity`**, which allows us to combine headers and body

Call **`postForObject`** with the **`HttpEntity`** and class to post for (Reservation)

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

## Day 12: Implementing PUT Requests

- PUT requests are similar to POST requests in that they usually have both headers and a payload contained in the message body.
  - We can write code for a PUT request much like our POST code but using the put method rather than postForObject method.

```
// Create instance of RestTemplate
RestTemplate restTemplate = new RestTemplate();
// Create instance of HttpHeaders and set Content-Type to application/json
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
// Combine headers with existing user object to form HttpEntity
HttpEntity<User> entity = new HttpEntity<>(newUser, headers);

// Put (update) the existing user using the entity.
restTemplate.put(API_URL + "users/23", entity);
```

## Day 12: Implementing DELETE Requests

- DELETE requests are similar to GET requestS In that they usually have only headers and not a payload contained in the message body.
  - We can write code for a DELETE request much like our GET code but using the delete method rather than getForObject method.

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.delete(API_URL + "users/23");
```

## Day 12: Exceptions and Error Handling

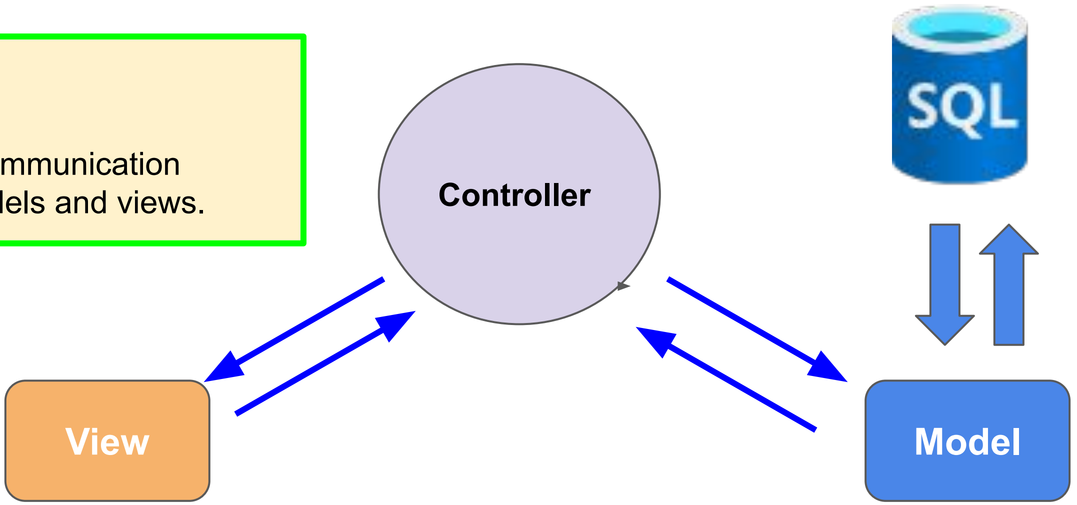
There are 2 exceptions to be aware of when dealing with APIs:

- **`RestClientResponseException`** - is thrown when a status code other than a 2XX is returned.
    - Can check status code via this Exception's **`getRawStatusCode()`** method
    - Can get  text description  of the status code (i.e. Not Found for 404) from this Exception's **`getStatusText()`** method
- **`ResourceAccessException`** - is thrown when there was a network issue that prevented a successful call.

# Day 13: Model View Controller (MVC) Pattern in APIs

**Controller:**:

- Facilitates communication between models and views.



**Model**:

Represents app data

- Properties
- Business logic
- Validation rules

**View**:

- JSON data which can be used by calling applications.

## Day 13: Request and response lifecycle

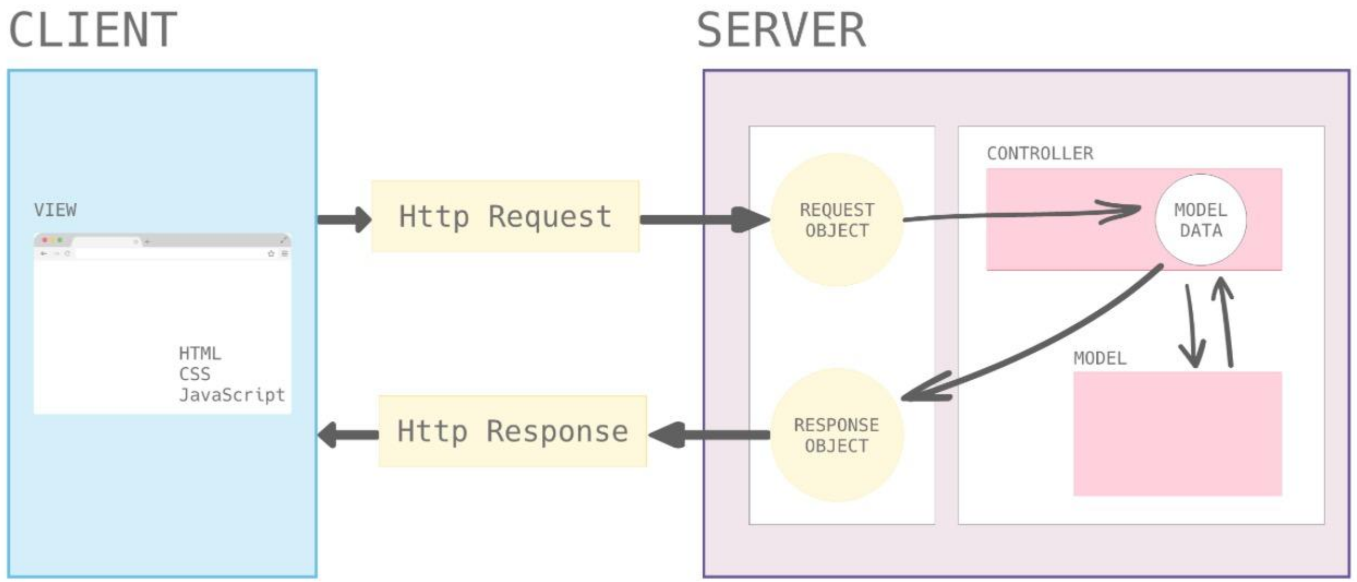


*Image - The flow and handling of HTTP packets within an MVC application*

# Day 13: REST Controllers

The **@RestController** annotation tells Spring that this will be a controller class that can accept and return data.

```
@RestController
public class TodoController {

    @RequestMapping(path = "/todo", method = RequestMethod.GET)
    public List<String> getTodos() {
        return todos;
    }

    @RequestMapping(path = "/todo", method = RequestMethod.POST)
    public void addTodo(String task) {
        if (task != null) {
            todos.add(task);
        }
    }

}
```

## Day 13: REST Controllers

The **`@RequestMapping`** defines a method as being an endpoint.

Note that the same path can have different meaning if there are different HTTP methods for the same path (i.e. GET and POST map to different methods even though the path is the same.

```java
@RestController
public class TodoController {

    @RequestMapping(path = "/todo", method = RequestMethod.GET)
    public List<String> getTodos() {
        return todos;
    }

    @RequestMapping(path = "/todo", method = RequestMethod.POST)
    public void addTodo(String task) {
        if (task != null) {
            todos.add(task);
        }
    }

}
```

## Day 13: @RequestMapping Parameters

The **`@RequestMapping`** annotation has several arguments that define which handler method responds to a given web request.

- **`@RequestMapping(path=)`**
    - The **`path=`** parameter maps a request path to the annotated method.
- **`@RequestMapping(method=)`**
    - **`method=`** allows you to define a specific HTTP verb (method) for the request mapping (GET, POST, PUT, DELETE)
    - If not defined, *ANY* verb will map to the annotated method.

## Day 13: Model Binding

**Model Binding** provides us ways of mapping the various ways that data may be passed in an HTTP request to data types we can use in our Controller methods.

We'll look at how to work with each of these HTTP requests data passing methods:

- Query Parameters
- Path Variables
- Request Body

`

○

## Day 13: Request parameters

There are times when you'll want to pass some information along to your API as part of the request. Imagine that you created a method in your Todo API where someone could retrieve all todos based on the completed status. You might use a query parameter named **`filter`** with a value of completed. It would look like this:

```
# request parameter example
https://localhost:8080/todo?filter=completed
```

We can use the **`@RequestParam`** annotation to map the **`filter`** query param to a **`String`** param called **`filter`** in our handler method:

```
@RequestMapping("/todo")
public List<Todo> getCompletedTodos(@RequestParam String filter) {
```

## Day 13: Request parameters

We can specify a default value for a **`@RequestParam`** by using the **`defaultValue=`** param of **`@RequestParam`**.

- Note that we can remove the **`required=false`** param because there will now always be a value for the **`@RequestParam`**.
- Also note that in this example we use quotes around the value even though it is a **`defaultValue`** for an **`Integer`**. This is because **`defaultValue`** is implemented as a **`String`**. Spring will convert this value to the appropriate data type - **`Integer`** in this case,

```
@RequestMapping("/todo")
public List<Todo> getCompletedTodos(@RequestParam String filter.
    @RequestParam(defaultValue="0") Integer limit) {
```

# Day 13: Path Variables

If you wanted to find a specific todo by its ID, you could use a query parameter for that—like **`/todo?id=1`**—but a popular REST convention is to use the ID as a part of the path:

```
# request path variable example
https://localhost:8080/todo/2
```

We can add a placeholder for the id (or other variable) by adding **`{id}`** in the path where the id value would go and then annotating a param with the same name in our method with the **`@PathVariable`** annotation. We can use **`required=false`** with this annotation as well:

```
@RequestMapping(path = "/todo/{id}/{name}", method = RequestMethod.GET)
public Todo getTodo(@PathVariable Integer id,
     @PathVariable(required = false) String name) {
```

## Day 13: Request Body

When using the **`POST`** or **`PUT`** methods, we can bind the request body (payload) to a model that matches the payload by using the **`@RequestBody`** annotation. The code below would take JSON data representing a **`Hotel`** object  and deserialize it into a Java **`Hotel`** object which is declared as a param type of a param to our handler

```
@RequestMapping( path = "/hotels", method = RequestMethod.POST)
public void addHotek(@RequestBody Hotel newHotel) {
```

Note that you may only annotate one handler param with @RequestBody since there isn't a way to POST multiple request bodies in a request.

# Day 14: Dependency Injection

**Dependency Injection** is where instances of classes that are depended on are injected into a new object rather than created by that object. This further decouples the classes from each other and allows the controller to be completely independent from any implementation of the DAO interface.

In Spring, if you want to inject one class into another, you add an annotation to make Spring aware of the class. For the DAOs, you'd add an `@Component` annotation:

```
@Component
public class MemoryHotelDao implements HotelDao {
    ...
}
```

## Day 14: Dependency Injection

Now, the MemoryHotelDao class can be injected into other classes, like the controller. To inject the DAO into the controlller, you declare a constructor parameter of the class you depend on in the controller like this:

```
private HotelDao dao;

public HotelController(HotelDao dao) {
    this.dao = dao
}
```

Now, when Spring creates the controller, an instance of the DAO class with the **`@Component`** annotation is also created and injected into the controller for you.

## Day 14: Server-Side Data Validation

It's important to handle data validation in back-end code and front-end code. Validation in Java is done using a Java standard called **Bean Validations**.

Bean Validations are annotations that are added to Java model classes and verify that the data in objects match a certain set of criteria. Here's an example:

```java
public class Product {
    @NotBlank( message="Product name cannot be blank" )
    private String name;

    @Positive( message="Product price cannot be negative" )
    private BigDecimal price;

    @Size( min=20, message="Description cannot be less than twenty characters" )
    private String description;

    /*** Getters and Setters ***/
}
```

## Day 14: Server-Side Data Validation: Bean Validations

- `@NotNull`
- `@NotEmpty`
- `@NotBlank`
- `@Min`
- `@Max`
- `@DecimalMin`
- `@Size`
- `@Past`
- `@Future`
- `@Pattern`

## Day 14: Server-Side Data Validation: @Valid

The validations that you add to your model objects aren't automatically checked for you. To check them in your Controller, you need to add the **`@Valid`** annotation to the model:

```
@RequestMapping(path = "", method = RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void create(@Valid @RequestBody Product product) {
    // data validation
    productsDao.add(product);
}
```

If the validation fails, meaning that the data supplied from the request doesn't pass the validation tests you defined in the model, then the response returns a status code of **`400 Bad Request`**, and the a JSON object containing info about the validation error is returned to the client.

## Day 14: More About REST

**REST** stands for **Representational State Transfer** and is a series of guidelines for defining Web Services that are flexible and usable by a wide range of services.

- Uses all the standard web technologies you've already learned:
    - HTTP
    - URLs
    - JSON

- Building web services off of already widely adopted standards, REST makes it easy to tie APIs into existing applications.

## Day 14: More About REST

REST APIs are based on the concept of **resources** and building addresses (in the form of URLs) and actions (in the form of HTTP methods) on those resources. It also uses HTTP Status Codes to alert the API users about the results of the call

Resources are the objects defined in the application (i.e. Hotel, Reservation). You access resources through URLs, which specify the resource and, depending on which HTTP method you use, retrieve and modify them.

## Day 14: Addressing Resources

| Goal | URL |
|---|---|
| Access a single resource | /products/342333 |
| Access a resource that belongs to another resource | /products/342333/reviews |
| Access a particular review for a given resource | /products/342333/reviews/5674 |

## Day 14: Handling PUT Requests

```java
/**
 * Updates a product based on the ID and the request body
 *
 * @param product the updated product
 * @param id the id of the product that is getting updated
 */
@RequestMapping( path = "/products/{id}", method = RequestMethod.PUT )
public void update(@RequestBody Product product, @PathVariable int id) {
    product.setId(id);

    // Update product in underlying datastore
}
```

## Day 14: Handling DELETE Requests

```
/**
 * Removes a product based on the ID
 *
 * @param id the ID of the product to remove
 */
@RequestMapping( path = "/products/{id}", method = RequestMethod.DELETE )
public void delete(@PathVariable int id) {

    // Removes the product in underlying datastore

}
```

## Day 14: HTTP Status Codes

- **200 OK**
- **201 Created**
- **204 No Content**
- **400 Bad Request**
- **401 Unauthorized**
- **403 Forbidden**
- **404 Not Found**
- **405 Method Not Allowed**
- **500 Internal Server Error**

## Day 14: Returning Meaningful Status Codes

Normally, the correct status code is returned by default, but there may also be times when you want to return a different status code than the default. For instance, by default the status code **`200 OK`** is returned for a successful **`DELETE`**. However, REST suggests returning status code **`204 No Content`** on a successful **`DELETE`**. This is accomplished by adding the **`@ResponseStatus(value = HttpStatus.NO_CONTENT)`** annotation:

```
@RequestMapping( path = "/products/{id}", method = RequestMethod.DELETE )
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void delete(@PathVariable int id) {

    // Remove the product from underlying datastore

}
```

## Day 14: Returning Meaningful Status Codes

There may also be times when you want to return a different status code than the default due to an error. For example, if a user wanted to update the product with an ID of 13 and that ID wasn't in the database, you'd want to return a **`404 Not Found`** status code.

You can do that by setting up a special **`Exception`** that's linked to that status code with a **`@ResponseStatus`** annotation:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ProductNotFoundException extends Exception {}
```

## Day 16: Authentication vs. Authorization

**Authentication** is the act of validating that users are whom they claim to be. This is the first step in any security process.

Common forms of authentication:

- Username/Password
- One-time pins
- Authentication apps.
- Biometrics

**Authorization** is the process of giving the user permission to access a specific resource or function.

## Day 16: HTTP Status Codes: Authentication & Authorization

The HTTP Status Codes associate with authentication and authorization fall into the 4xx (client-related) group of statuses.

- **401 Unauthorized** indicates that the user has not been authenticated
- **403 Forbidden** indicates that the user is authenticated, but is not allowed to access the resource specified in the HTTP request.

## Day 16: Java Web Token (JWT)

- Compact size allows for quick transfer with requests.
- Often used as authorization mechanism, storing user info such as permissions or roles in payload. These are called **claims**.
- Can contain any data that can be represented in JSON.
- JWT actually contains JSON, but it's encoded.

## Day 16: Setting Server Authorization Rules

- **Authorization** is the process of giving a user permission to access a specific resource or function.
- Can define rule for each resource including
    - **`@PreAuthorize("permitAll")`**
    - **`@PreAuthorize("isAuthenticated()")`**

## Day 16: Role-Based Authorization

- It's not uncommon to have certain resources or functions only be accessible to certain people or roles.
- We can authorize access by the user's role (i.e. admin user can access but non-admin cannot)
  - **`@PreAuthorize("hasRole('ADMIN')")`**
  - **`@PreAuthorize("hasAnyRole('USER', 'ADMIN')")`**

## Day 16: Getting Info About the Current User

There are times where you'll need access to the current logged in user. You have secured the **`delete()`** method to users with the role **`ADMIN`**, but what if you wanted to keep an audit log of which user deleted each reservation?

Spring gives you access to information about the current user if you add an argument of type **`Principal`** to your method.

When you annotate a method with **`@RequestMapping`**, that method has a flexible signature, and you can choose from a range of supported controller method arguments.

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-methods.

## Day 16: The exchange Method

```
LoginDTO loginDTO = new LoginDTO(credentials);
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
HttpEntity<LoginDTO> entity = new HttpEntity<>(loginDTO, headers);
ResponseEntity<Map> response = null;

response = restTemplate.exchange(BASE_URL + "/login",
    HttpMethod.POST, entity, Map.class);
```

**`exchange`** returns a **`ResponseEntity`** of a specified type, rather than an object of the actual type.

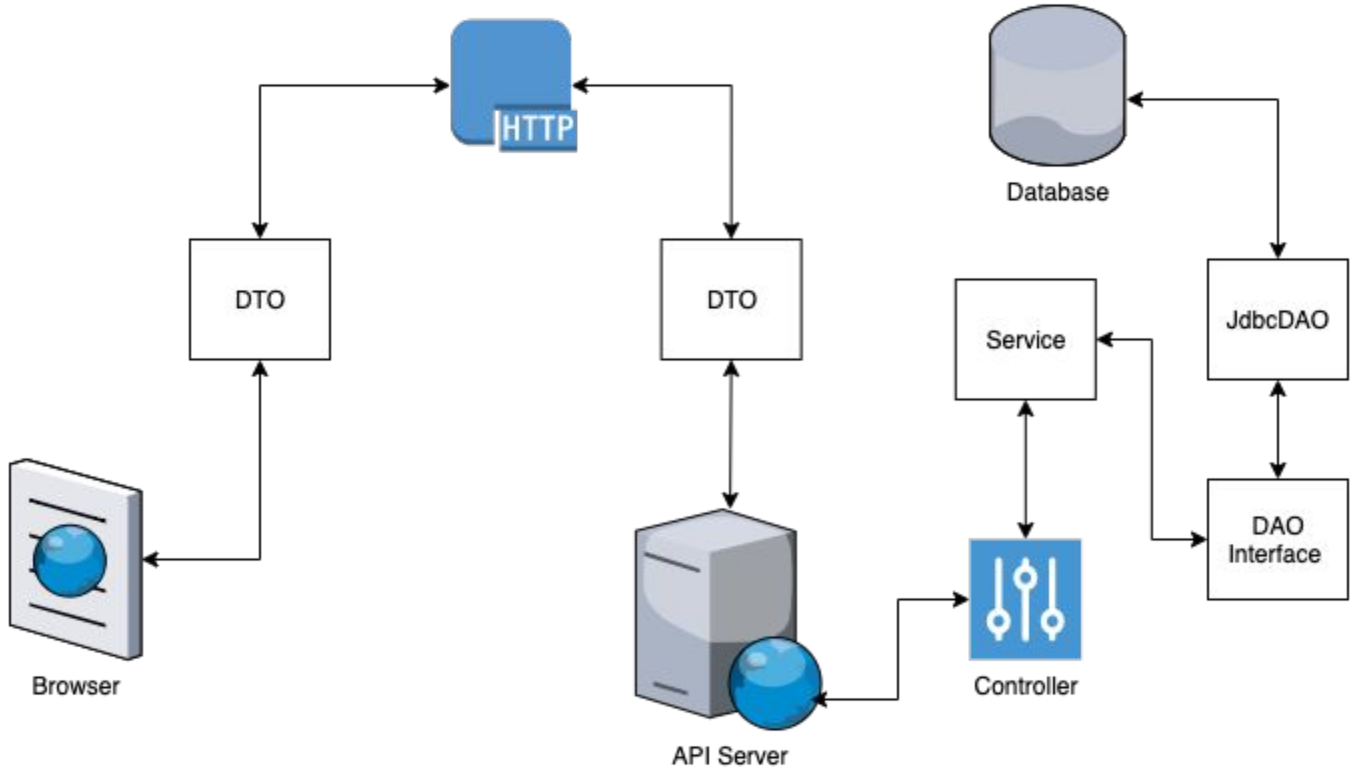
## Day 16: The exchange Method

```java
private HttpEntity makeAuthEntity() {
    HttpHeaders headers = new HttpHeaders();
    headers.setBearerAuth(AUTH_TOKEN);
    HttpEntity entity = new HttpEntity<>(headers);
    return entity;
}

hotels = restTemplate.exchange(BASE_URL + "hotels",
    HttpMethod.GET, makeAuthEntity(), Hotel[].class).getBody();
```

Using **`exchange`** here allows us to attach an **`HttpEntity`** with **`AUTH_TOKEN`** in **`Bearer Auth`** header even though **`GET`** has no body to attach headers to in the **`HttpEntity`**

## Flow Overview


HTTP
Database
DTO
DTO
Service
JdbcDAO
DAO Interface
Browser
Controller
API Server