Server-Side APIs: Part 1

What is MVC?

Applications can become quite large. As a result, it often becomes difficult to manage their size and complexity as new features emerge or existing requirements change. To address this, software developers rely on design patterns that assist in keeping the code clean and maintainable. One such pattern is called **Model-View-Controller (MVC)**.

MVC is a design pattern that separates an application into three main components:

- Model
- View
- Controller

The MVC pattern promotes loose coupling by helping to create applications so that the logic across web applications can be reused while not allowing any particular part to do too much.

Model View Controller (MVC) Pattern

Controller:

- Facilitates communication between models and views.

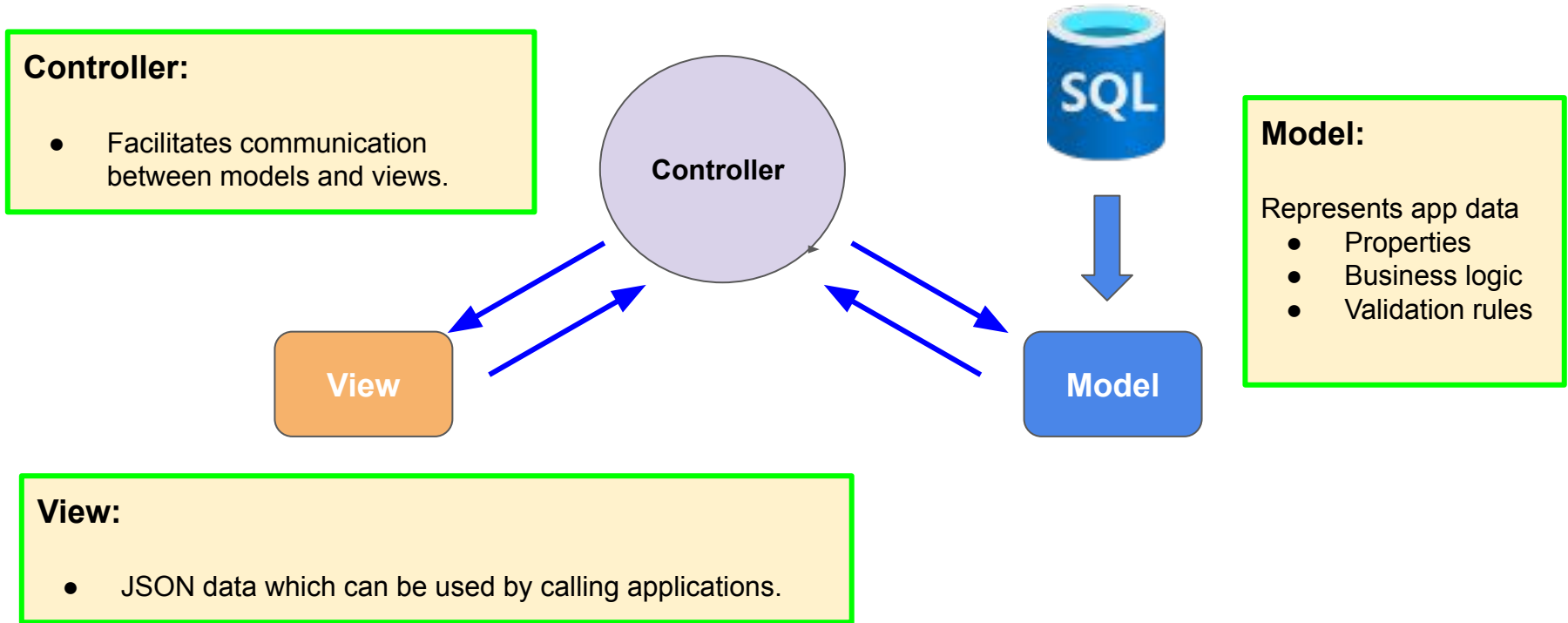**Controller**

View

Model

SQL
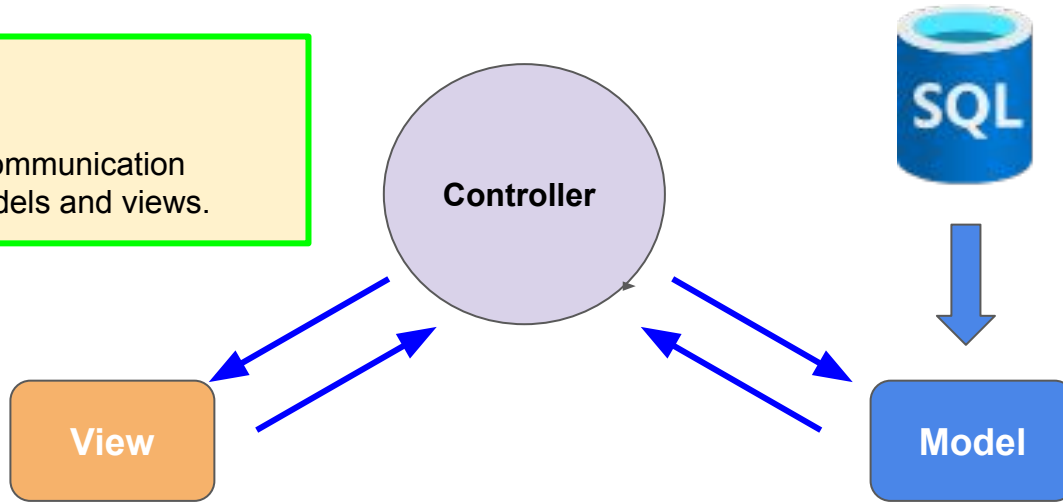
Model:

Represents app data
- Properties
- Business logic
- Validation rules

View:

- Displays the user interface
- Template to present model data

Model View Controller (MVC) Pattern in APIs

Controller:

- Facilitates communication between models and views.

Controller

SQL

Model:

Represents app data
- Properties
- Business logic
- Validation rules

View

Model

View:

- JSON data which can be used by calling applications.

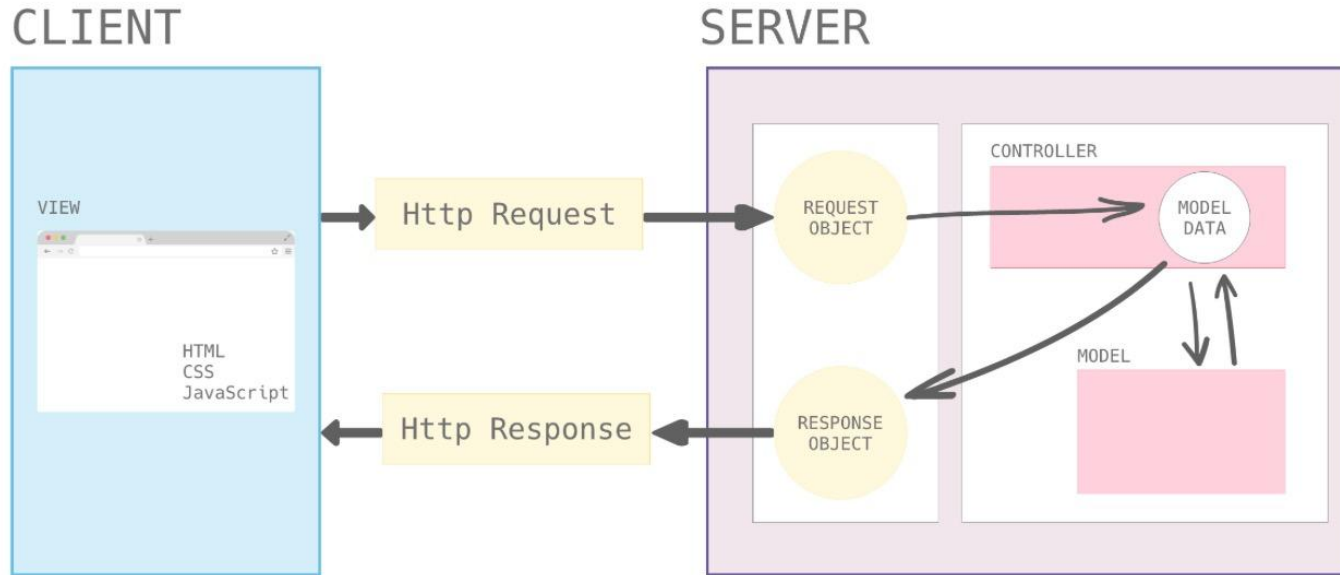# Request and response lifecycle

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

REST Controllers

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

```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);
        }
    }

}
```
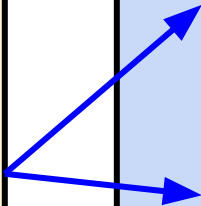
REST Controllers

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);
        }
    }

}
```

@Request Mapping 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.

Models and Content Negotiation

The Data Transfer Objects (DTOs) we discussed previously act as the models in Java code.

Content negotiation is an HTTP mechanism that's used for serving different formats of a resource at the same URL. The client specifies which formats it can understand in the `Accept` header of the HTTP request. JSON is the default format for Spring MVC.

Spring MVC serializes the Java return data into JSON data which is returned to the calling client code. For instance, the List of Strings returned by the method is a JSON array of strings:

```
@RequestMapping(path = "/todo", method = RequestMethod.GET)
public List<String> getTodos()
```

```
['Wake up', 'Shower', 'Drive to work'];
```

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

`

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.

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

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) {
```

Request parameters

We can do this with multiple query params as well:

```
# request parameter example
https://localhost:8080/todo?filter=completed&limit=5
```



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

We can mark a `@RequestParam` as optional by adding the `required=false` modifier:

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

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) {
```

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.