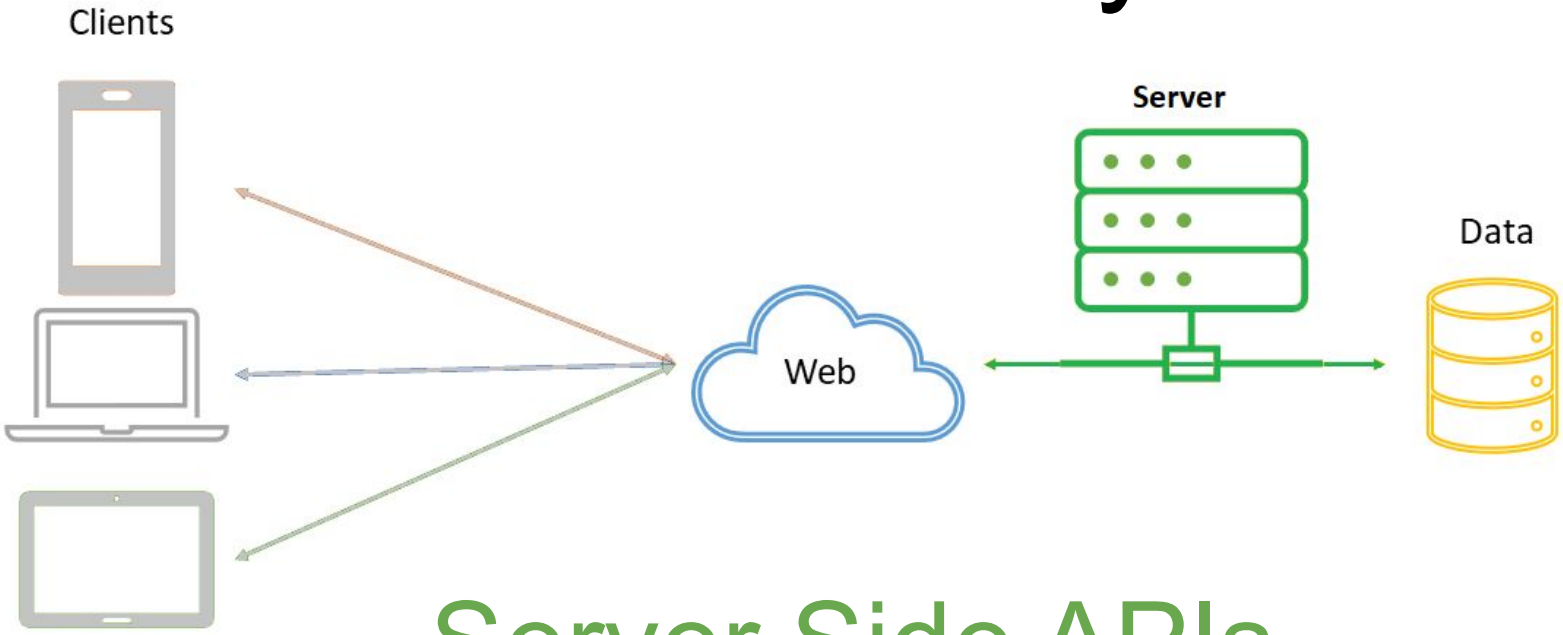
Module 2 Day 13



Server Side APIs

Part 1

Module 2 Day 13
Can you ?

1. Describe the differences and responsibilities of backend code versus frontend code
2. Describe the MVC pattern and why programmers use it Study LMS
3. Describe the purpose and scope of the Spring Boot
4. Create a ***RESTful*** web ***Application Programming Interface*** that accepts GET and POST requests and returns JSON
5. Run a web API on a local development machine
6. Connect a local client application to a locally running API

- and -

… Have you read the "Server Side APIs" Chapters in the Text Book?

The MVC Pattern

Advantages of MVC Architecture in Java

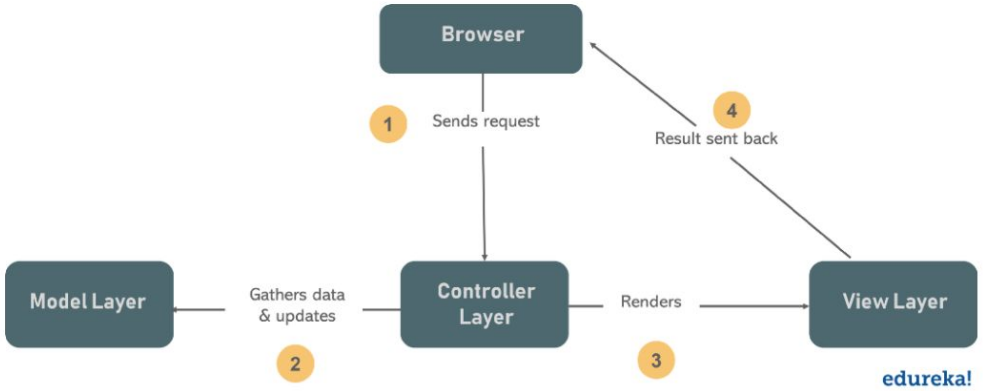
MVC architecture offers a lot of advantages for a programmer when developing applications, which include:

- Multiple developers can work with the three layers (Model, View, and Controller) simultaneously
- Offers improved *scalability*, that supplements the ability of the application to grow
- As components have a low dependency on each other, they are easy to maintain
- A model can be reused by multiple views which provides reusability of code
- Adoption of MVC makes an application more expressive and easy to understand
- Extending and testing of the application becomes easy

What is MVC Architecture in Java?

Model designs based on MVC architecture follow the MVC design pattern and they separate the application logic from the user interface when designing software. As the name implies MVC pattern has three layers, which are:

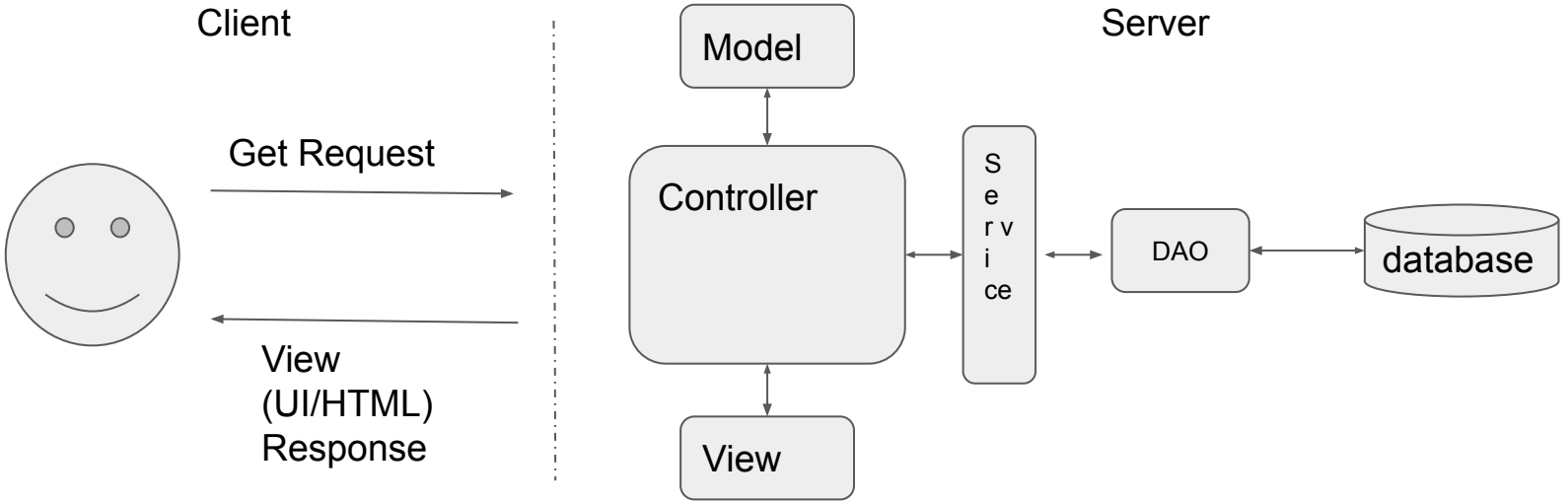
- ***Model*** — Represents the business layer of the application
- ***View*** — Defines the presentation of the application
- ***Controller*** — Manages the flow of the application



Swatee Chand 8/2/2019 https://medium.com/edureka/mvc-architecture-in-java-a85952ae2684

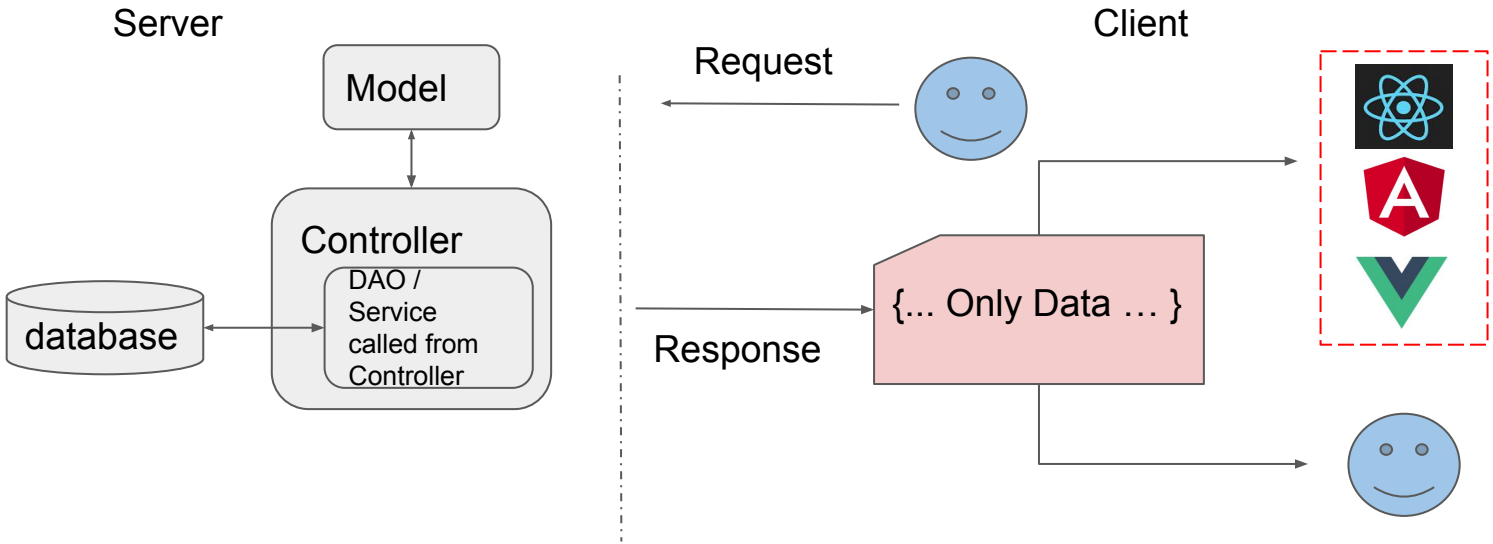
API's as a source of data

After a controller receives a GET request in a traditional MVC Application, the user is presented with a view… the "website":



API's as a source of data

Data API controllers only return data (JSON) as the View instead of returning an HTML document.



This data can now be sent and used by JS applications.

Building these "front end" apps is the focus of Module 3!

… or just for a user to analyze the raw data.
(analysts and data scientists?)

REST Controllers

- In order to send back data instead of a view, we need to change our controllers to REST controllers.
  - REST is short for **Re**presentational **S**tate **T**ransfer.

- Spring makes this easy using annotations. The **@RestController** annotation marks a class as a controller for an API.

```
@RestController

public class ProductReviewsController {
… }
```

JSON Review

- The data generated by the controllers are in JSON format
- JSON has three basic rules:
  - Objects in JSON are delimited by curly braces { … }
  - Arrays in JSON are delimited by brackets [...]
  - Data is listed in key-value pairs (key : value)

JSON Review: Example

Here is an example of JSON data:

```
{
        firstName: "John",
        lastName: "Smith",
        age: 40,
        lang: ["English", "Spanish", "Esperanto"]
}
```

https://www.w3schools.com/js/js_json_intro.asp

- The object itself is enclosed with a set of curly braces.
- Each property of the object is listed as a key value pair.
- An array is enclosed with square brackets.

Structuring an endpoint (GET)

This is a request mapping that takes care of GET requests, that respond to the /hotels endpoint.

```
@RequestMapping(path = "/hotels", method = RequestMethod.GET)
public List<Hotel> list() {
    return hotelDAO.list();
}
```

This is a request mapping that takes care of GET requests sent to the /hotels endpoint with a path variable {id}. An actual call to this endpoint would look something like “/hotels/5”. The parameter ***id*** is populated with the 5 from the path, it is a @PathVariable.

```
@RequestMapping(path = "/hotels/{id}", method = RequestMethod.GET)
public Hotel get(@PathVariable int id) {
    return hotelDAO.get(id);
}
```

Structuring an endpoint (GET) with parameters

We can also specify that our GET endpoints take parameters as opposed to fixed resource identifiers:

```
@RequestMapping(path = "/hotels/filter", method = RequestMethod.GET)
 public List<Hotel> filterByStateAndCity(@RequestParam String state, @RequestParam(required = false) String city) {
…
}
```

The above mapping specifies that the get request can take two parameters, one for state and the other for city, with the latter being optional. An acceptable call to this endpoint could be:

/hotels/filter?**state**=Ohio&**city**=Cleveland

Inside the method, having made the above call, the String state will take on the value of Ohio, and the String city will take on the value of Cleveland.

Structuring an endpoint (POST)

POST requests require a body which contains the data we are sending to the endpoint.

```
@RequestMapping( path = "/hotels/{id}/reservations", method = RequestMethod.POST)
public Reservation addReservation(@RequestBody Reservation reservation, @PathVariable("id") int hotelID) {
    return reservationDAO.create(reservation, hotelID);
}
```

The above mapping handles a POST request which takes a body that can be deserialized into a Reservation object.

@RequestBody deserialization

Java Class

```
public class Reservation {

    private int id;
    private int hotelID;
    private String fullName;
    private String checkinDate;
    private String checkoutDate;
    private int guests;

// … getters + setters + other methods
}
```

Request Body (in JSON)

```
{
    "id": 5,
    "hotelID": 5,
    "fullName": "Jane Smith",
    "checkinDate": "2020-06-09",
    "checkoutDate": "2020-06-12",
    "guests": 4
}
```

The valid request sent to the endpoint should have a body in the form of JSON, the @RequestBody annotation will deserialize it into an object of the specified class.

Let’s implement some GETs and POSTs

Consuming GETs and POSTs

- In the first two lectures we talked about how to consume an API from a Java Application using a RestTemplate object.
    - At that time, we had to use a “simulated API” courtesy of JSON Server.
- We now know how to build the real thing, and can set aside the mock JSON Server!
- Let’s review how ***OUR*** API could be consumed ...

Consuming a GET

This is the GET endpoint with a path variable that we defined today:

```
@RequestMapping(path = "/hotels/{id}", method = RequestMethod.GET)
public Hotel get(@PathVariable int id) {
    return hotelDAO.get(id);
}
```

… and this is how a different Java application can consume it:

```
hotel = restTemplate.getForObject(BASE_URL + "hotels/" +  id, Hotel.class);
```

Consuming a POST

This is the GET endpoint with a path variable that we defined today:

```
@RequestMapping( path = "/hotels/{id}/reservations", method = RequestMethod.POST)
public Reservation addReservation(@RequestBody Reservation reservation, @PathVariable("id") int hotelID) {
    return reservationDAO.create(reservation, hotelID);
}
```

… and this is how a different Java application can consume it:

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
 reservation = restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() + "/reservations",
      entity, Reservation.class);
```