Agenda.

- → MVC pattern.
- → Journey of API request in Spring.
- → REST

MVC ⇒ Model View Controller.

↓

frontend / UI.

⇒ Writing the complete code in a single file is not a good idea as our code won't be

- → Extensible
- → Maintainable
- → Readable.

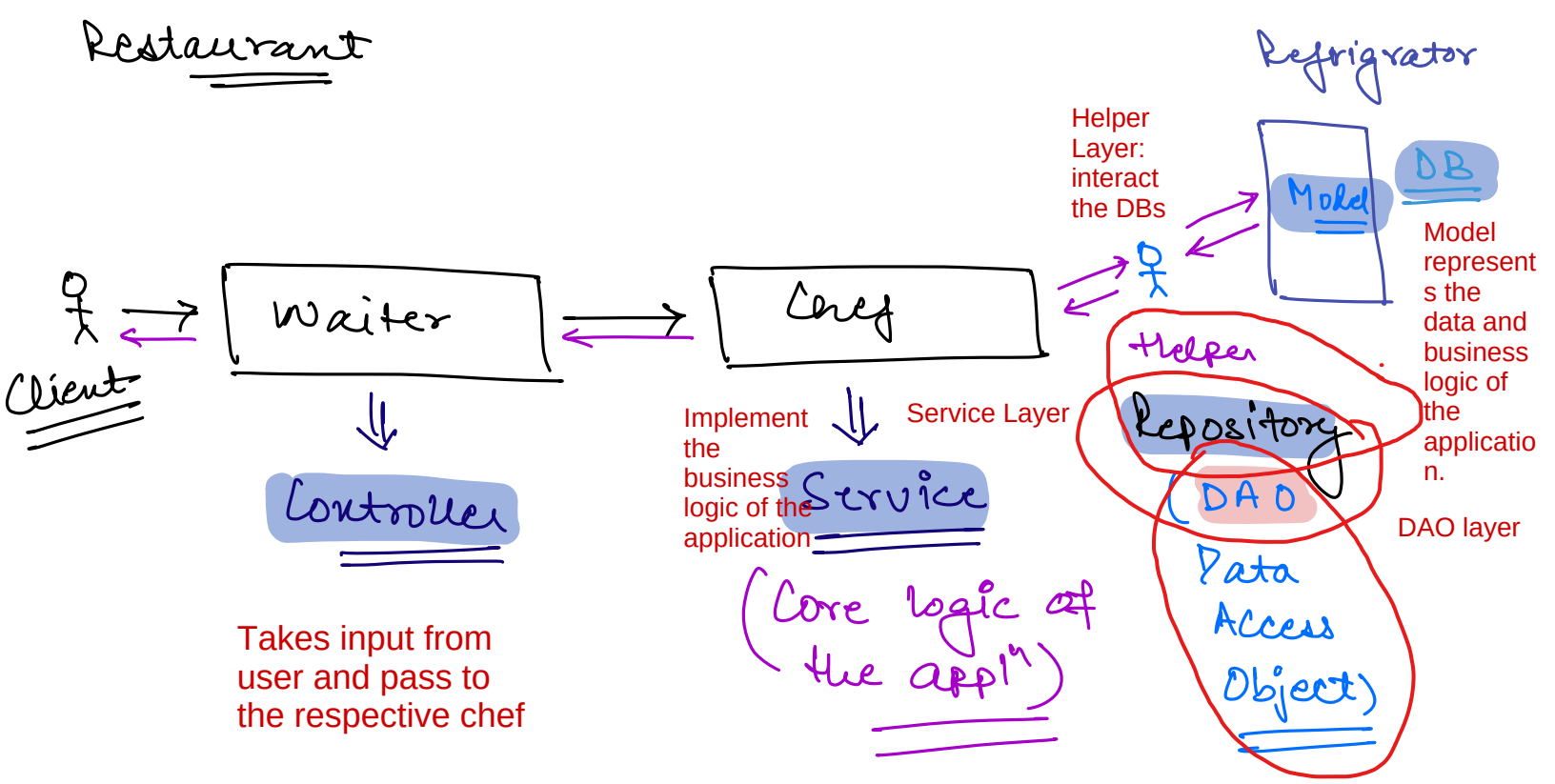
→ Don'ts

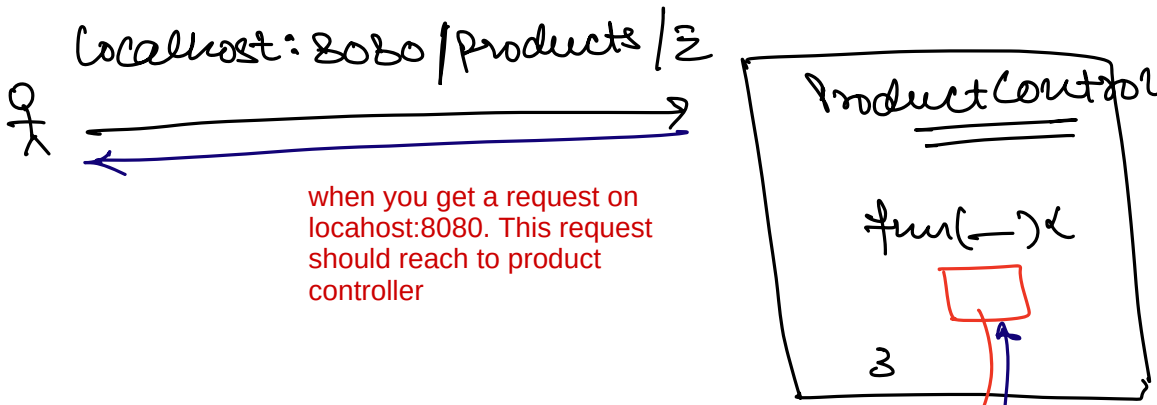
→ DO's

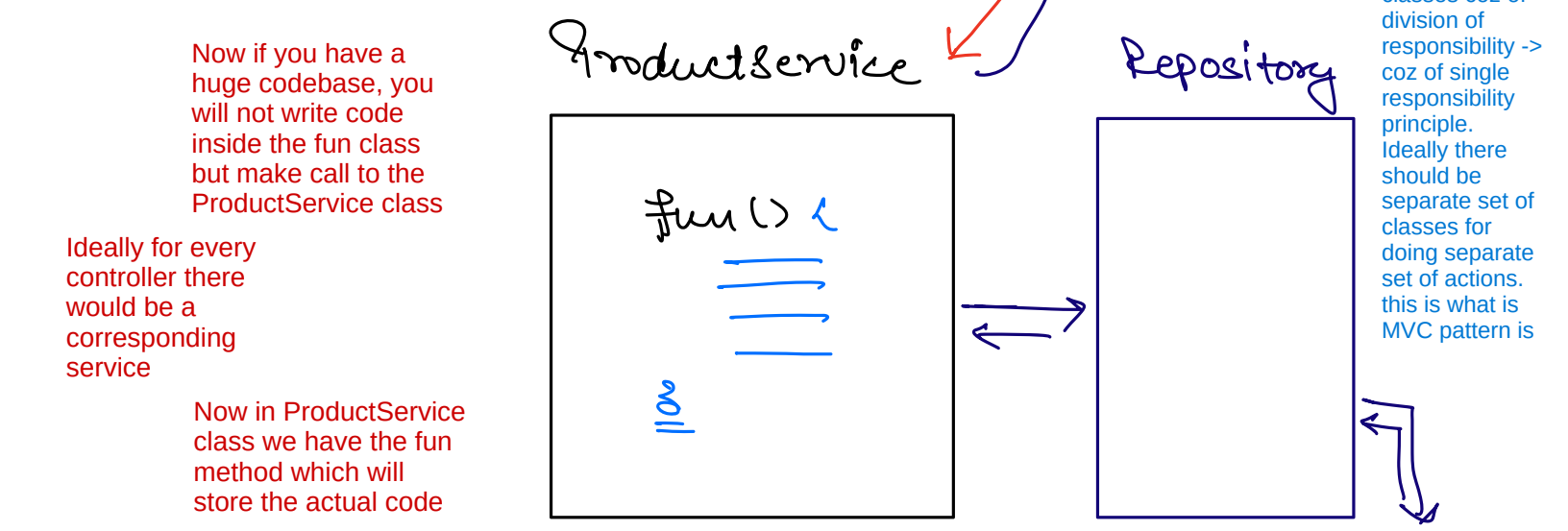
⇒ We should structure our code well in order to make it Extensible, maintainable etc.

⇒ MVC

To make the code structured we use MVC Pattern


Restaurant
Refrigrator
Helper Layer: interact the DBs
Model
DB
Model represents the data and business logic of the application.
Client
Waiter
Chef
Helper
Repository
DAO
DAO layer
Data Access Object
Controller
Takes input from user and pass to the respective chef
Implement the business logic of the application
Service
Service Layer
(Core logic of the appln)


localhost:8080/products/2
when you get a request on locahost:8080. This request should reach to product controller
ProductController
fun(--)
3
When this request reach to product controller, there would be some method say fun method which should get executed for this api


Now if you have a huge codebase, you will not write code inside the fun class but make call to the ProductService class
Ideally for every controller there would be a corresponding service
ProductService
Repository
fun()
Now in ProductService class we have the fun method which will store the actual code
We are creating so many classes coz of division of responsibility -> coz of single responsibility principle. Ideally there should be separate set of classes for doing separate set of actions. this is what is MVC pattern is
Now if we want to interact with database to get some details, we cannot write code to fetch data from db in this fun method else it will be violating single responsibility principle
We will create a layer: Repository to help to get the data back from Db to service layer and then it goes to Controller and then it goes back to Client

MVC: A design pattern on how our API's should be structured.

→ Divide our code into multiple classes with each class serving a specific usecase.

All objects, Spring will create, we will not create objects manually

/models/

Ideally your code should be structured -> you should have the all models in one package, inside which you should have all the models classes.

/controllers/

All the Controllers in one package

/services/

All the Services in one package

/repositories/

All the repositories in one package

#how the journey of a request from server/browser reaches to the Controller

⇒

There are too many controllers in one codebase. Now lets say every controller is a rest controller. So can we say that every controller will have its own endpoint

```
@RestController
@RequestMapping("/products")
ProductController {
    ...
}
```

endpoint of Product Controller

```
@RestController
@RequestMapping("/orders")
OrderController {
    ...
}
```

since this a products endpoint -> the request should go in Product Controller

localhost:8080/products/1 ⇒ ProductController.

localhost:8080/orders/1 ⇒ OrderController.

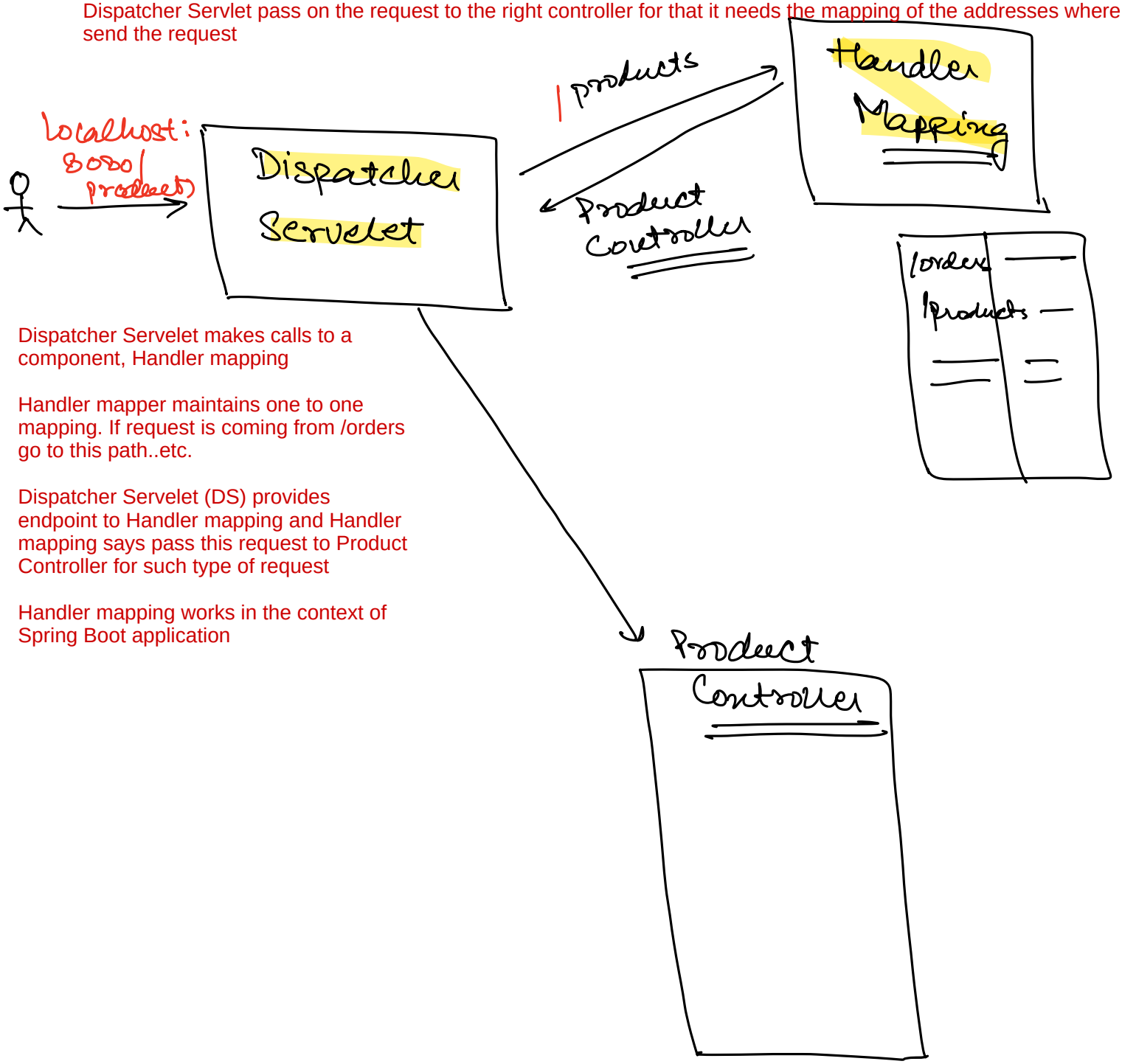
since this a order endpoint -> the request should go in Order Controller

## ⇒ Dispatcher Servlet

DispatcherServlet acts as the Front Controller for Spring-based web applications.

So now what is Front Controller? Any request is going to come into our website the front controller is going to stand in front and is going to accept all the requests and once the front controller accepts that request then this is the job of the front controller that it will make a decision that who is the right controller to handle that request.

Dispatcher Servlet pass on the request to the right controller for that it needs the mapping of the addresses where to send the request




Dispatcher Servelet makes calls to a component, Handler mapping

Handler mapper maintains one to one mapping. If request is coming from /orders go to this path..etc.

Dispatcher Servelet (DS) provides endpoint to Handler mapping and Handler mapping says pass this request to Product Controller for such type of request

Handler mapping works in the context of Spring Boot application

In Spring MVC, Handler Mapping is responsible for mapping incoming HTTP requests to the appropriate controller methods. It determines which controller should handle a given request based on the URL pattern, request method, or other criteria.

The DAO (Data Access Object) Layer is responsible for interacting with the database in a Spring application. It abstracts database operations, making it easier to manage persistence logic.

What is DispatcherServlet in Spring MVC?

DispatcherServlet is the front controller in Spring MVC, responsible for handling all incoming HTTP requests, routing them to appropriate controllers, and returning the response. It acts as the central hub of a Spring Web application.

1. API request is received by Dispatcher Servlet in Spring.
2. Dispatcher Servlet checks with Handler Mapping about which Controller to call.
3. Finally the respective method will be triggered inside the Controller.

SOLID/ Design Patterns -> are to tell how to write good code
MVC -> how APIs should be structured
REST -> how APIs should be named

⇒ REST.

- How the API's should be named.
- Best practices to create API's.

/users/create
/users/delete
/users/get
/users/update

Not as per REST standard.

⇒ Each API must be working on some entity.
Either an API will be creating / Reading / Updating / Deleting some Entity.

1. Every API should be structured around the resource that they are working upon.

product/ order services

/products/create ✗

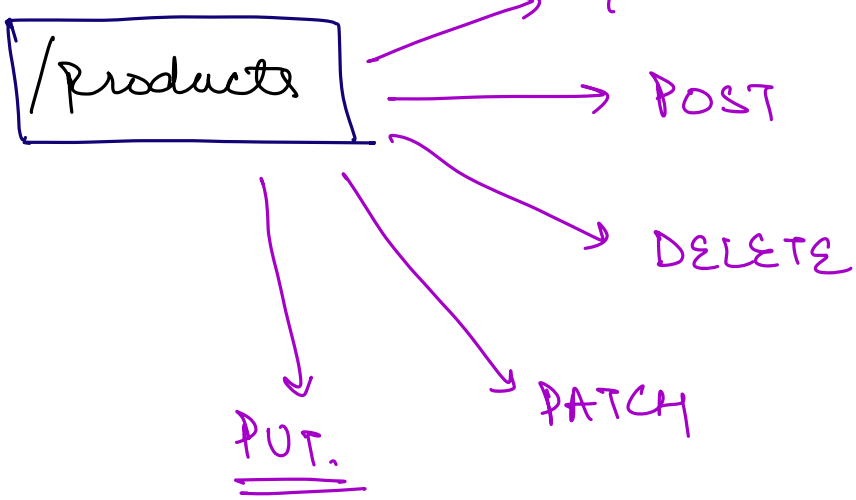
/products/get ✗

⇒ The type of action that API is doing should not be a part of API endpoint.

⇒ The type of action that API is doing should be defined by HTTP method.

whatever action like create or get data the user request should not be included in the API name.

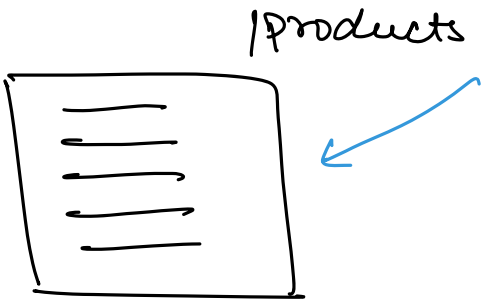
The type of action is defined by HTTP method like get, post, delete, etc




/videos/upload
/videos/delete
} ✗

/videos/

All action should be centered around resource




# HTTP Methods -

Type of http method

→ GET : Fetch data

→ POST : Creating an entity.

↳ Create a product in Products table.

```
{
    "name" : ____
    "title" : ____
    "qty" : ____
    "price" : ____
}
```

with post request, we need to give a lot amt of data as the product details we need to create

→ PATCH : Update an existing entity.

PATCH /products/1

```
{
    "qty" : "100"
}
```

⇒ Partial update

→ PUT : Replace an entity

PUT /products/1

```
{
  "name": —
  "desc": —
  "price": —
  "qty": —
}
```

→ DELETE

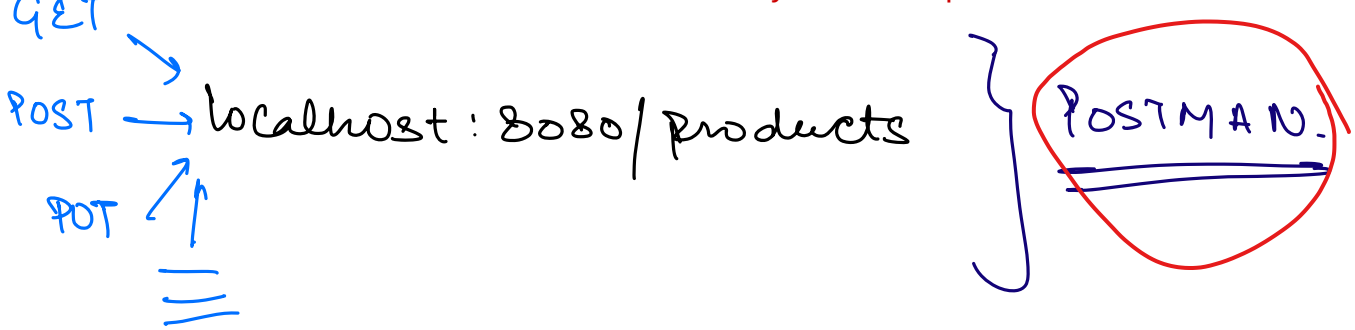
DELETE /products/10

⇒ Delete product with id = 10.

@GetMapping
@PostMapping
@PatchMapping
@PutMapping

Postman is an API platform that helps users build and use APIs. It includes a variety of tools that help with the API lifecycle, including design, testing, documentation, and mocking.

Postman is a client through which you can send API request to a server using which you have to choose which request it will be and a/c you have to pass data for that



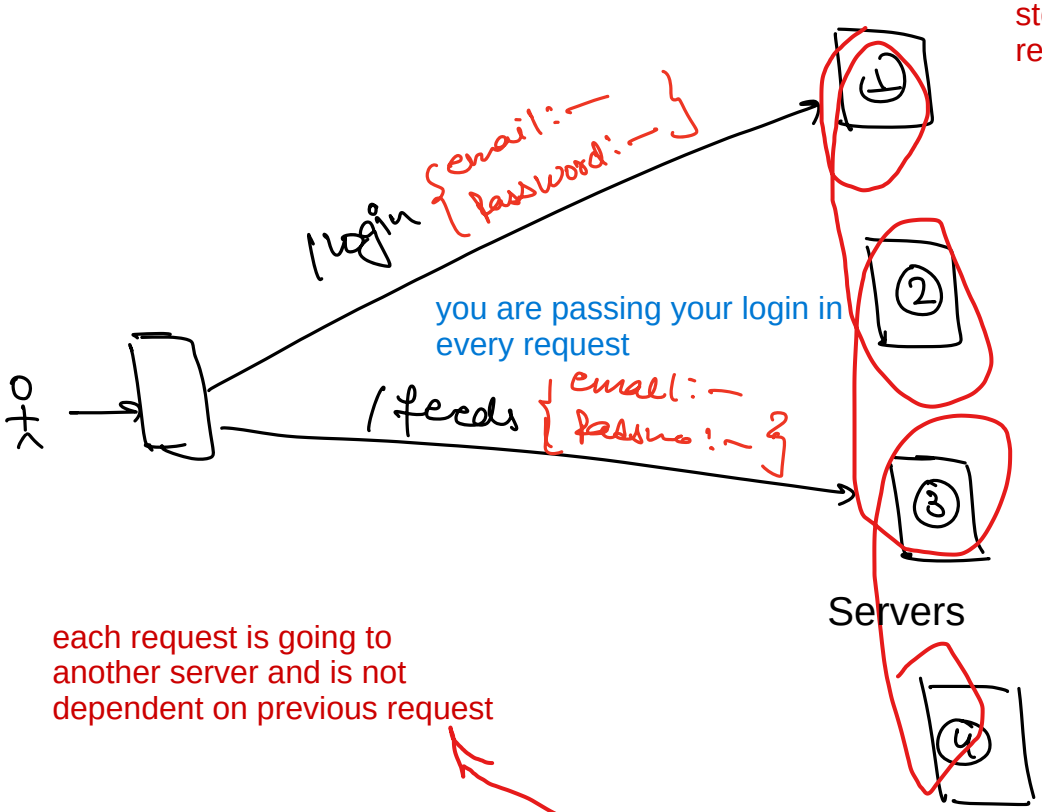
⇒ Rest API's should be Stateless.

In REST, statelessness refers to when the client is responsible for storing and handling the session-related information on its own side.

Each request from the client to server must contain all the information needed to process that request and that information cannot be stored at the server side for any future reference. This restriction is called Statelessness.

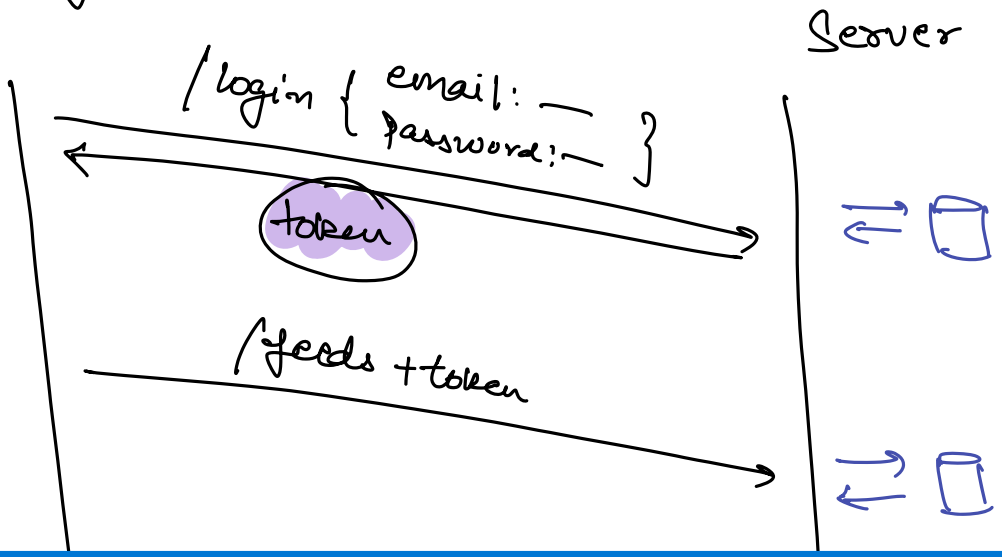
The client is responsible for storing and handling the session-related information on its own side. If any information is required from the previous request, the client will share it in the current request.

"Statelessness means that every HTTP request happens in complete isolation".




each request is going to another server and is not dependent on previous request

⇒ Every request should be completely independent & self sufficient.



⇒ (FTP) : file Transfer Protocol. → stateful request

⇒

users

| id | name | email | phone | address. |
|---|---|---|---|---|
| 5 | Rahul | — | — | — |

mentors

| id | company | sessionsCount | user_id |
|---|---|---|---|
| 10 | Amazon | 100 | 5 |

GET /mentors/10 ⇒ {

"id": 10

"company": —

"SessionsCount": —

"user_id": x

}

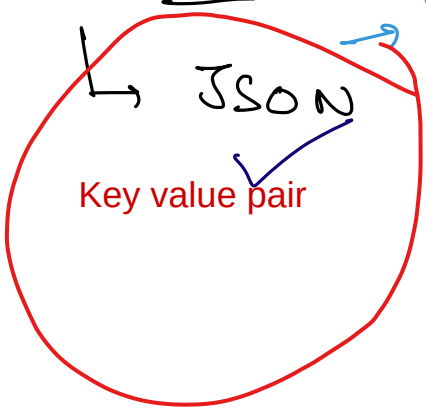
GET /users/x

Chatty API's : Not returning all the relevant data in one go.

↳ Requires client to make multiple API calls to get complete data.

⇒ No Chatty API's.

# No restriction over the return type of our API




→ Most widely used.

```
{
  "id" : 10
  "Company" : ——
  "SessionsCount" : ——
  "user_id" : ——
}
```

→ XML

→ Protobuf.

———— * ————

Apache Tomcat is a free, open-source web server that hosts Java-based web applications. It's developed by the Apache Software Foundation and is written in Java.

Tomcat is a web server (can handle HTTP requests/responses) and web container (implements Java Servlet API, also called servletcontainer) in one. Some may call it an application server, but it is definitely not an fullfledged Java EE application server (it does not implement the whole Java EE API).