Agenda.

→ MUC pattern.
→ Journey of API request in Spring.
→ REST

MVC ⇒ Model View Controller.

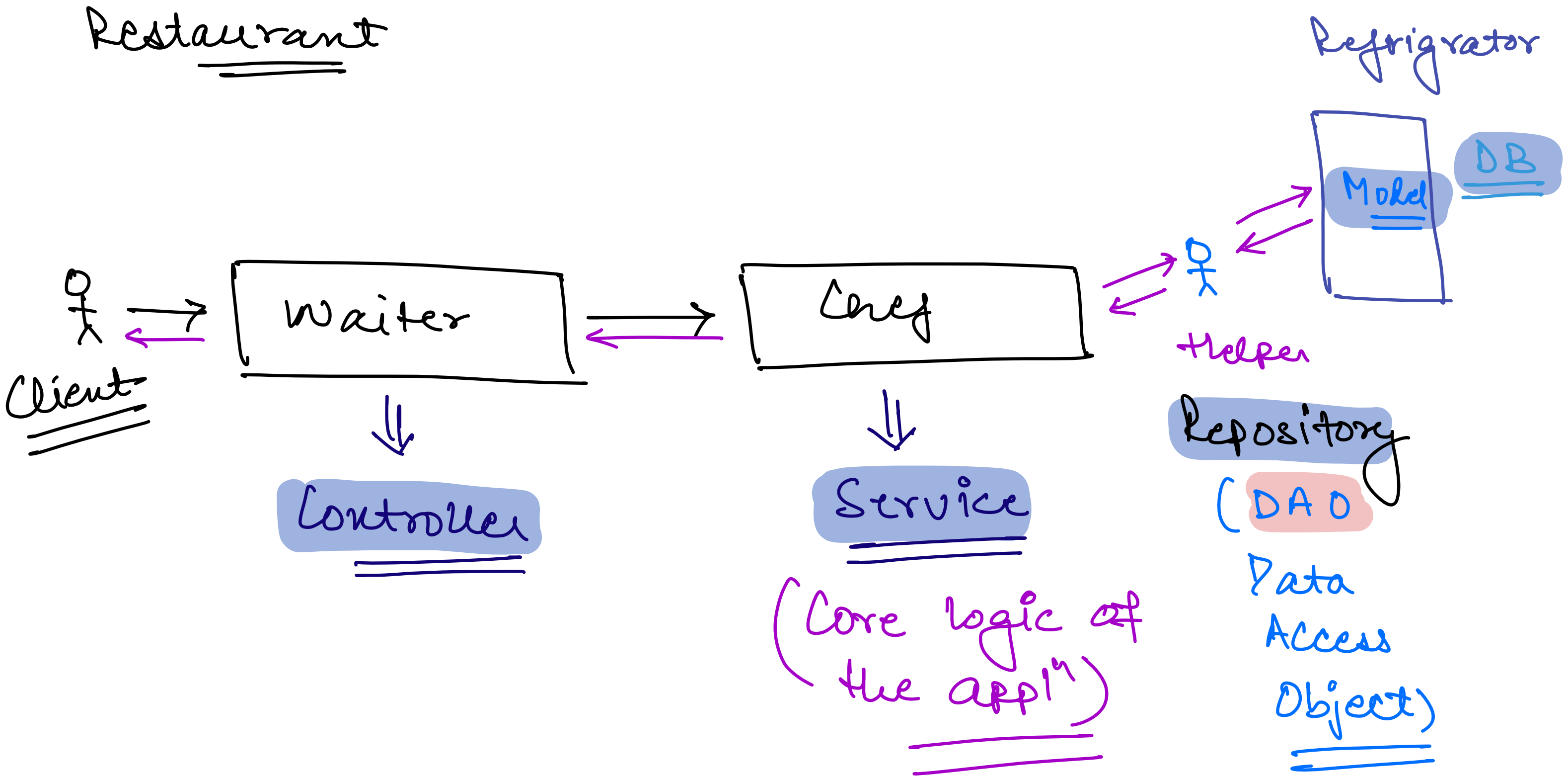
↓

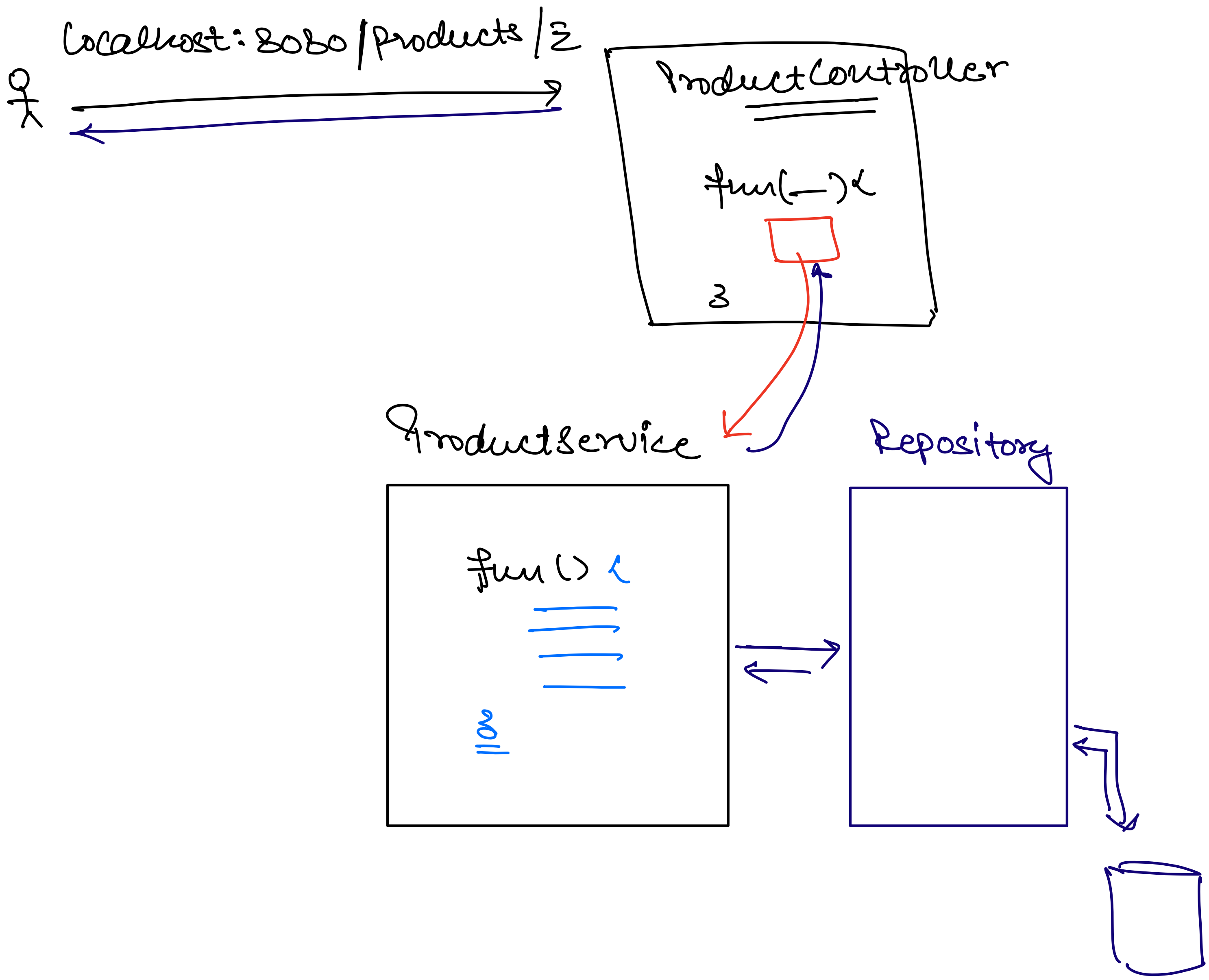
frontend | UI.

⇒ Writing the complete code in a single file is not a good idea as our code won't be
- → Extensible
- → Maintainable
- → Readable.

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

⇒ MUC


Restaurant
Refrigrator
DB
Model
Client
Waiter
Chef
Helper
Controller
Service
(Core logic of the appl^n)
Repository
(DAO
Data
Access
Object)


localhost:8080/products/{
ProductController
fun(—){
}
ProductService
fun() {
}
Repository

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

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

/models/

/Controllers/

/services/

/repositories/

⇒

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

}
```

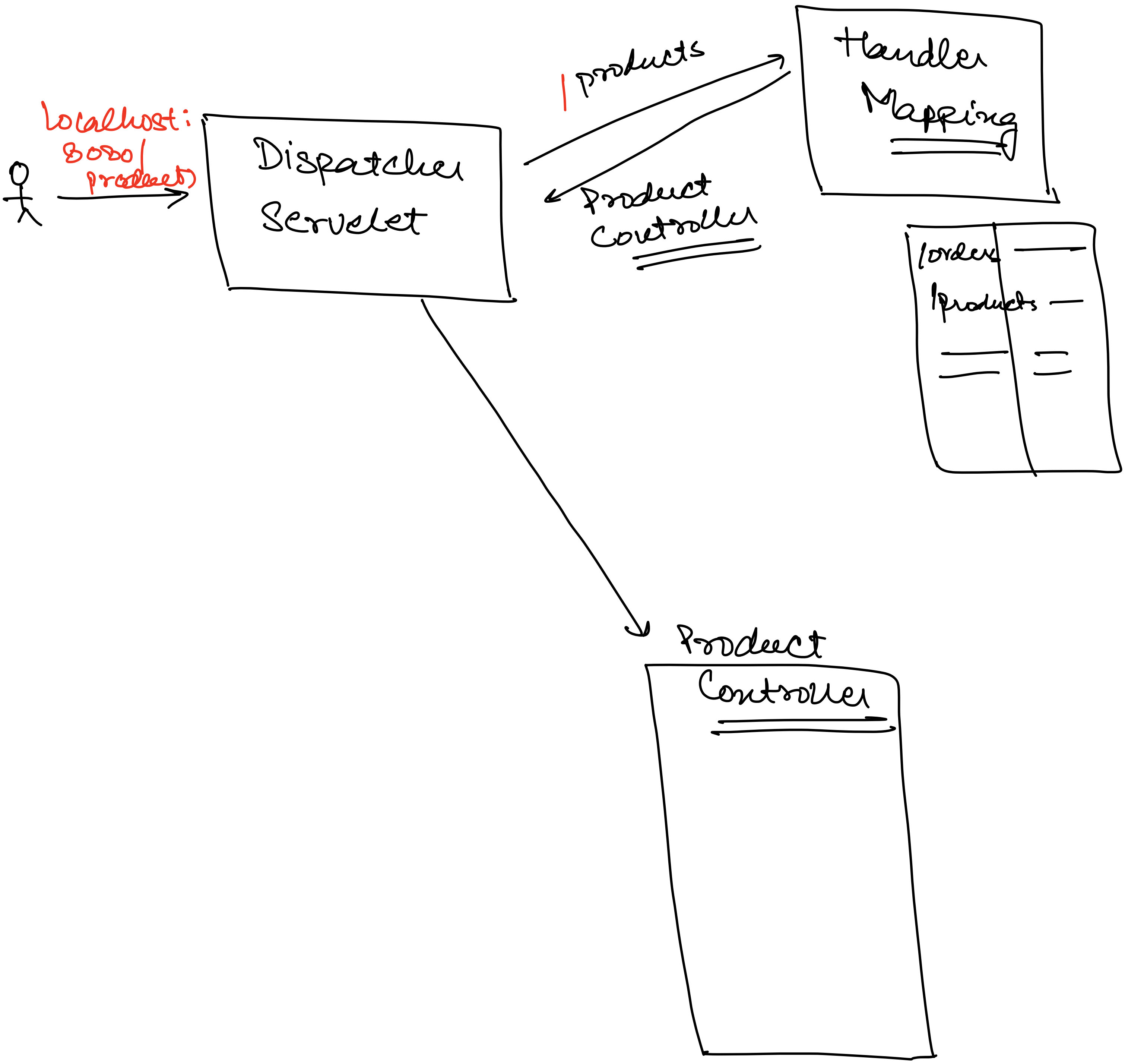
```
@RestController
@RequestMapping("/orders")
OrderController {

}
```

localhost:8080/products/1 ⇒ ProductController.

localhost:8080/orders/1 ⇒ OrderController.

⇒ Dispatcher Servelet


localhost:8080/products
Dispatcher Servelet
/products
Handler Mapping
Product Controller
/order
/products
Product Controller

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.

⇒ 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.

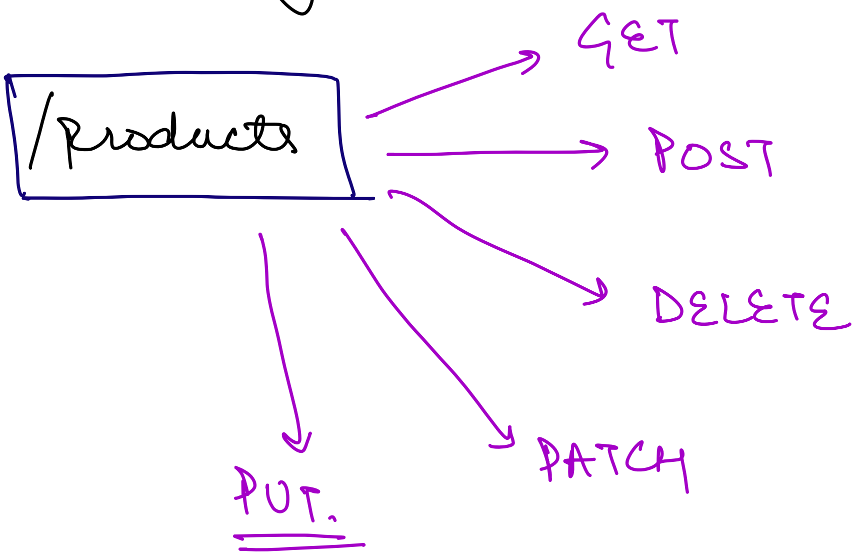
1. Every API should be structured around the resource that they are working upon.

/products/create ✗

/products/get ✗

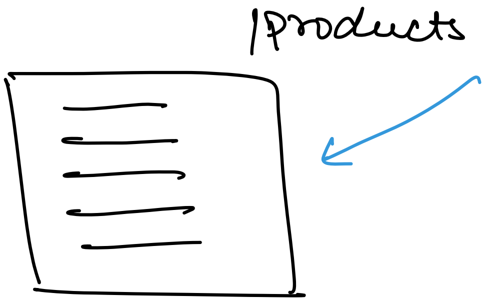
⇒ The type of action that API is doing should not be a part of API endpoint.

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



/videos/upload
/videos/delete } ✗

/videos/



HTTP Methods -

→ GET : fetch data

→ POST : Creating an entity

↳ Create a product in Products table.

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

→ 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

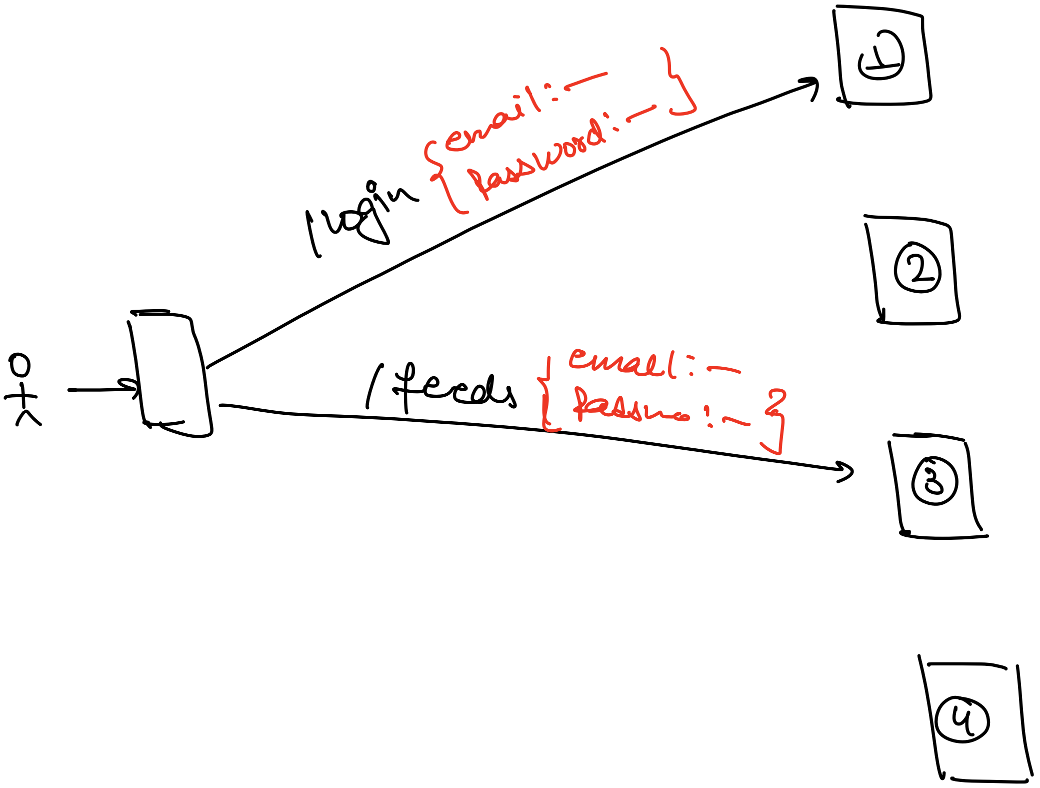
@PutMapping

GET ↘

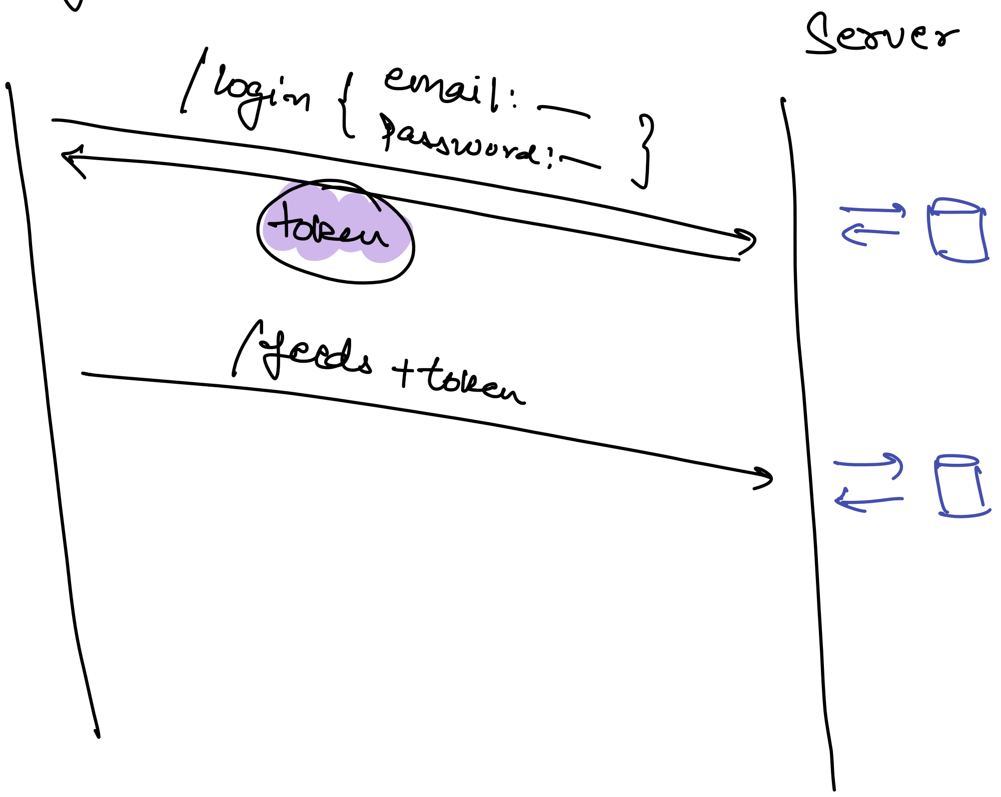
POST → localhost:8080/products } **POSTMAN.**

PUT ↗

⇒ Rest API's should be Stateless.



⇒ Every request should be completely independent & self sufficient.



⇒ FTP : File Transfer Protocol.

⇒ userd

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 /userd/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 own API

└→ JSON ✓ → Most widely used.

```
{
    "id" : 10
    "company" : ___
    "SessionsCount" : ___
    "user_id" : ___
}
```

→ XML

→ Protobuf.

——— * ———