If you have any questions or suggestions regarding the notes, please feel free to reach out to me on

Sanjay Yadav Phone: 8310206130

https://www.linkedin.com/in/yadav-sanjay/

https://www.youtube.com/@GeekySanjay

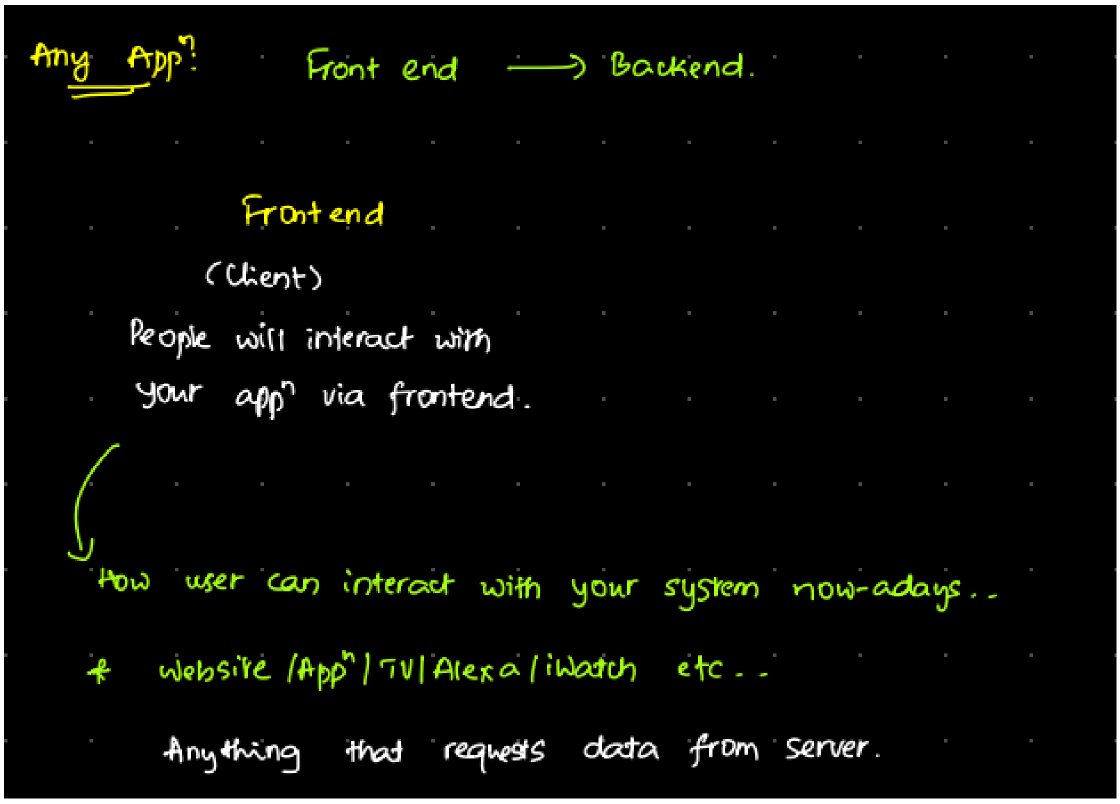
https://github.com/geeky-sanjay/

Backend LLD-4: Intro to Backend Module, Version Control - March 12

- **Frontend v/s Backend**
- **Curriculum**
- **Version Control System (Git)**

Frontend v/s Backend

Frontend: The frontend is the part of a website or application that you see and interact with. It includes everything you experience directly, such as buttons, menus, and the overall design. Frontend developers use technologies like HTML, CSS, and JavaScript to create a user-friendly and visually appealing interface.

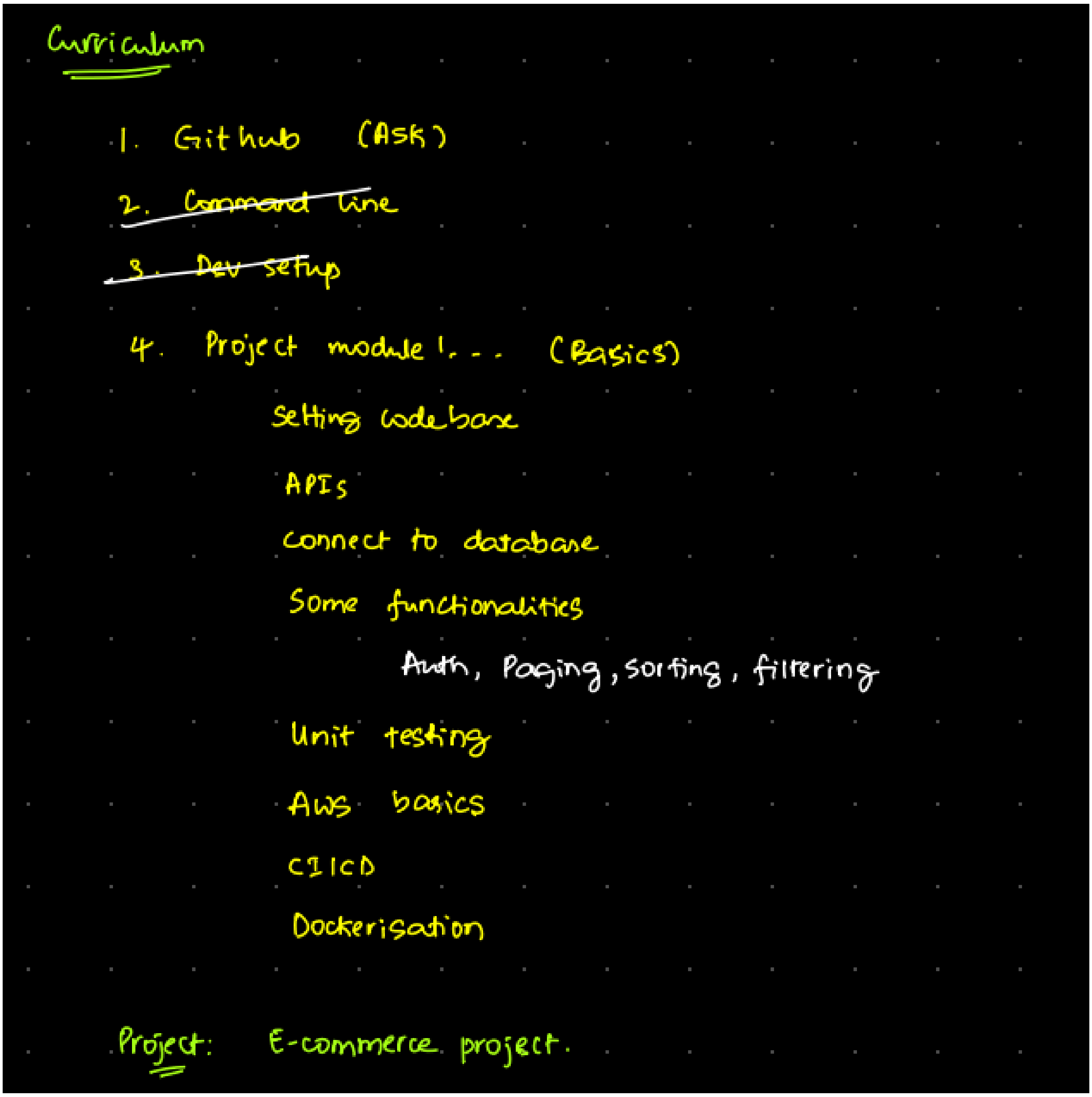

Any Appⁿ?
Front end ⟶ Backend.
Frontend
(Client)
People will interact with
your appⁿ via frontend.
How user can interact with your system now-adays..
* website /Appⁿ / TV / Alexa / iWatch etc..
Anything that requests data from server.

Backend -> The backend is the behind-the-scenes part of a website or application that users don't see. It consists of servers, databases, and application logic that work together to process requests, manage data, and ensure the overall functionality of the system. Backend developers focus on making sure the application runs smoothly, handles data securely, and responds to user actions from the front end.

It's beneficial to place the logic on the backend and send only the necessary data to render the UI. Avoid asking the front end for filtering or complex logic, and don't hand over complicated objects for modifications. It's a good practice to keep the UI as simple as possible, treating the backend as the source of truth for all data.

Various devices like websites, applications, TVs, Alexa, and smartwatches interact with the database in multiple ways.

Curriculum



Curriculum

1. Github (ASK)
2. ~~Command line~~
3. ~~Dev setup~~
4. Project module 1... (Basics)
    - Setting codebase
    - APIs
    - connect to database
    - Some functionalities
        - Auth, Paging, sorting, filtering
    - Unit testing
    - AWS basics
    - CI|CD
    - Dockerisation

Project: E-commerce project.

Version Control System (Git)

A Version Control System (VCS) is essential in software development for several reasons:

History and Tracking Changes:

- VCS allows developers to keep track of changes made to the codebase over time. Every modification, addition, or deletion is recorded, providing a detailed history.

Collaboration:

- In a collaborative environment, multiple developers can work on the same project simultaneously. VCS helps manage and merge changes made by different team members, preventing conflicts.

Branching and Merging:

- VCS enables the creation of branches, allowing developers to work on separate features or bug fixes without affecting the main codebase. Branches can later be merged back, preserving code integrity.

Reverting to Previous States:

- If an issue arises or a mistake is made, VCS allows developers to roll back the codebase to a previous state, undoing changes and restoring stability.

Backup and Redundancy:

- VCS acts as a backup mechanism. All code changes are stored, providing a safety net in case of data loss or system failures.

Traceability and Documentation:

- Each commit in a VCS includes a message describing the changes made. This acts as documentation, providing insights into the development process and making it easier to understand the purpose of each modification.

Continuous Integration and Deployment (CI/CD):

- VCS integrates with CI/CD pipelines, facilitating automated testing, building, and deployment processes. This ensures that only validated and tested code is pushed to production.

Facilitating Code Reviews:

- VCS supports code review processes by providing a clear view of changes made by contributors. Reviewers can examine modifications, suggest improvements, and ensure code quality.

The most widely used VCS is Git, which offers distributed version control and is renowned for its efficiency and flexibility. By adopting a VCS like Git, development teams enhance collaboration, maintain code quality, and manage software projects more effectively.

Version control system (VCS)

When you're working on a project, will you be the only one? -

'X' of engineers work parallely on a codebase.

=) Managing the codebase becomes difficult.

VCS -> $V_1 \to V_2 \to V_3$ - - -

Your codebase will have multiple versions.. instead of we trying to manage those, we're relying on VCS.

Why we need to track versions? -

1) revert because of a change
2) Check the changes that one person has done
3) Compare the code

   ⋮

Formal defn: Keeping track of how codebase looked at a particular time.

Types of VCS

1. Centralised VCS

A single server is maintaining the complete codebase.

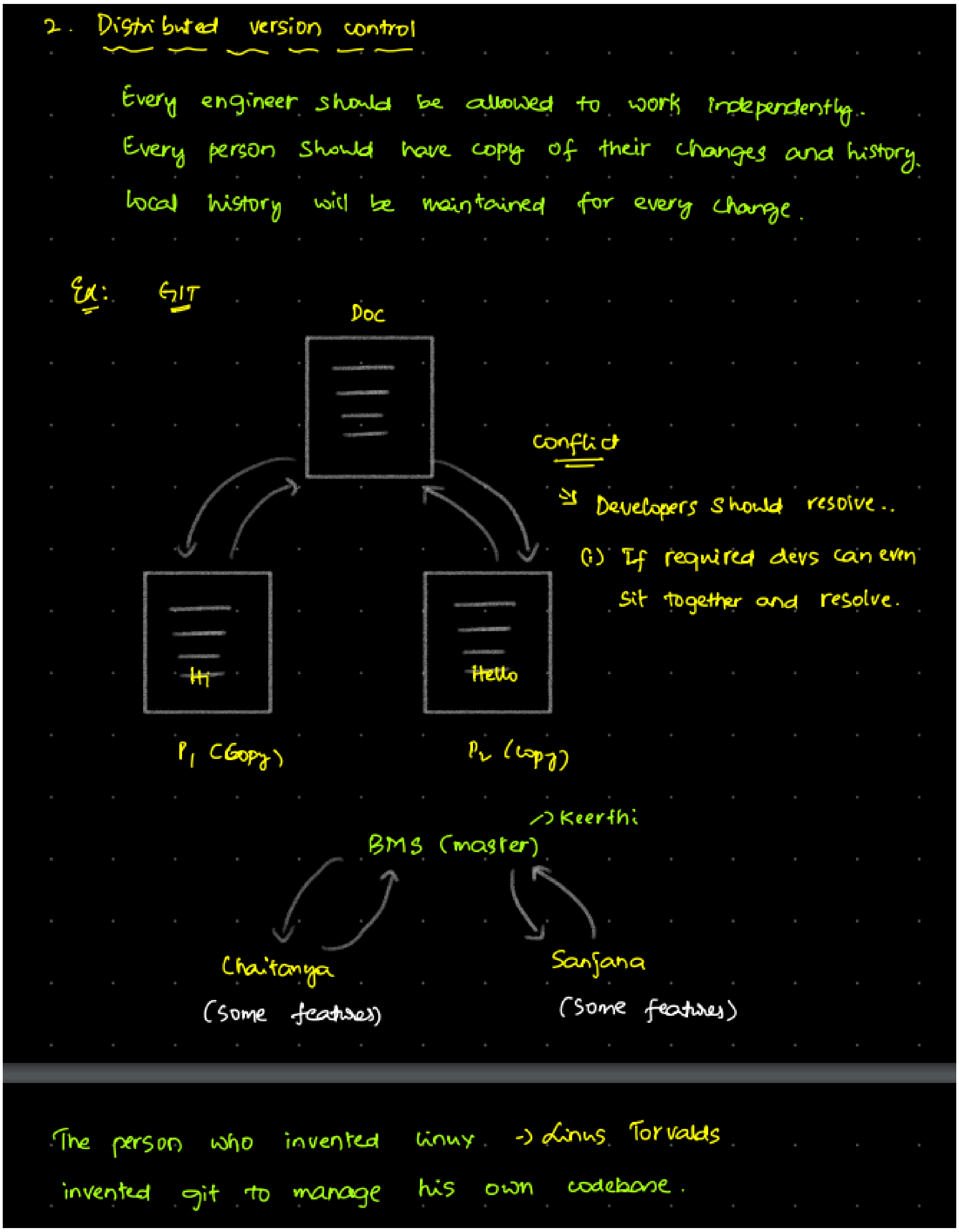
ex: Single server.

3 people are working on it and if for some reason the machine on which google doc is deployed fails. Then all of them will lose the data.

Ex: SVN, Perforce

One issue: Single point of failure.

A centralized version control system (VCS) encounters issues if it fails because there is no backup, creating a single point of failure.



2. Distributed version control

Every engineer should be allowed to work independently.
Every person should have copy of their changes and history.
local history will be maintained for every change.

Ex: GIT

→ Developers should resolve..
(i) If required devs can even sit together and resolve.

The person who invented linux -> Linus Torvalds invented git to manage his own codebase.

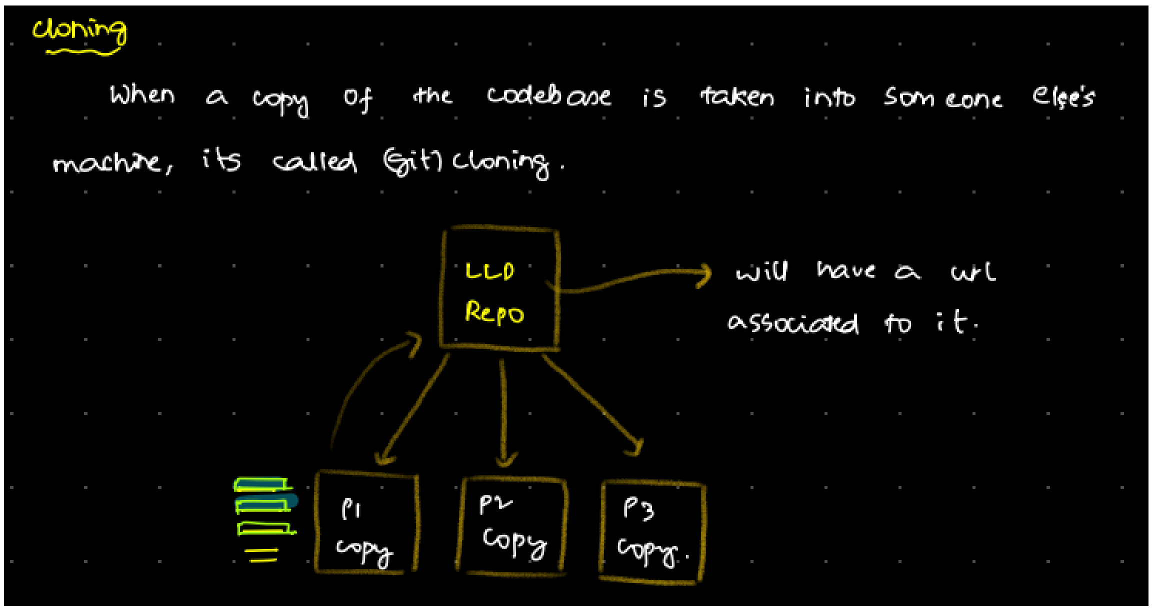
A distributed version control system (DVCS) is a type of version control system where each user has their own local repository, and changes can be synchronized between multiple repositories. Git is a widely used example of a distributed version control system.

learn branching
https://learngitbranching.js.org/

Backend LLD-4: Working with Git Remotes - March 21

- What is Clone
  - Steps involved when working on a project
  - Git vs GitHub
  - How were we adding commits for lld repo
  - creating a new project
  - How to see what has changed
  - Understanding better with few files
  - creating a new repo in GitHub
- Git branch
- Understanding merging
- Understanding branching in learngitbranching.js.org
- Demo of branches in Intellij
- Merge with conflicts

What is Clone -> It is a process of creating a local copy of a remote Git repository. When you clone a repository, you download all of its files, commit history, and branches to your local machine. This allows you to work on the project locally, make changes, and synchronize your work with the remote repository


Cloning

When a copy of the codebase is taken into someone else's machine, its called (Git) Cloning.

Here's how the cloning process typically works in Git:

Find the Repository: First, you need to locate the URL of the Git repository you want to clone. This could be a repository hosted on a platform like GitHub, GitLab, Bitbucket, or a repository hosted on a server accessible via SSH or HTTP.

Clone the Repository: Use the git clone command followed by the URL of the repository to clone it. For example:

git clone <repository_url> -> This command will create a new directory with the same name as the repository and download all of its contents into that directory.

Optional: Specify Directory Name: You can also specify a directory name to clone the repository into, using: **git clone <repository_url> <directory_name>**

Local Copy Created: After running the git clone command, Git will copy the entire history of the repository, including all branches and commits, to your local machine. You'll now have a complete copy of the project that you can work on.

Remote Tracking: When you clone a repository, Git automatically sets up a remote connection to the original repository from which you cloned. By default, this remote is named 'origin'. This allows you to fetch updates from the remote repository and push your changes back to it.

Cloning is a fundamental operation in Git and is typically one of the first steps when starting to work on a project hosted in a Git repository. It provides you with a local copy of the project's entire history, enabling you to work offline and collaborate with others on the same codebase.

Backend LLD-4: Intro to Sprintboot & Calling 3rd Party APIs - 28 march

- **Microservices**
    - **Monolithic Architecture**
    - **Problems with monolithic services**
- **What are frameworks & why they are needed**
    - **Difference between library and framework**
- **Intro to Spring, Dependency injection**
- **Spring Boot**
- **Create Frist API**

===

Springboot Notifications

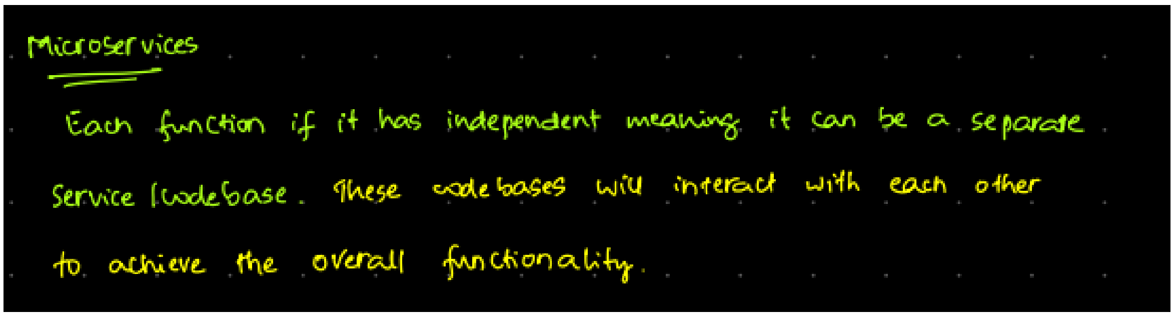
===

@MappedSuperclass
@Inheritance(strategy = InheritanceType.JOINED)
@PrimaryKeyJoinColumn(name="user_id")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Entity
@Service
@Controller
@RestController
@RestTemplate
@Component
@Bean
@ExceptionHandler(ProductNotFoundException.class)
@RestControllerAdvice

===

Microservices

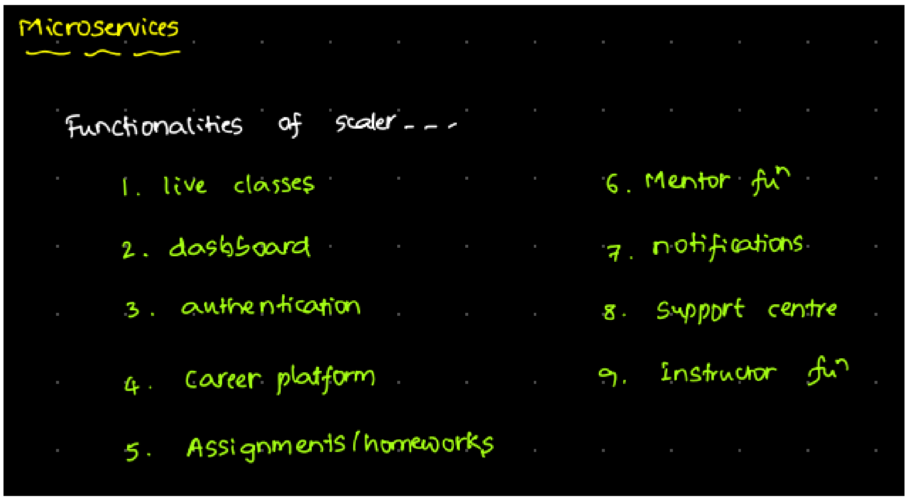
===



Microservices

Each function if it has independent meaning it can be a separate service /codebase. These codebases will interact with each other to achieve the overall functionality.

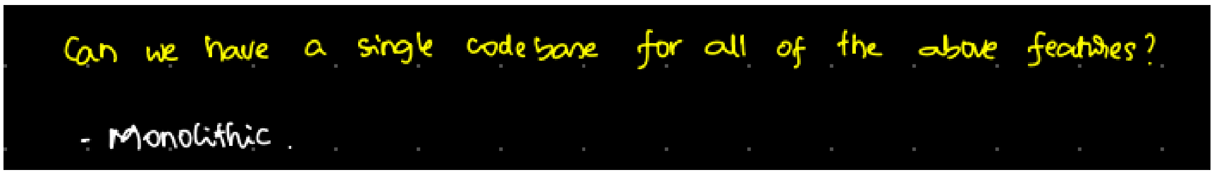
Microservices are a way of building software applications as a collection of small, independent services that work together. Each service focuses on a specific task and can be developed, deployed, and scaled independently.


Microservices

Functionalities of scaler - - -

1. live classes
2. dashboard
3. authentication
4. career platform
5. Assignments/homeworks
6. Mentor fu^n
7. notifications
8. support centre
9. Instructor fu^n

Let's consider the example of a scaler We'll list down its core functionalities and discuss how it operates as microservices compared to if it were monolithic, highlighting the issues that might arise in the monolithic approach.

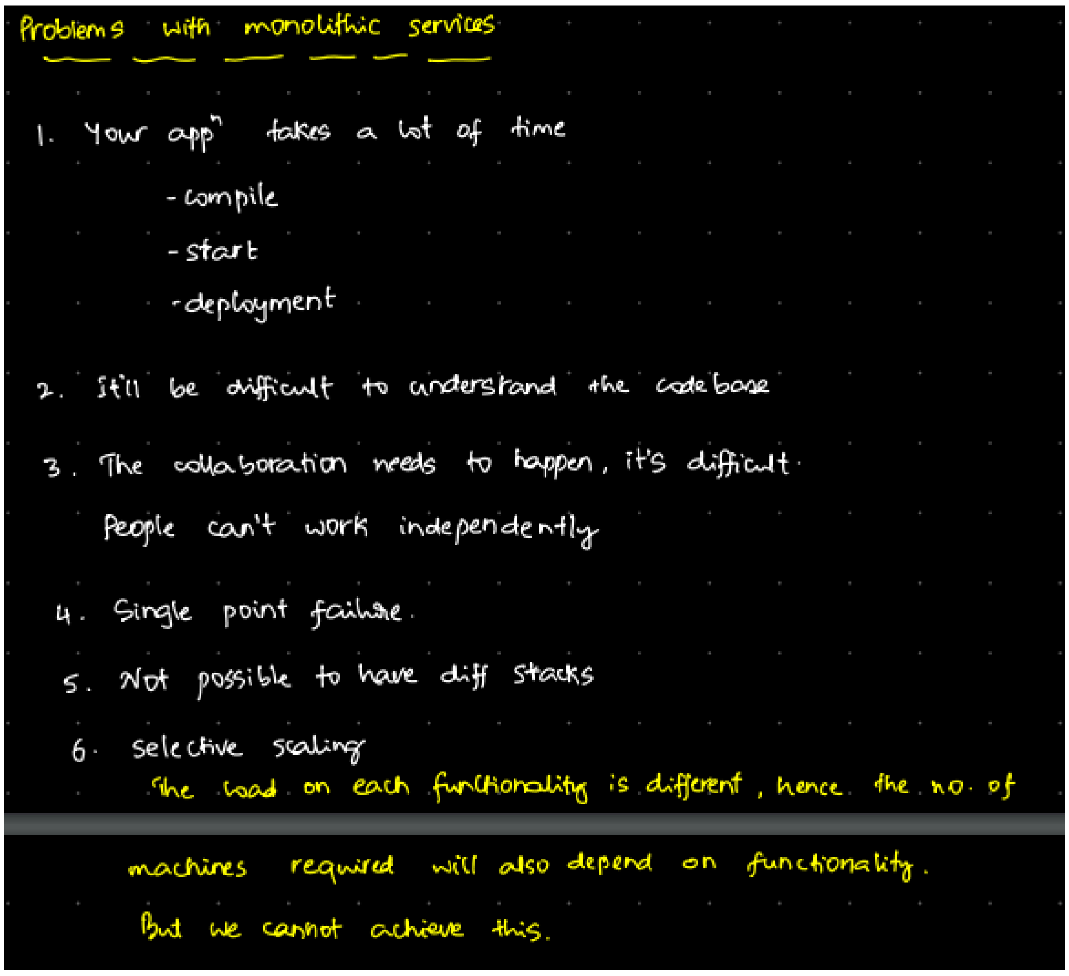

Can we have a single code base for all of the above features?

- Monolithic.

We discussed example of Flipkart like No of people visit of flipcart page is also visit on order confirmation page No. It is vary less so whatevr load or valume we will get on serach page it required many resource and what when we we look to confimation page because low volume its required less resource thatsby microservices benificial based on the requimnet we can up and down the resources and if we will not use microervices then we have to upscale resurce for both which is not reuirend in the case of confi
Let's consider an example similar to Flipkart's website. Suppose the number of people who visit the Flipkart search page is also the same number who visit the order confirmation page. However, this number is relatively low. Therefore, no matter the load or volume on the search page, it requires many resources. On the other hand, since the volume is low on the confirmation page, it requires fewer resources. This is where microservices become beneficial. With microservices, we can adjust the resources based on the requirement. If we didn't use microservices, we'd have to upscale resources for both pages, which isn't necessary for the confirmation page due to its lower volume.rmation page because of low volume.

Monolithic Architecture -> Monolithic architecture refers to a traditional way of building software where all the components of an application are tightly integrated into a single system. In this approach, different functionalities of the application are all bundled together, making it harder to scale or update individual parts independently

Problems with monolithic services



Problems with monolithic services

1. Your app takes a lot of time
   - compile
   - start
   - deployment
2. It'll be difficult to understand the codebase
3. The collaboration needs to happen, it's difficult.
   People can't work independently
4. Single point failure.
5. Not possible to have diff stacks
6. selective scaling
   The load on each functionality is different, hence the no. of machines required will also depend on functionality.
   But we cannot achieve this.

==

What are frameworks & why they are needed

==

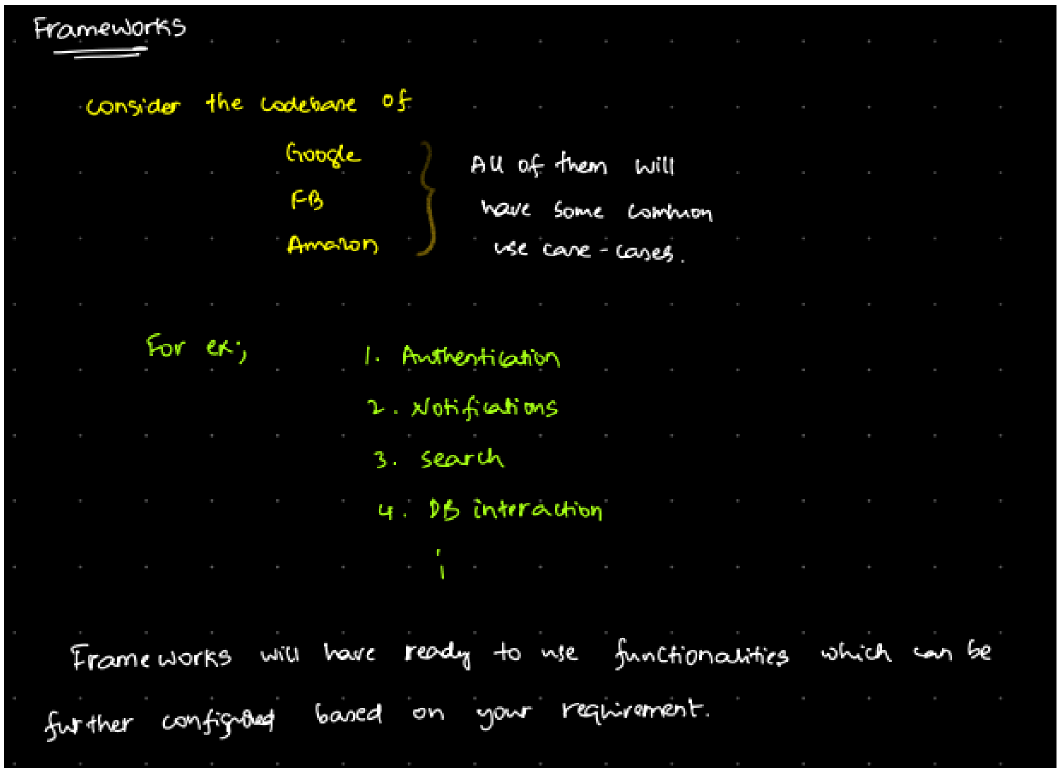
Frameworks are pre-built structures or sets of tools that developers use to build software applications more efficiently. They provide a foundation and a set of guidelines for organizing code and implementing common functionalities, such as user authentication, database interactions, and routing.

Frameworks are needed for several reasons:

1. Efficiency: Frameworks provide ready-made solutions for common tasks, reducing the need to write code from scratch. This saves time and effort during the development process.
2. Consistency: Frameworks enforce consistent coding practices and project structures across development teams, making it easier for multiple developers to work on the same project without confusion.
3. Scalability: Frameworks often include built-in features and best practices for scalability, making it easier to handle large amounts of traffic or data as the application grows.
4. Security: Many frameworks include security features and best practices to help developers protect their applications against common threats, such as cross-site scripting (XSS) or SQL injection attacks.

5. Community Support: Frameworks typically have large communities of developers who contribute plugins, extensions, and documentation, making it easier to find solutions to problems and stay updated on best practices.

Overall, frameworks streamline the development process, improve code quality, and help developers build robust, scalable, and secure applications more quickly.



Frameworks

consider the codebase of

Google
FB
Amazon

All of them will have some common use case - cases.

For ex;

1. Authentication
2. Notifications
3. search
4. DB interaction

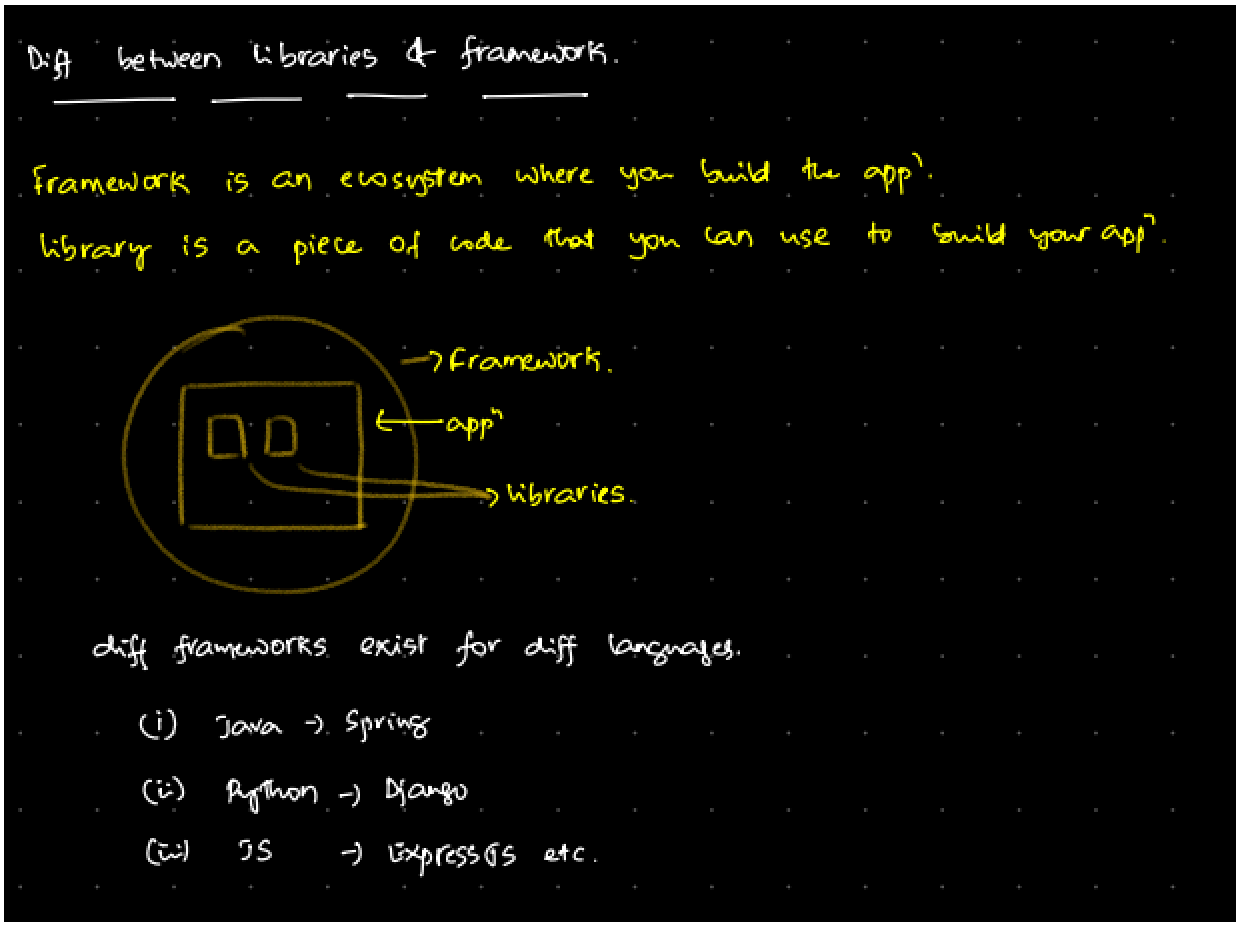
⋮

Frameworks will have ready to use functionalities which can be further configured based on your requirement.

These two sayings are often used in the context of learning from others' experiences and leveraging existing solutions:

1. "**Stand on the shoulders of giants**": This means to benefit from the knowledge and achievements of those who came before us. By studying the work of experts and building upon their ideas, we can achieve greater success than if we were to start from scratch.
2. "**Do not reinvent the wheel**": This implies that instead of creating something entirely new, it's often more efficient to use existing solutions that have already been proven effective. By reusing established methods or tools, we can save time, resources, and effort.

Difference between library and framework -> while both libraries and frameworks provide reusable code components, libraries are more modular and allow developers to pick and choose specific functions, whereas frameworks provide a more structured environment and guide the overall architecture of an application.


Diff between libraries & framework.

framework is an ecosystem where you build the app.

library is a piece of code that you can use to build your app.

framework.
app
libraries.

diff frameworks exist for diff languages.

(i) Java → Spring

(ii) Python → Django

(iii) JS → ExpressJS etc.

As shown in the above screen, the framework serves as the primary outer layer. Within this framework, we develop our application. Within the application, we incorporate multiple libraries based on our specific needs.

There are various frameworks available for different programming languages. For instance, Java has Spring Boot, Python has Django, and JavaScript has Express, React.js, and Angular.
==

Intro to Spring, Dependency injection

==

Spring is a popular framework for building Java applications. It provides comprehensive infrastructure support for developing Java applications. One of the key features of Spring is dependency injection.

Dependency Injection:

Dependency Injection (DI) is a design pattern used in software development. In Spring, it's a technique where objects are provided with their dependencies instead of creating them internally. This means that instead of a class creating its own dependencies, they are provided (injected) from the outside.

- **Dependency:** A dependency is an object that another object depends on to perform its function. For example, if you have a UserService class that needs to interact with a

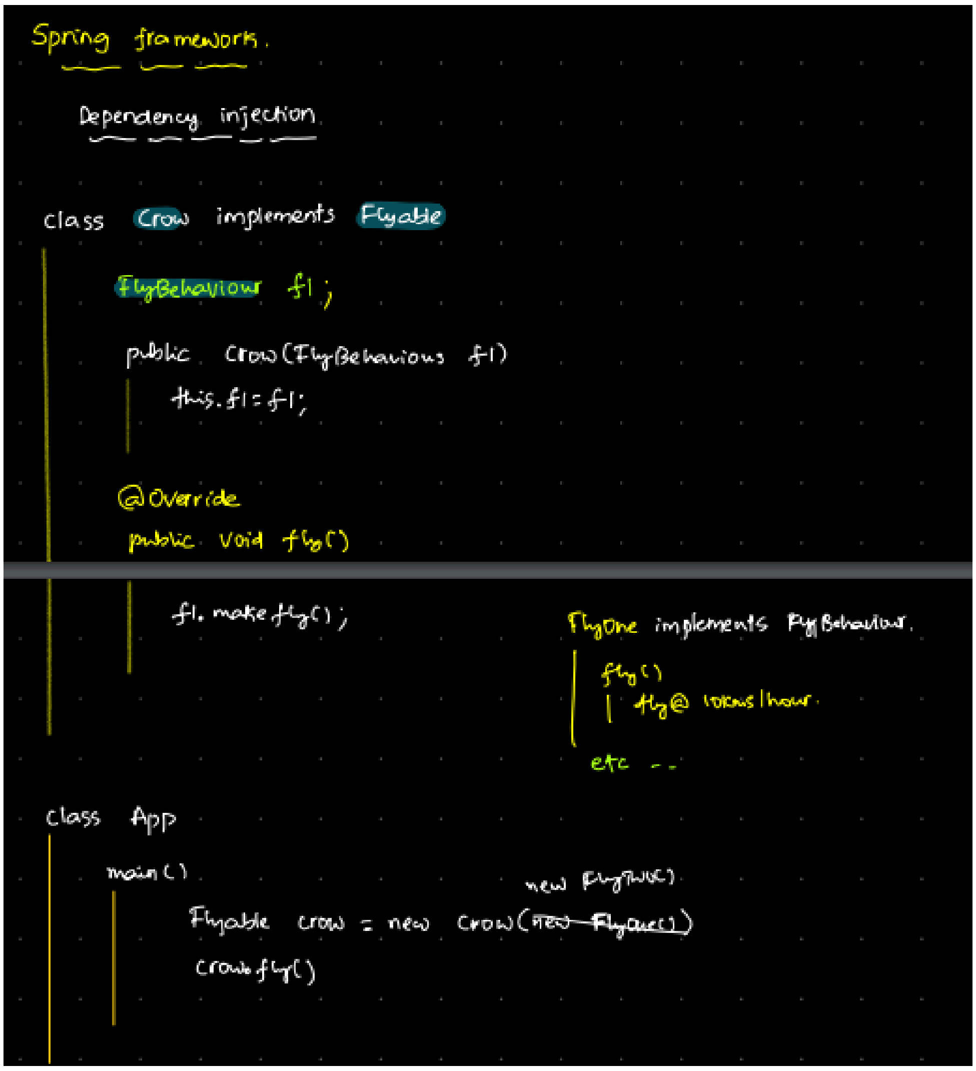
UserRepository to fetch user data from a database, UserRepository is a dependency of UserService.

- **Injection:** Injection refers to the process of supplying a dependency to a class that requires it. This can be done in several ways in Spring, such as constructor injection, setter injection, or field injection.

Advantages of Dependency Injection:

1. **Decoupling:** Dependency injection helps in decoupling components of an application. Classes don't need to directly instantiate their dependencies, making them more reusable and easier to test.
2. **Flexibility:** With dependency injection, it's easier to change implementations or configurations of dependencies without modifying the dependent classes. This promotes flexibility and makes the codebase more maintainable.
3. **Testability:** By providing dependencies externally, it becomes easier to mock or stub dependencies during unit testing, allowing for more isolated and reliable tests.

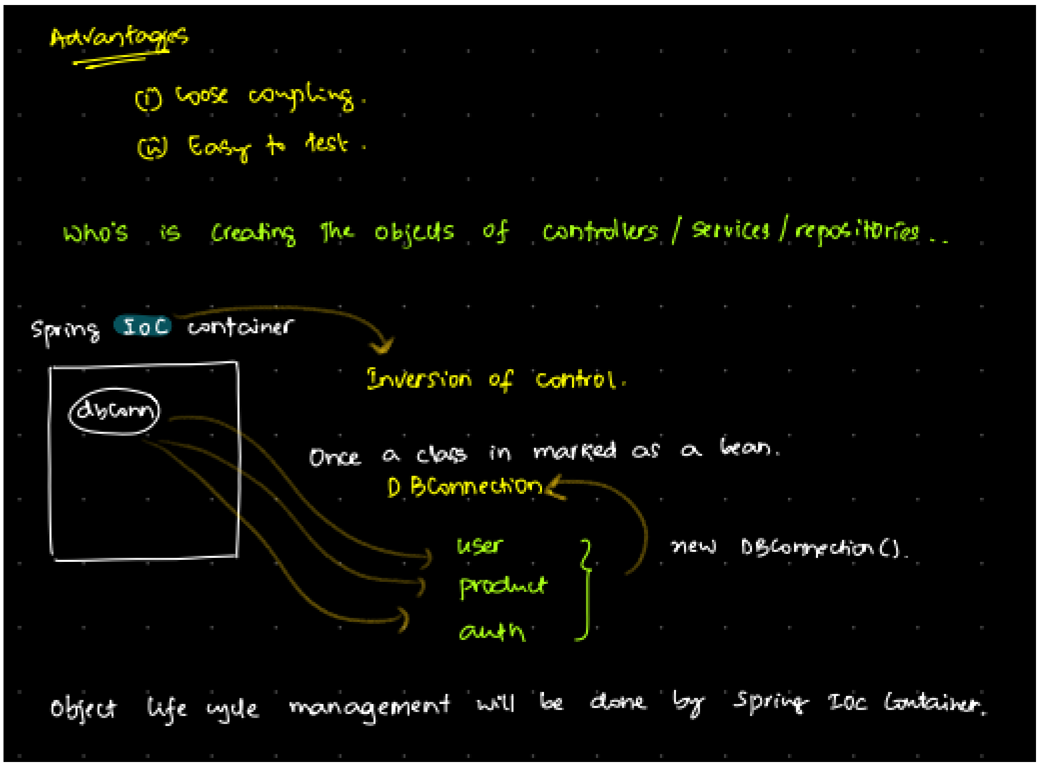
In the example shown below, we have a Crow class that implements the Flyable interface and f1 as a dependency. **f1** is of type **FlyBehaviour** interface, following the abstraction principle. In the constructor of **Crow**, we're accepting a **FlyBehaviour** type object, promoting loose coupling. While it's possible to create a new instance of **f1** within the **Crow** class, this approach would lead to tight coupling, which isn't ideal for programming and scalability. Instead, we opt for dependency injection, where we receive the **FlyBehaviour** object during the instantiation of the **Crow** class. This allows us to pass different dependencies, promoting reusability and enabling the use of the same code multiple times.



Spring framework.

Dependency injection

```
class Crow implements Flyable
    FlyBehaviour f1;
    public Crow(FlyBehaviour f1)
        this.f1=f1;

    @Override
    public void fly()
        f1.makefly();

                                FlyOne implements FlyBehaviour.
                                    fly()
                                        fly@ 10kms/hour.
                                    etc --

class App
    main()
                                        new FlyTwo()
        Flyable crow = new Crow(~~new FlyOne()~~)
        crow.fly()
```



Advantages

(i) Loose coupling.

(ii) Easy to test.

Who's is creating the objects of controllers / services / repositories..

Spring IoC container

Inversion of Control.

Once a class in marked as a bean.

D BConnection

user
product
auth

new DBConnection().

object life cycle management will be done by Spring IoC Container.

In Spring, dependency injection is achieved through the Spring **IoC (Inversion of Control)** container, which manages the creation and configuration of application components. Through the use of annotations or XML configuration, Spring can automatically inject dependencies into your classes, making development more streamlined and modular.

Bean -> a "bean" is simply an object that is managed by the Spring IoC (Inversion of Control) container. In other words, a bean is an instance of a class that Spring manages and configures.

The term "bean" is derived from JavaBeans, which are reusable software components based on the Java programming language. In Spring, beans are typically Java objects that are created, configured, and managed by the Spring container.

Beans in Spring are configured using XML, Java annotations, or Java-based configuration classes. They can represent any application object, such as a service, data access object, controller, or any other type of object.

Key characteristics of Spring beans include:

1. **Lifecycle Management:** Spring manages the lifecycle of beans, including their creation, initialization, usage, and destruction.
2. **Dependency Injection:** Beans can have dependencies on other beans, and Spring handles injecting these dependencies into the bean at runtime.
3. **Scoping:** Beans can have different scopes, such as singleton (one instance per Spring container), prototype (a new instance each time it's requested), request (scoped to an HTTP request), session (scoped to an HTTP session), etc.
4. **Configuration Metadata:** Spring beans are typically configured using metadata such as XML configuration files, Java annotations, or Java-based configuration classes.

Overall, beans are the building blocks of a Spring application, and the Spring IoC container manages these beans, providing features such as dependency injection and lifecycle management.

once a class is marked as a bean in the Spring framework, the Spring IoC (Inversion of Control) container will indeed create an **instance of that class**. This happens during the application's startup process, where the Spring container scans the configuration files or annotations to identify the classes marked as beans. Once identified, the container instantiates these bean classes and manages their lifecycle, including initialization, dependency injection, and destruction if necessary.

Annotation-based Configuration: Beans can be defined using annotations such as **@Component, @Service, @Repository**, or **@Controller**. These annotations are used to mark classes as Spring-managed components, and the Spring container automatically detects and registers these beans during component scanning.

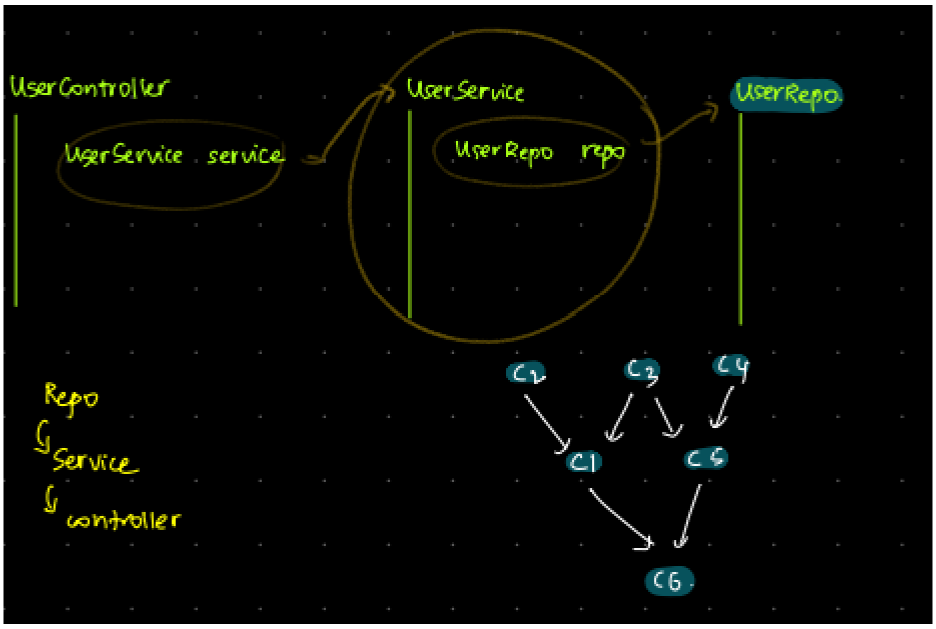
Example annotation-based bean definition:

```java
@Component
public class UserService {
   @Autowired
   private UserRepository userRepository;
   // Bean methods...
}
```

Java Configuration: Beans can be defined using Java configuration classes annotated with @Configuration and bean creation methods annotated with @Bean. In this approach, you write Java code to define and configure beans programmatically.

Example Java configuration:

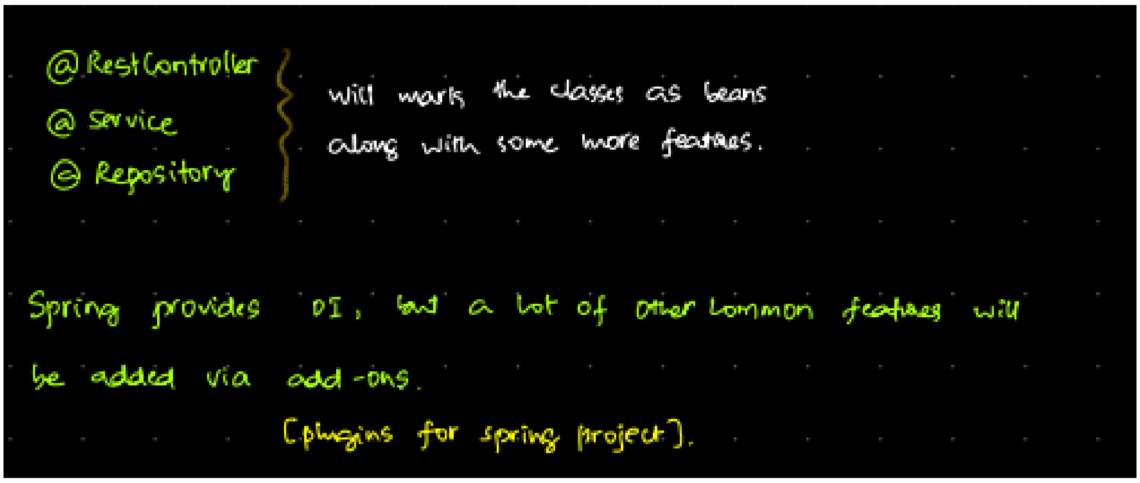
```java
@Configuration
public class AppConfig {
   @Bean
   public UserService userService() {
       return new UserService();
   }
   @Bean
   public UserRepository userRepository() {
       return new UserRepository();
   }
}
```



This simple scenario illustrates above how dependencies function. In this example, the UserController relies on an instance of UserService, and the UserService instance, in turn, relies on an instance of UserRepository. Consequently, the UserRepository instance is created first, followed by the instantiation of UserService. Without UserService, we cannot create UserController because it depends on the creation of other instances. This dependency issue is resolved by the Inversion of Control (IoC) container, which dynamically creates all instances at runtime based on bean annotations.

Similarly, in the case of c2, c3, and c4, c2 and c3 must be created first before c1, and c3 and c4 must exist before c5. Likewise, c6 will only be instantiated when all other instances are created.

Using the annotations listed below, Spring helps us with dependency injection and offers many extra features through add-ons, like plugins for Spring projects.



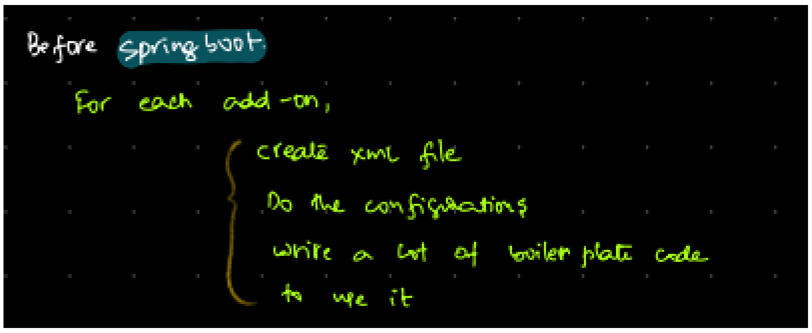
@RestController
@Service
@Repository

will mark the classes as beans along with some more features.

Spring provides DI, but a lot of other common features will be added via add-ons.

[plugins for spring project].

Before Spring -> dependency injection can be configured using XML in Spring.



Before spring boot

For each add-on,

create xml file
Do the configurations
write a lot of boiler plate code
to use it

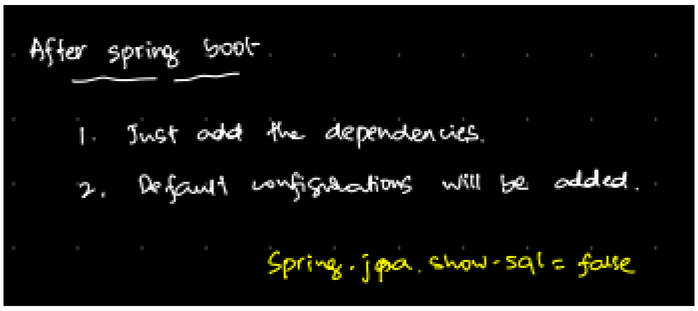
The **<bean>** tags define the beans, with each bean having an id attribute to uniquely identify it and a class attribute specifying the class to be instantiated.
The **<property>** tag within the **<bean>** tag is used to inject dependencies into the beans. In this case, the userService bean has a property named userRepository, which is set to the userRepository bean using the ref attribute.

This XML configuration instructs the Spring container to manage the dependencies between the UserService and UserRepository classes, ensuring that the UserService instance has access to the UserRepository instance when it's created.

XML Configuration: Beans can be defined using XML configuration files. In this approach, you specify the bean definitions in an XML file, including the class name, bean ID, and any dependencies or properties.

```
<bean id="userService" class="com.example.UserService">
   <property name="userRepository" ref="userRepository"/>
</bean>

<bean id="userRepository" class="com.example.UserRepository"/>
```

After spring boot -> With Spring Boot, there's no need to write configuration files from scratch. It provides default configurations that are sufficient to start with, and if necessary, we can override them. For example, in a configuration file, we might set spring.jpa.show.sql to false to disable SQL logging


After spring boot

1. Just add the dependencies.
2. Default configurations will be added.

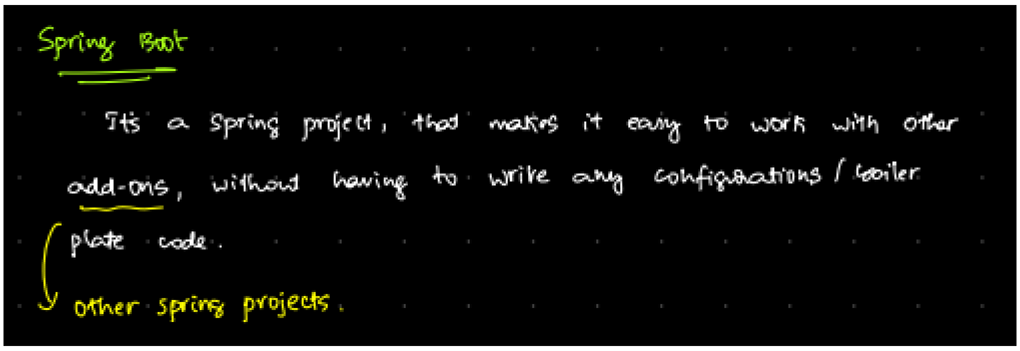
spring.jpa.show-sql = false

==

What is Spring Boot

==

Spring Boot is a popular open-source framework built on top of the Spring framework for developing Java-based enterprise applications. It aims to simplify the process of building and deploying production-ready applications by providing a set of opinionated conventions and defaults.


Spring Boot

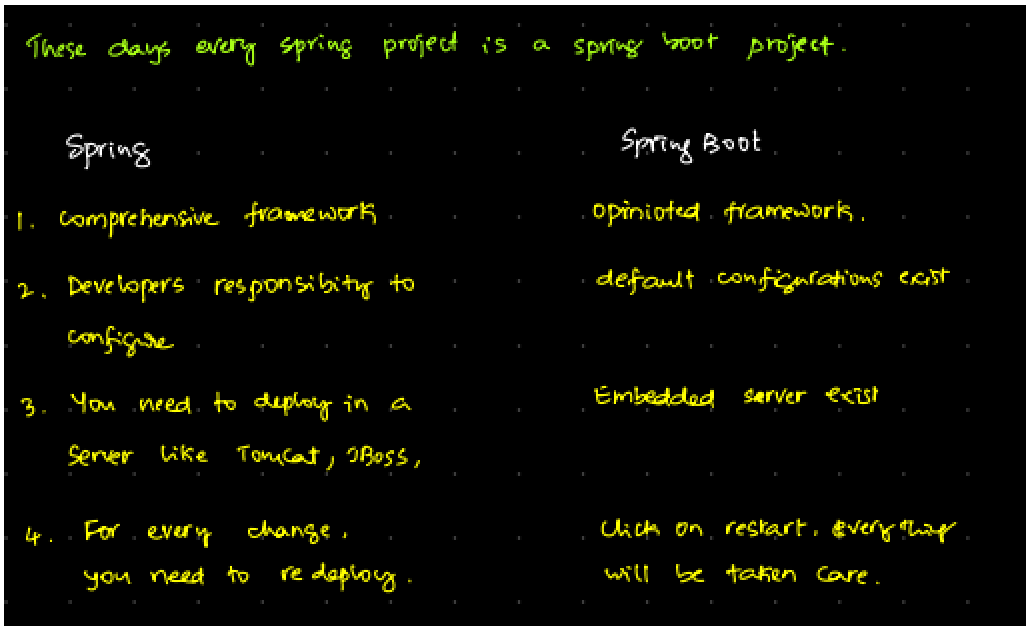
It's a Spring project, that makes it easy to work with other add-ons, without having to write any configurations / boiler plate code.

other spring projects.

Key features of Spring Boot include:

- **Autoconfiguration:** Spring Boot automatically configures many aspects of the application based on the dependencies and environment, reducing the need for manual configuration.
- **Standalone:** Spring Boot applications can be run as standalone JAR files, which embed the application server and all required dependencies. This eliminates the need for deploying applications in traditional servlet containers like Apache Tomcat or Jetty.
- **Spring Ecosystem Integration:** Spring Boot seamlessly integrates with other Spring projects such as Spring MVC, Spring Data, Spring Security, and more, allowing developers to leverage the full power of the Spring ecosystem.
- **Production-ready Features:** Spring Boot includes built-in support for features like health checks, metrics, monitoring, and externalized configuration, making it easier to develop and deploy production-ready applications.
- **Embedded Servers:** Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, allowing applications to be easily packaged and deployed without the need for external servers.

Overall, Spring Boot simplifies and accelerates the development process by providing a streamlined approach to building Java-based applications, reducing boilerplate code, and offering out-of-the-box support for common tasks and features.


These days every spring project is a spring boot project.

Spring	Spring Boot
1. comprehensive framework	opiniated framework.
2. Developers responsibity to configure	default configurations exist
3. You need to deploy in a server like Tomcat, JBoss,	Embedded server exist
4. For every change, you need to re deploy.	Click on restart. everything will be taken care.

An "opinionated" framework like Spring typically comes with built-in conventions and defaults that dictate how applications should be structured and configured. These opinions or preferences are based on best practices and recommendations from the framework's developers. In the context of Spring, being opinionated means that it provides a set of predefined patterns and configurations, simplifying development by reducing the need for extensive customization. However, it also means that developers may need to adhere to these conventions, which can limit flexibility in some cases.

When starting a Spring Boot project, it's suggested to use the Spring Initializr tool. You can find it by searching on Google or by visiting https://start.spring.io. Here's how to use it:

1. Choose your project settings. For example, select Maven as the project type and Java as the language. Pick the latest stable version of Spring Boot, like 3.2.4.
2. Provide project metadata, such as the group (e.g., com.firstProject) and artifact (e.g., springboointro).
3. Choose packaging (e.g., Jar).
4. Select your Java version (e.g., Java 17).
5. Finally, add any additional dependencies you need for your project.
    - First, we'll add the Configuration Processor. This is necessary because we're managing various configurations like application.properties, and if we ever need to override them, we'll need this.
    - Next, we'll include Spring Web. This is for creating a web-based project.
    - After that, we'll add MySQL. This is to connect with MySQL databases.
    - Then, we'll add JPA, which stands for Java Persistence API.
    - Next up is Lombok. It automatically generates getter and setter methods with annotations like **@Getter and @Setter**.
    - We'll also include Dev Tools, which speeds up the application reload process during development.

Once done, **click on generate**. Extract the downloaded file and open the **pom.xml** file in IntelliJ as a project. It will install all the required dependencies. After the installation is complete, you'll find the directory **structure: src -> java -> com.projectname ->**

ProjectnameApplication.java. This is the main file that loads first on startup. If you see "Spring Boot application started" after starting the application, it means everything is set up correctly.
==

Create Frist API

==
 To create a new API, we typically write it within the Controller package. Inside this package, we'll create a HelloController class. For each controller, we'll annotate it with @RestController. Here's an example method:
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

```
@RestController
public class HelloController {
   @GetMapping("/greet")
   public String sayHello() {
       return "Hello. Good Morning";
   }
}
```

In this simple method, we're returning a string. To convert this into an API, we'll use the @GetMapping annotation. This annotation specifies that this method should handle GET requests at the specified endpoint ("/greet"). When we call this endpoint from **http://localhost:8080/greet**, the controller will execute this method, and the return value will be shown in JSON format. The port 8080 is the local server port, which you'll see in the console after starting the application.

Param using query string -> To use parameters in a query string, you typically append them to the URL separated by an ampersand (&) character. For example, to pass parameters name and age with values "John" and "30", respectively, you would construct the URL as follows:
http://localhost:8080/greet?name=John&age=30
We can pass a query like **localhost:8080/greet?name=Sanja**y and retrieve it using **@RequestParam("name")** String name. Here's how we do it:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

  @GetMapping("/greet")
  public String sayHello(@RequestParam("name") String name) {
      return "Hello, Good Morning " + name;
  }
}
```

In this example, we use @RequestParam("name") to get the value of the "name" parameter from the query string. Then, we concatenate it with the greeting message to form the response.

Backend LLD-4: Handling Exceptions & Introduction to Spring Data JPA - 30 march

- **Project overview**
- **Github student developer pack**
- **How to mention this project in resume**
- **How requests are mapped into methods**
- **REST**
- **common http methods**
- **HttpStatus codes**
- **Rest conventions**

==
Project overview
==

Product Requirements Document (PRD) for Ecommerce Website

Functional Requirements

1. User Management

1.1. Registration: Allow new users to create an account using their email or social media profiles.
1.2. Login: Users should be able to securely log in using their credentials.
1.3. Profile Management: Users should have the ability to view and modify their profile details.
1.4. Password Reset: Users must have the option to reset their password through a secure link.

2. Product Catalog

2.1. Browsing: Users should be able to browse products by different categories.
2.2. Product Details: Detailed product pages with product images, descriptions, specifications, and other relevant information.
2.3. Search: Users must be able to search for products using keywords.

3. Cart & Checkout

3.1. Add to Cart: Users should be able to add products to their cart.
3.2. Cart Review: View selected items in the cart with price, quantity, and total details.
3.3. Checkout: Seamless process to finalize the purchase, including specifying delivery address and payment method.

4. Order Management

4.1. Order Confirmation: After making a purchase, users should receive a confirmation with order details.
4.2. Order History: Users should be able to view their past orders.
4.3. Order Tracking: Provide users with a way to track their order's delivery status.

5. Payment

5.1. Multiple Payment Options: Support for credit/debit cards, online banking, and other popular payment methods.
5.2. Secure Transactions: Ensure user trust by facilitating secure payment transactions.
5.3. Payment Receipt: Provide users with a receipt after a successful payment.

6. Authentication

6.1. Secure Authentication: Ensure that user data remains private and secure during login and throughout their session.
6.2. Session Management: Users should remain logged in for a specified duration or until they decide to log out.

===

Github student developer pack

===

Please fill access support form. You will get access in 3 days after filing this form

https://docs.google.com/forms/d/166iMAhQdvLDpdqpMOi2ulv_5hi0J5FPtAD9xjMA1g9A/viewform?edit_requested=true

==

How to mention this project in resume

==

- Architected a Microservices based ECommerce Platform with support of core functionalities of any ECommerce Website like Product Catalog, Payment Gateway Integration, Order Service, Notification Service etc.
- Implemented Event Driven Email Service to allow sending emails at large scale across different services within the Platform.
- Implemented powerful sorting, filtering using ElasticSearch to allow for efficient discovery of products.
- Optimized the response time of APIs from ~500 ms to ~50 ms by making effective usage of Caching for static data using Redis Cache.
- **Tools and Frameworks Used:** Spring Boot, Spring Cloud, MySQL, Redis, Razorpay Payment Gateway, JUnit, Kafka.

==

How requests are mapped into methods

==

We might have lots of different APIs in our application. Finding them all by checking each class or controller can be really hard.

To make it easier, we can change our logging settings in a file called application.properties. By adding these lines:

```
logging.level.org.springframework.web = TRACE
```

We're telling our program to make detailed logs for the parts of the code that handle web stuff in Spring Boot. These logs will show things like which HTTP method is used, the name of the API, and the method being called.

This detailed logging helps a lot when we're trying to figure out what's happening with our web requests and responses in Spring MVC or Spring WebFlux.

Here's how it works:

When a request comes in, a part of the program called DispatcherServlet tries to find out where it should go. Once it knows that, when another request comes in for a specific API, it figures out which method in the code is supposed to handle it, like "sayHello" for example. Then, it runs that method.

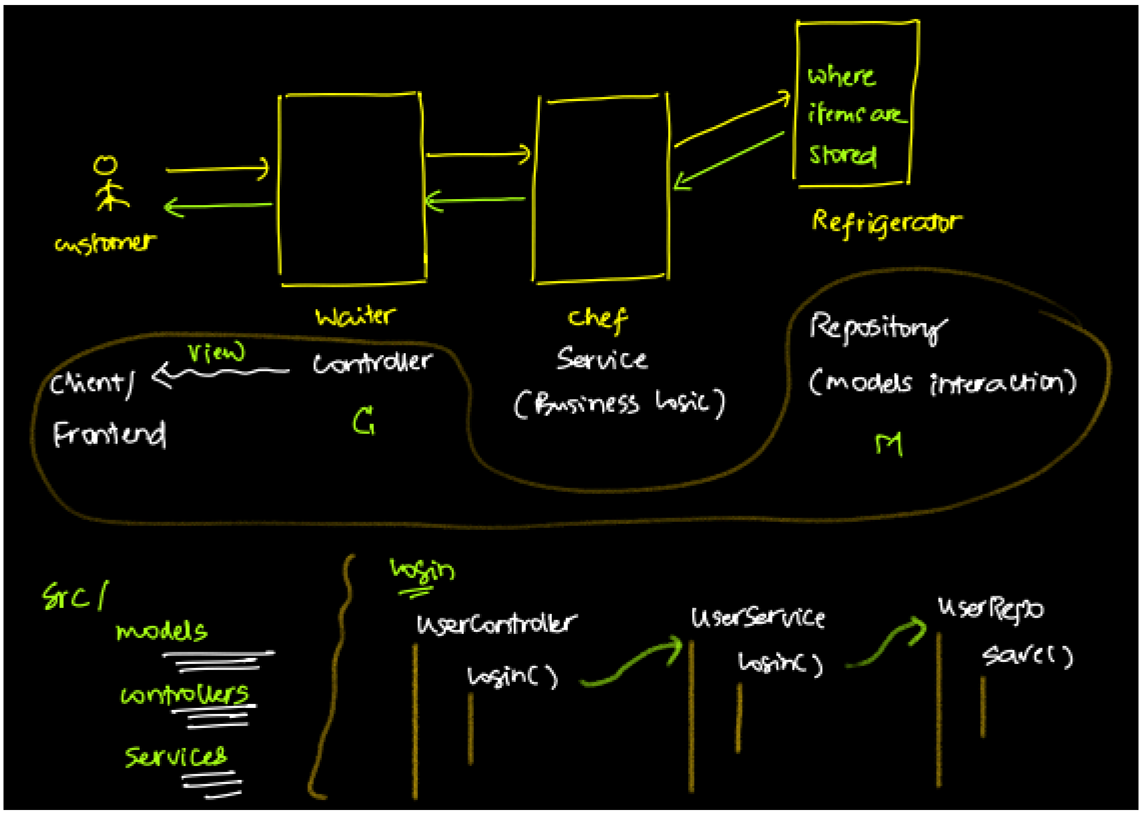
This logging setup makes it much easier to track what's happening with our APIs and methods, which is super useful for debugging and monitoring our application.

===

MVC Pattern Revision

==

Dividing different responsibilities among different classes is what the MVC pattern is all about.



===

API - Application Programming Interface

===

An API, or Application Programming Interface, can be thought of as a contract. It's like a promise: when you give something to it, it returns something back.

In the context of an application, the API acts as a way for different parts of the program to communicate. It sets up a standard way for classes within the application to interact with each other.

Similarly, in a microservices architecture, APIs play a crucial role in enabling communication between different services. They define a set of rules or contracts that services must follow when interacting with each other.

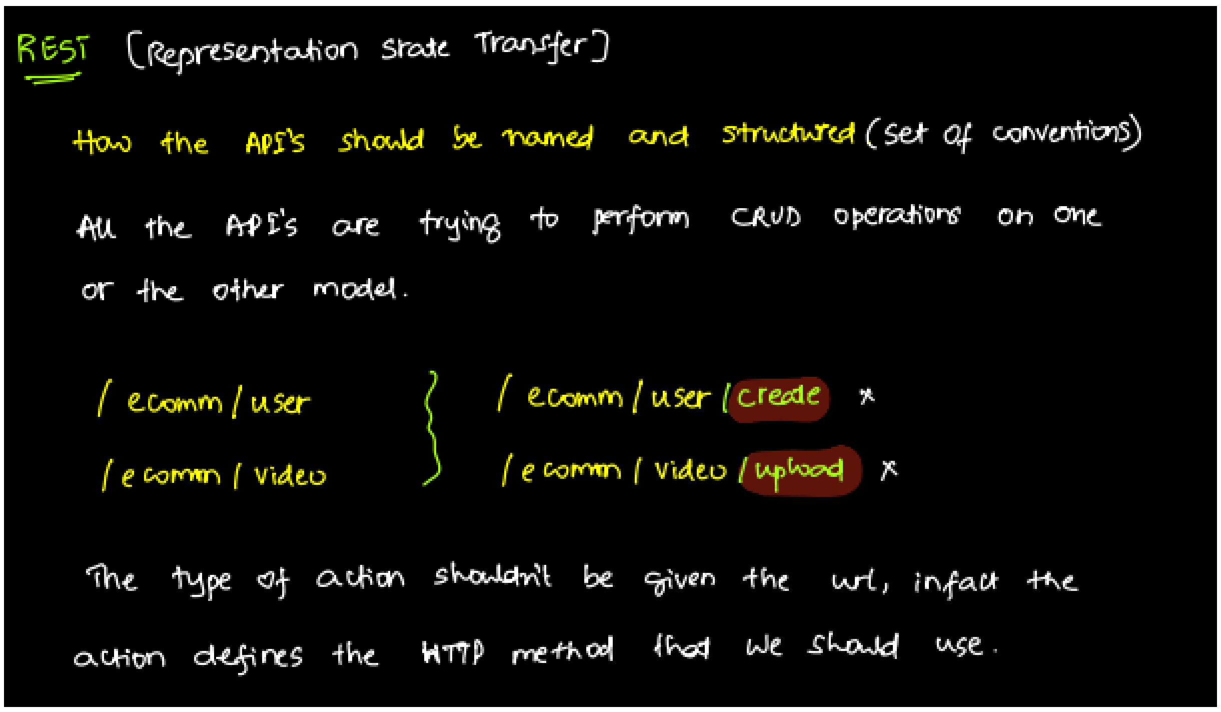
===

REST - Representation state Transfer

===

REST, which stands for Representational State Transfer, is a set of standards that guide how APIs should be designed and structured. It's like a rulebook we follow when creating APIs.

Instead of creating specific API endpoints like "/ecom/video/create" or "/ecom/video/upload" for different functionalities, we keep it more generic, like "/ecom/video/". We then use different HTTP methods to differentiate actions. For example, we use "GET" to retrieve data and "POST" to create data, both using the same URL. This helps keep our APIs consistent and easier to understand.



REST (Representation State Transfer)

How the API's should be named and structured (set of conventions)

All the API's are trying to perform CRUD operations on one or the other model.

/ecomm/user
/ecomm/video

/ecomm/user/create ✗
/ecomm/video/upload ✗

The type of action shouldn't be given the url, infact the action defines the HTTP method that we should use.

==

Common HTTP methods

==

Here are some common HTTP methods explained:

1. **GET**: Used for retrieving data based on a query. GET requests should not modify or create any data. All required data is passed through the query string in the URL, and no data is accepted in the request body.
2. **POST**: Used for creating a new record. POST requests accept data in the request body, which is used to create the new record.
3. **PATCH**: Used to update existing data on the server. It allows for partial updates, meaning you can modify one or more values within the data without replacing the entire object.
4. **PUT**: Used for complete updates. PUT requests replace the entire object on the server with the data provided in the request body. If we will not pass any values empty it will considered as empty.
5. **Delete:** It is used to delete record based on the id.
   Note: if we will not provid id then it will delete al record.

==

HttpStatus codes

==

HTTP status codes provide information about the result of a client's request. Here's a summary of the most important ones:

- **2xx (Success):** Indicates that the request was received, understood, and accepted successfully. Examples include:
  - 200 OK: The request was successful.
  - 201 Created: The request resulted in the creation of a new resource.
  - 204 No Content: The server successfully processed the request but is not returning any content.
- **4xx (Client Error):** Indicates that there was a problem with the client's request. Examples include:
  - 400 Bad Request: The request could not be understood by the server due to malformed syntax.
  - 401 Unauthorized: The request requires user authentication.
  - 403 Forbidden: The server understood the request, but it refuses to authorize it.
  - 404 Not Found: The requested resource could not be found on the server.
- **5xx (Server Error):** Indicates that the server encountered an error while processing the request. Examples include:
  - 500 Internal Server Error: A generic error message indicating that something unexpected happened on the server.
  - 502 Bad Gateway: The server received an invalid response from an upstream server while acting as a gateway or proxy.
  - **503 Service Unavailable:** The server is currently unable to handle the request due to temporary overloading or maintenance.

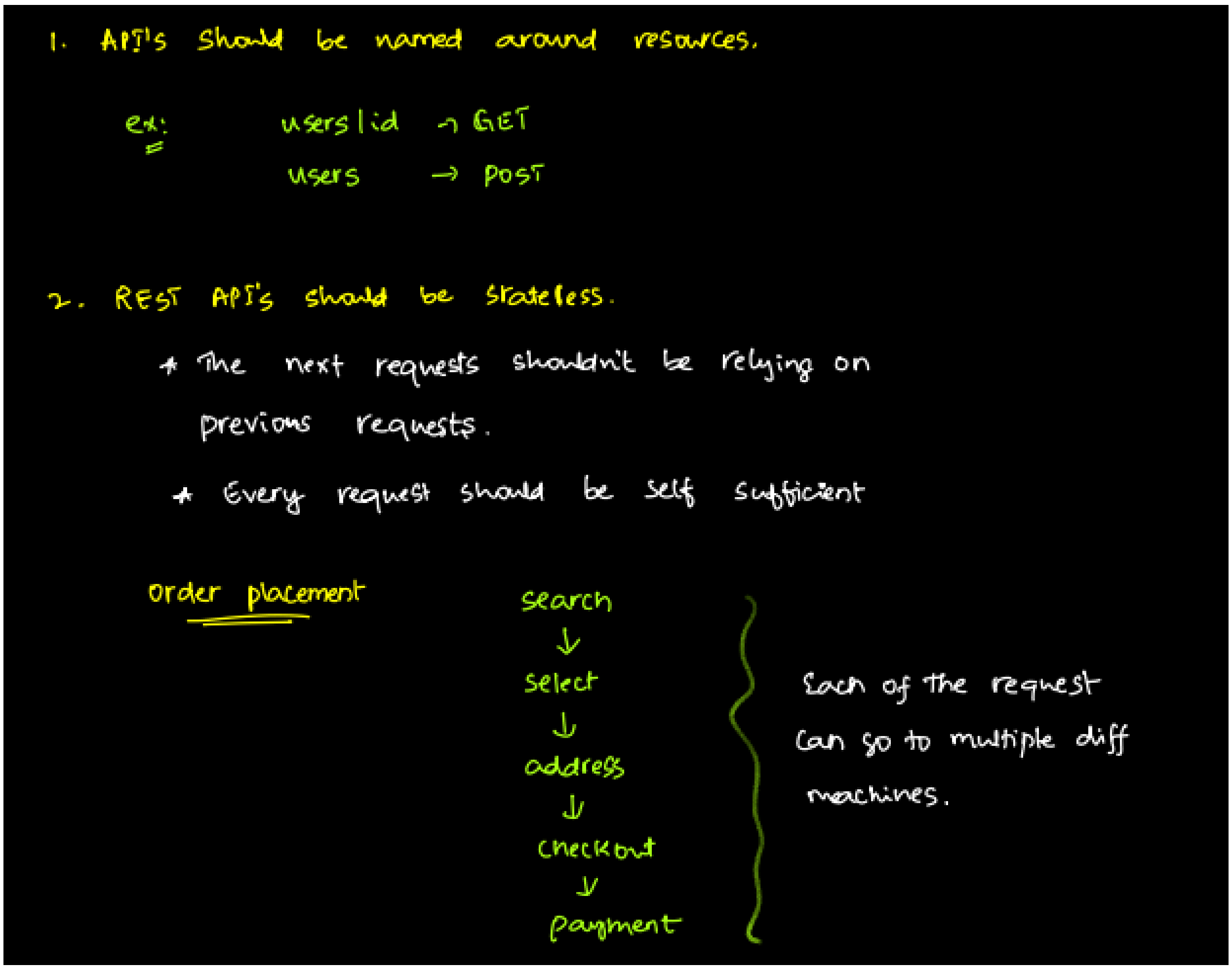
===

Rest conventions

===

REST conventions are a set of guidelines and best practices for designing and structuring RESTful APIs. Here's a summary of some common REST conventions:

1. **Resource Naming:** Use nouns to represent resources rather than verbs. For example, use "/users" instead of "/getUsers" to retrieve user data.
2. **Endpoint Naming:** Use plural nouns to represent collections and singular nouns to represent specific resources within a collection. For example, "/users" represents a collection of users, while "/users/{id}" represents a specific user.
3. **HTTP Methods:** Use HTTP methods to perform different actions on resources:
    a. GET: Retrieve resource representations.
    b. POST: Create new resources.
    c. PUT: Update existing resources (replace).
    d. PATCH: Partially update existing resources.
    e. DELETE: Remove resources.
4. **Use of Status Codes:** Use appropriate HTTP status codes to indicate the outcome of a request. For example, use 200 for successful responses, 404 for not found, 400 for bad requests, etc.
5. **Use of Headers:** Utilize HTTP headers for conveying additional information, such as authentication tokens, content type, caching directives, etc.
6. **Versioning:** If needed, include API versioning in the URL or headers to manage changes and backward compatibility.
7. **Error Handling:** Provide informative error messages in response bodies for failed requests, along with appropriate HTTP status codes.
8. **Pagination:** For large collections, implement pagination to limit the number of results returned in a single response.
9. **Filtering, Sorting, and Searching:** Allow clients to filter, sort, and search resources using query parameters in the URL.
10. **Hypermedia:** Optionally, include hypermedia links in responses to enable clients to navigate the API dynamically.



1. API's should be named around resources.

ex: users/id → GET
users → POST

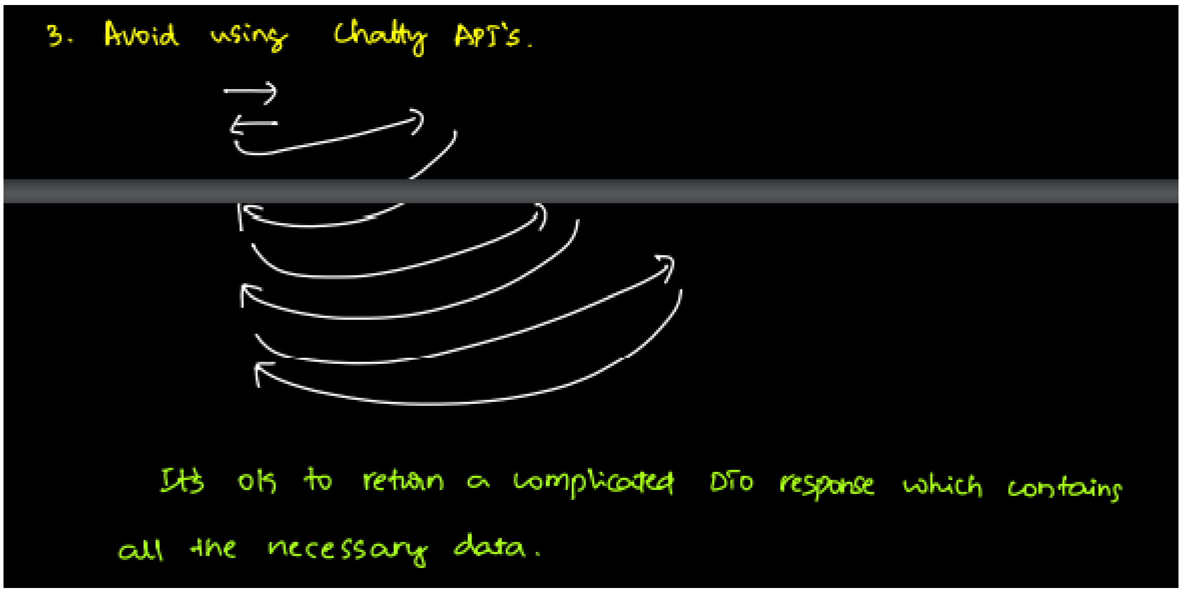
2. REST API's should be stateless.

* The next requests shouldn't be relying on previous requests.
* Every request should be self sufficient

order placement

search
↓
select
↓
address
↓
checkout
↓
payment

Each of the request can go to multiple diff machines.

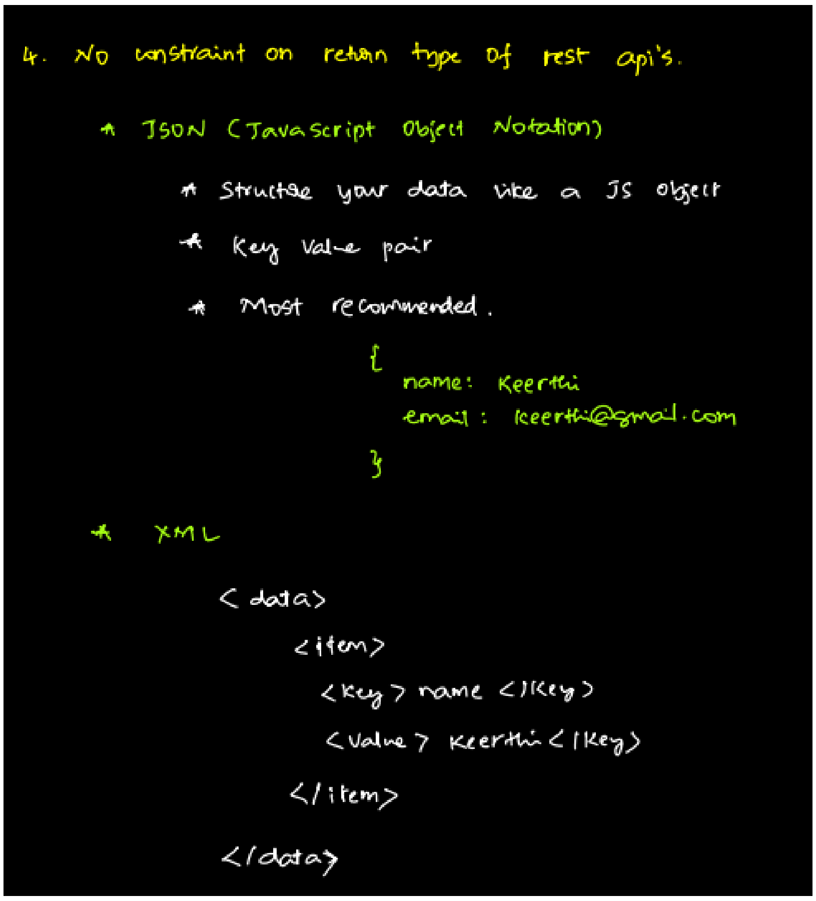


3. Avoid using Chatty API's.

It's ok to return a complicated DTO response which contains all the necessary data.

As highlighted in the second point, it's crucial for the system to be stateless. This means that a process shouldn't rely on specific machines, making it challenging to maintain if data is stored and shared across multiple machines.

In the third point, we emphasize the importance of avoiding chatty APIs. Chatty APIs involve making numerous small requests to get data, leading to increased HTTP calls. Instead, it's preferable to return complex responses, such as those seen on e-commerce websites like Flipkart, where clicking on an order provides detailed information without requiring multiple requests. Json is the most commonly used and recommended data type.


4. No constraint on return type of rest api's.

* JSON (Javascript Object Notation)

  * Structure your data like a JS object

  * Key value pair

  * Most recommended.

```
{
  name: Keerthi
  email : keerthi@gmail.com
}
```

* XML

```
<data>
    <item>
      <key> name </key>
      <value> Keerthi </key>
    </item>
</data>
```

To enable functionality in an API method, it is typically coded within a controller class, which is annotated with @RestController. This annotation ensures that the controller is treated as a RESTful resource provider. By implementing the necessary logic within this annotated controller, the API method becomes operational.

```
@GetMapping("/greet")
public String sayHello(RequestParam ("nane") String name)(
       return "Hello, Good Morning name";
}
@PostMapping("/create/something")
public Integer postExample( requestBody Emplyoee employee){
   return emloyee.getSalary(); // emloyee is the class
}
```

The **@JsonIgnoreProperties(ignoreUnknown=false)** annotation in Java with Jackson library configuration is used to instruct the deserializer to ignore any unknown properties found in the JSON data during deserialization.
By setting ignoreUnknown=false, it means that if there are any properties in the JSON data that cannot be mapped to fields in the Java object being deserialized, an exception will be thrown. This helps ensure that all properties in the JSON data are accounted for and mapped correctly to fields in the Java object.

```
@Getter
@Setter
@JsonIgnoreProperties(ignoreUnknown = false)
public class Employee {
   private String name;
   private String email;
   private int salary;
}
```

Backend LLD-4: Calling 3rd Party APIs - 2rd April

- **Classes for Product service and fake store APIs**
- **Create a new project**
- **Creating a product service**
- **ProductController**
- **Interface for service**
- **Using RestTemplate**
- **Autowiring rest template object**
- **getAllProducts**

==

Classes for Product service and fake store APIs

==

Product Catalog

2.1. Browsing: Users should be able to browse products by different categories.
2.2. Product Details: Detailed product pages with product images, descriptions, specifications, and other relevant information.
2.3. Search: Users must be able to search for products using keywords.

Classes ->
Product
----
id, image, description, specifications

Category
----
id, name

==

API Documentation for Products

==

for Fake data
https://fakestoreapi.com/docs

Get all products (Typically, all GET APIs utilize the GET method)

- Endpoint: https://fakestoreapi.com/products
- Response:

```
[
        {
                "id": 1,
                "title": "...",
                "price": "...",
                "category": "...",
                "description": "...",
                "image": "..."
        }
]
```

Get a single product

- Endpoint: https://fakestoreapi.com/products/1
- Response:

```
{
 "single same object of all product"
}
```

Limit results

- Endpoint: https://fakestoreapi.com/products?limit=5
- Response:
  [{same array of object of all product}]

Sort results

- Endpoint: https://fakestoreapi.com/products?sort=desc
- Response:
  [{same array of object of all product}]

Get all categories

- Endpoint: https://fakestoreapi.com/products/categories
- Response:

```
[
 "electronics",
 "jewelry",
 "men's clothing",
 "women's clothing"
```

Get products in a specific category

- Endpoint: https://fakestoreapi.com/products/category/jewelry
- Response:

```
[
        {
                "id": 5,
                "title": "...",
                "price": "...",
                "category": "jewelry",
                "description": "...",
                "image": "..."
        }
]
```

Add new product

- Endpoint: https://fakestoreapi.com/products
- method: ”POST”
- Request:

```
{
        "title": "test product",
        "price": 13.5,
        "description": "lorem ipsum set",
        "image": "https://i.pravatar.cc",
        "category": "electronics”
}
```

Update a product

- Endpoint: https://fakestoreapi.com/products/7
- method: "PUT" | "PATCH"
- Request:

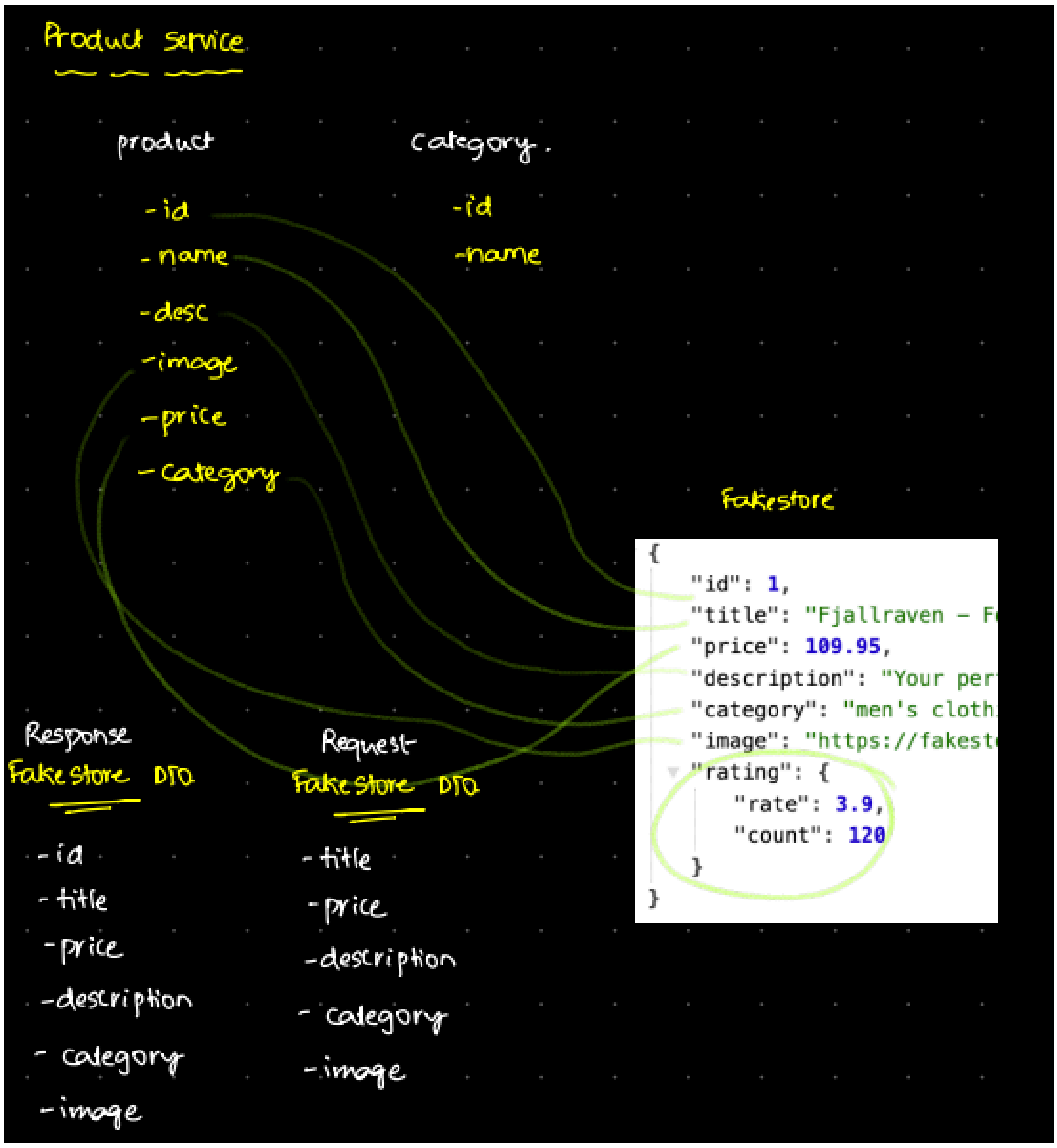
```
{
        "title": "test product",
        "price": 13.5,
        "description": "lorem ipsum set",
        "image": "https://i.pravatar.cc",
        "category": "electronics”
}
```

Delete a product

- Endpoint: https://fakestoreapi.com/products/6
- method:"DELETE"
- Response

```
 {
          id:6,
          title:'...',
          price:'...',
          category:'...',
          description:'...',
          image:'...'
   }
```

When engaging with another service, APIs are typically utilized. To align fields, we generate DTOs (Data Transfer Objects) such as RequestDTO and ResponseDTO, as illustrated below.



```
{
   "id": 1,
   "title": "Fjallraven - F
   "price": 109.95,
   "description": "Your per
   "category": "men's cloth
   "image": "https://fakest
 ▾ "rating": {
      "rate": 3.9,
      "count": 120
   }
}
```

==

Create a new project

==

A step-by-step guide to creating a new Spring Boot project using Spring Initializr and adding database dependencies:

Navigate to Spring Initializr:
Go to https://start.spring.io/

Configure Project:

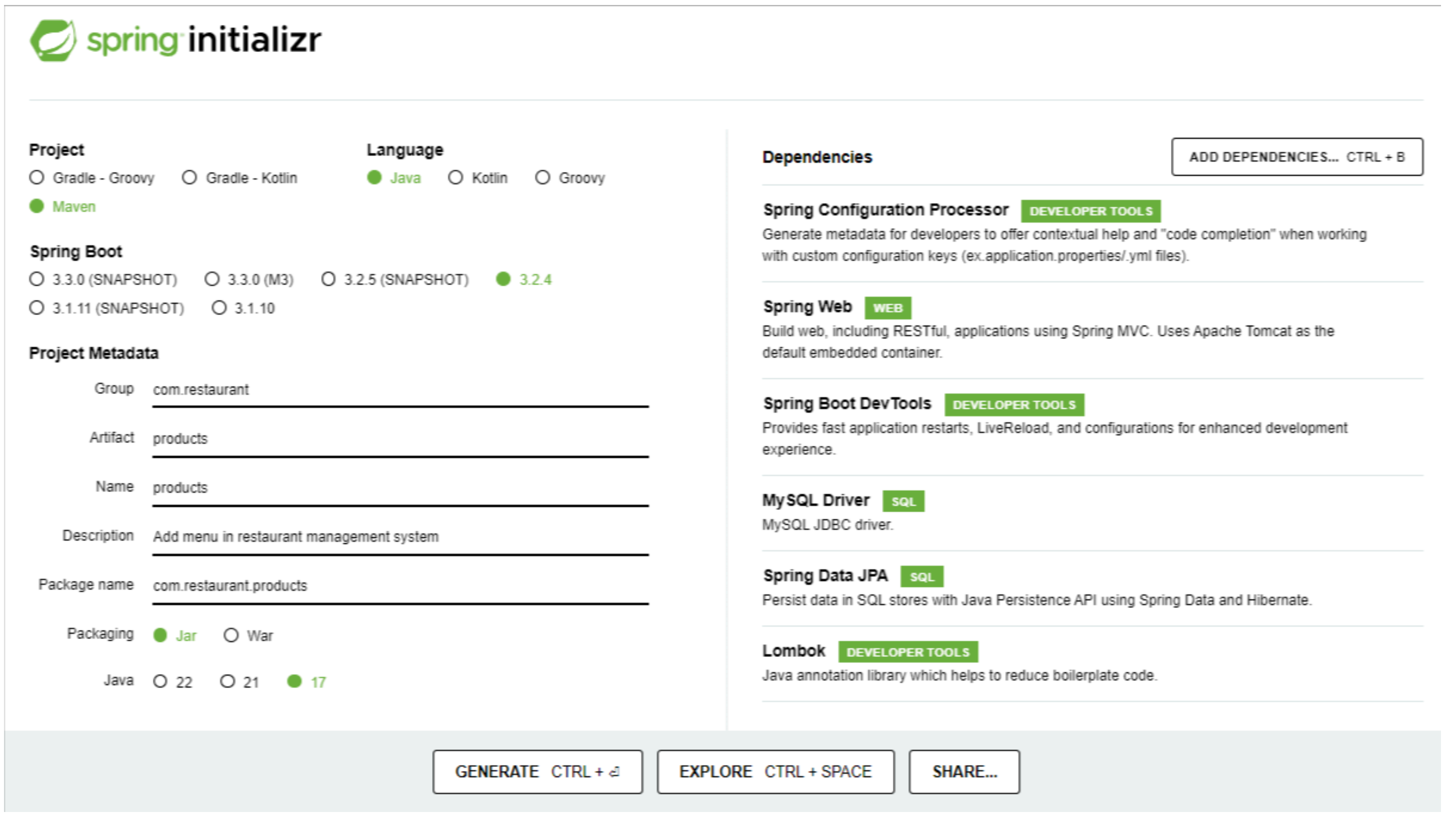
- Choose your project details like language (Java), Spring Boot version, and project metadata.
- Add project metadata like Group, Artifact, and Name.

Add Dependencies:

- Add dependencies for database access. For example, if you're using MySQL, select "**Spring Configuration Processor**", "**Spring Web**", "**Spring Boot DevTools**", "**MySQL Driver**", "**Spring Data JPA**", "**Lombok** ".
- You can also add other dependencies depending on your project requirements.

Generate Project:

Click on the "Generate" button to download the project zip file.



spring initializr

Project
Gradle - Groovy   Gradle - Kotlin   [✓ Maven]

Language
[✓ Java]   Kotlin   Groovy

Spring Boot
3.3.0 (SNAPSHOT)   3.3.0 (M3)   3.2.5 (SNAPSHOT)   [✓ 3.2.4]
3.1.11 (SNAPSHOT)   3.1.10

Project Metadata

Group	com.restaurant
Artifact	products
Name	products
Description	Add menu in restaurant management system
Package name	com.restaurant.products
Packaging	[✓ Jar] War
Java	22 21 [✓ 17]

Dependencies

Spring Configuration Processor DEVELOPER TOOLS
Generate metadata for developers to offer contextual help and "code completion" when working with custom configuration keys (ex.application.properties/.yml files).

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

MySQL Driver SQL
MySQL JDBC driver.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Extract the Project:

Extract the downloaded zip file to a directory of your choice.

Open in IDE:

Open your preferred IDE (like IntelliJ IDEA, Eclipse, or Visual Studio Code).

Import Project:

- In IntelliJ IDEA, select "Import Project" or open **pom.xml** file as a project and choose the directory where you extracted your project.
- In Eclipse, select "Import" > "Existing Maven Projects" and choose the project directory.

Create Database: Open Workbench and MySQL. Create a database and establish a connection with Workbench. Then, copy the server URL, database name, username, and password.

Configure Database Properties:

- Open **application.properties** or **application.yml** file in src/main/resources.
- Configure database connection properties like URL, username, and password.

```
spring.application.name=products
spring.datasource.url = jdbc:mysql://127.0.0.1:3306/productservice
spring.datasource.username = root
spring.datasource.password =
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
server.port=8080
```

Once you've connected to the database, run the app to ensure everything is working. Then, start by creating packages in the com.products directory.

Next First, let's create a **models** package. Inside this package, we'll create a **BaseModel** and Product class.

@MappedSuperclass -> It is a JPA annotation used to designate a superclass whose mapping information is applied to its subclasses.

When you annotate a class with @MappedSuperclass, it indicates that this class will not be mapped to its own table in the database, but its attributes will be inherited by its subclasses. This allows you to define common attributes or behavior that should be shared among multiple entities without the need for redundancy in the database schema.

In other words, @MappedSuperclass allows you to create a common base class for your entities, abstracting away common fields and behaviors, and then extending this base class to create concrete entity classes. The attributes and mappings defined in the superclass will be automatically applied to the subclasses, reducing duplication and promoting code reusability.

@Id -> It is a JPA annotation used to designate a field in a Java class as the primary key of an entity. When you annotate a field with @Id, it signifies that this field uniquely identifies each instance of the entity in the database.

For example, in a User entity class, you might have a field called "id" annotated with @Id. This indicates that each user entity will have a unique identifier stored in this field, and it will be used as the primary key when storing and retrieving data from the database.

The @Id annotation is crucial for entity mapping in JPA, as it allows JPA to determine how to uniquely identify and manage instances of entities in the database.

@GeneratedValue -> when you're working with databases, you often have tables with primary keys, which are unique identifiers for each row. The @GeneratedValue annotation helps automate the process of generating these unique identifiers.

The parameter **strategy = GenerationType.IDENTITY** specifies a particular strategy for generating these IDs. In this case, IDENTITY indicates that the database will automatically assign an ID to each new row as it's inserted into the table. This is commonly used with databases like MySQL where the database itself manages the auto-incrementing of primary keys. So, whenever a new row is added to the table, the database will assign it a unique ID automatically, starting from a specified initial value and incrementing by 1 for each new row.

BaseModel

```
@MappedSuperclass
@Getter
@Setter
public class BaseModel {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

Product

```
@Entity
@Getter
@Setter
public class Product extends BaseModel{

   private String description;
   private String image;
   private float price;

   @ManyToOne
   private Category category;
}
```

Category

```
@Entity
@Setter
@Getter
public class Category extends BaseModel{

}
```

Next Create dtos package and **productRequestDto** and **productReposneDto**

```
 @Getter
 @Setter
 public class ProductRequestDto {
   private String title;
   private float price;
   private String description;
   private String category;
   private String image;
 }
 @Getter
 @Setter
 public class ProductResponseDto {

   private Long id;
   private String title;
   private float price;
   private String description;
   private String category;
   private String image;
 }
```

Next Create controller package and create ProductController and create api methods

```
@RestController
public class ProductController {

}
```

Get all products

```
@GetMapping("/products")
public List<Product> getAllProducts(){
   return productService.getAllProducts();
}
```

@PathVariable -> Spring applications, when you define a controller method to handle HTTP requests, you might need to extract data from the request URL. The @PathVariable annotation helps you do this.

In the context of @PathVariable("id"), the id inside the parentheses specifies the name of the variable that you want to extract from the URL path. This variable's value will be passed to your controller method.

For example, if you have a URL like /users/123, where 123 is the ID of a user, you can use @PathVariable("id") to capture this value and use it in your controller method and if we have multiple variables like /users/123/45. then we can get defining 2 variables in method and get it sequtionaly like
public ResponseEntity<Post> getPostById(@PathVariable("userId") Long userId, @PathVariable("postId") Long postId) {}

Get single product

```
@GetMapping("/product/{id}")
public Product getSingleProduct(@PathVariable("id") Long id){
   return productService.getSingleProduct(id);
}
```

Get all categories

```
@GetMapping("/products/categories")
public List<Category> getAllCategories(){

   return new ArrayList<>();
}
```

Get all products in Category

```
@GetMapping("/products/category/{id}")
public List<Product> getAllProductsInCategory(@PathVariable("id") Long id){
   return new ArrayList<>();
}
```

Add product

```
@PostMapping("/products")
public Product addProduct(@RequestBody ProductRequestDto requestDto){
   return new Product();
}
```

Update product
```
@PatchMapping("/products/{id}")
public Product updateProduct(@PathVariable("id") Long id,
                        @RequestBody ProductRequestDto requestDto){
   return new Product();
}
```
Delete product
```
@DeleteMapping("/products/{id}")
public boolean deleteProduct(@PathVariable("id") Long id){
   return true;
}
```

Complete ProductController class

```
@RestController
public class ProductController {

   @Autowired
   IProductService productService;
   @GetMapping("/products")
   public List<Product> getAllProducts(){

       return productService.getAllProducts();
   }

   @GetMapping("/product/{id}")
   public Product getSingleProduct(@PathVariable("id") Long id){

       return productService.getSingleProduct(id);
   }

   @GetMapping("/products/categories")
   public List<Category> getAllCategories(){
       return new ArrayList<>();
   }

   @GetMapping("/products/category/{id}")
   public List<Product> getAllProductsInCategory(@PathVariable("id") Long id){
       return new ArrayList<>();
   }

   @PostMapping("/products")
   public Product addProduct(@RequestBody ProductRequestDto requestDto){
       return new Product();
   }

   @PatchMapping("/products/{id}")
   public Product updateProduct(@PathVariable("id") Long id,
                              @RequestBody ProductRequestDto requestDto){
       return new Product();
   }

   @DeleteMapping("/products/{id}")
   public boolean deleteProduct(@PathVariable("id") Long id){
       return true;
   }
}
```

Next Create service package and add interface IProductService and write signature of controller method Product getSingleProduct(Long id);

Long -> method signature getSingleProduct(@PathVariable **Long** id), it's preferable to use the object wrapper type Long instead of the primitive type long. Here's why:

Nullable Values Handling: If you use the primitive long, Spring won't be able to handle cases where the ID is not present in the URL, leading to potential runtime exceptions. However, by using the object wrapper type Long, Spring can handle null values gracefully, indicating that the ID is not provided in the URL.

Compatibility with Java Generics: Many Java APIs, including Spring, are designed to work with object wrapper types like Long for better compatibility with Java generics and collections. This consistency simplifies coding and enhances interoperability.

Ease of Use with Optional: If you're using Java 8 or later, you might prefer to use Optional<Long> instead of Long as the method parameter. This allows you to explicitly handle cases where the ID might be absent, providing clearer semantics and avoiding null pointer exceptions.

Convention and Consistency: In Java programming, it's a common convention to use object wrapper types (Long, Integer, etc.) instead of primitive types (long, int, etc.) for better handling of null values and compatibility with generics. Following this convention ensures consistency and makes your code more readable and maintainable.

```java
@Service
public interface IProductService {

  Product getSingleProduct(Long id);

}
```

Next step implement this interface and create a file FakestoreProductService and also create an instance of **IProductService** in **ProductController**.

start implementing **getSingleProduct** When we need to communicate with another REST API, we use something called RestTemplate provided by SpringBoot to interact with other services using API. Inside the FakeStoreProductService, in the getSingleProduct method, we'll make a RestTemplate object. Then, we'll use that object to call the getForObject method, which requires two parameters: the URL and the class type of ProductResponseDto. This class helps convert the data received into the ProductResponseDto format, and we'll store the response in a variable of type ProductResponseDto.

```java
@Service
public class FakestoreProductService implements IProductService{

   @Autowired
   RestTemplate restTemplate;

   @Override
   public Product getSingleProduct(Long id) {
       ProductResponseDto response = restTemplate.getForObject(
               "https://fakestoreapi.com/products/" + id,
               ProductResponseDto.class);

       return getProductFromResponseDTO(response);
   }
}
```

In the getSingleProduct method, we need to return a product, and our return type is ProductResponseDto. To get the product from ProductResponseDto, we'll create another method called getProductFromResponseDto in the same file. Inside this method, we'll call the getSingleProduct function and return the result.

Create a product object in getProductFromResponseDto and set all fields from response get as parameter and return the product.

Note: We're not receiving the category ID in the response. That's why we're creating a new category. Then, we extract the category from the product and set its name based on the information from the response.

```
private Product getProductFromResponseDTO(ProductResponseDto response) {

   Product product = new Product();
   product.setId(response.getId());
   product.setName(response.getTitle());
   product.setDescription(response.getDescription());
   product.setPrice(response.getPrice());
   product.setCategory(new Category());
   product.getCategory().setName(response.getCategory());
   product.setImage(response.getImage());

   return product;
}
```

There's an issue with creating a **RestTemplate** object in this method because if we need to create another method, we'd have to rewrite it, which isn't efficient. One solution could be to use **@Autowired**, but if it's not part of our codebase, it won't register as a Spring bean. Another solution is to create the **RestTemplate** object in the controller, where it's created once for all methods. However, if we need to call RestTemplate in another microservice, using @Autowired will throw an error because RestTemplate is a library and not marked with annotations like @RestController, @Service, or @Entity. Spring Boot won't create an object for it automatically.

To resolve this, when using a 3rd-party class as a **bean** in our codebase, we need to create another class. In this new class, we define beans that we want to use. We'll create a package called "**config**" and within it, a class named "ApplicationConfiguration". We'll annotate it with @Component and create a method named "createRestTemplate" that returns a RestTemplate object. We'll mark this method with **@Bean** to indicate that it's a bean.

What is Bean -> @Bean is an annotation used in the Spring Framework to indicate that a method returns a bean object managed by the Spring container. When Spring scans the application context for beans to manage, it identifies methods annotated with @Bean and registers them as bean definitions. These methods typically create and configure instances of classes that are required within the application. By using @Bean, developers can explicitly define beans and their configurations in Java code, offering more flexibility and control over the dependency injection process.

```
@Component
public class ApplicationConfiguration {

   @Bean
   public RestTemplate createRestTemplate(){
       return new RestTemplate();
   }
}
```

Switch back to **IProductService** interface and Create another method in List<Product> getAllProducts();

```java
@Service
public interface IProductService {

  Product getSingleProduct(Long id);
  List<Product> getAllProducts();
}
```

Next Step Now implement getAllProducts in class FakestoreProductService

When implementing the getAllProducts method in the FakestoreProductService using the restTemplate object, it returns a List of objects. However, we encounter a **compile-time error** when trying to convert it to a class using List<ProductResponseDto>.class. This error occurs due to type erasure, meaning that at **runtime, all lists are treated the same as List even it is object of list or string of list.**

So why does List<ProductResponseDto>.class fail to convert? It's because at runtime, there's no distinction between different types of lists; they're all just lists. We're attempting to specify a ResponseDto type when what we're actually getting is simply a list. This mismatch causes the conversion to fail.

The solution lies in adhering to Java's array conventions. Instead of using List<ProductResponseDto>.class, we should use ProductResponseDto[].class. This adjustment allows for successful conversion. Now, after making this change, we iterate through the response array and convert its elements into a list, which we then return as the list of objects.

```java
@Service
public class FakestoreProductService implements IProductService{
   @Autowired
   RestTemplate restTemplate;
   @Override
   public Product getSingleProduct(Long id) {
       ProductResponseDto response = restTemplate.getForObject(
               "https://fakestoreapi.com/products/" + id,
               ProductResponseDto.class);
       return getProductFromResponseDTO(response);
   }

   @Override
   public List<Product> getAllProducts() {
       ProductResponseDto[] products = restTemplate.getForObject(
               "https://fakestoreapi.com/products/",
               ProductResponseDto[].class);
       List<Product> output = new ArrayList<>();
       for(ProductResponseDto productResponseDto: products){
           output.add(getProductFromResponseDTO(productResponseDto));
       }
       return output;
   }
   private Product getProductFromResponseDTO(ProductResponseDto response) {
       Product product = new Product();
       product.setId(response.getId());
       product.setName(response.getTitle());
       product.setDescription(response.getDescription());
       product.setPrice(response.getPrice());
       product.setCategory(new Category());
       product.getCategory().setName(response.getCategory());
       product.setImage(response.getImage());
       return product;
   }
}
```

Backend LLD-4: Handling Exceptions & Introduction to Spring Data JPA - 4th April

- **How to return a response code from controller**
- **ResponseEntity**
- **Handling Exceptions**
- **@ExceptionHandler**
- **Controller Advice -> @RestControllerAdvice**
- **Introduction to DB interaction in spring boot**
- **Understanding application.properties**
- **How JPA can write implementations for any table and any column**
- **Spring Data JPA Hands on**
- **How to write custom methods in Repository**

Repository link: https://github.com/kkumarsg/productService

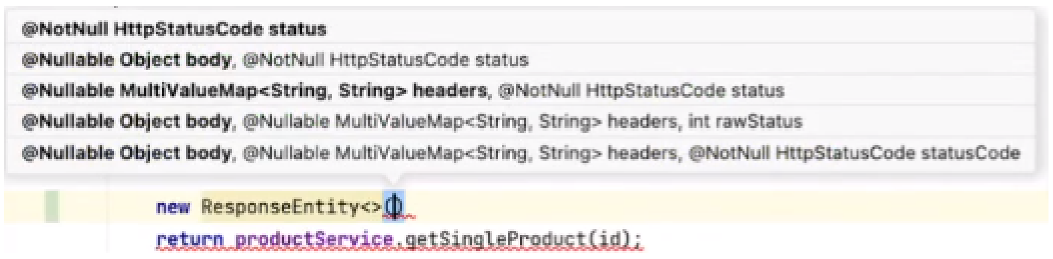
==

How to return a response code from controller

==

The current project implementation has a problem because it doesn't handle errors properly. This isn't the best way to write code because it could expose our code to the public in the form of errors. To fix this, we'll use a ResponseEntity. It wraps around the object we return, helping us manage errors more effectively.

Response entity ->
A ResponseEntity in Spring Framework represents the entire HTTP response: status code, headers, and body. It's a generic type that allows you to specify the type of the body that you expect to receive.
For example, if you're making a RESTful API call and expecting a JSON response, you might define your ResponseEntity as ResponseEntity<MyObject>, where MyObject is the class representing the JSON structure you expect to receive.
There are different type of ResponseEntity constructors



```
@NotNull HttpStatusCode status
@Nullable Object body, @NotNull HttpStatusCode status
@Nullable MultiValueMap<String, String> headers, @NotNull HttpStatusCode status
@Nullable Object body, @Nullable MultiValueMap<String, String> headers, int rawStatus
@Nullable Object body, @Nullable MultiValueMap<String, String> headers, @NotNull HttpStatusCode statusCode

        new ResponseEntity<>()
        return productService.getSingleProduct(id);
```

In Spring Framework, ResponseEntity is a class that represents an HTTP response, allowing you to control the response status, headers, and body. There are different constructors available to create instances of ResponseEntity based on different requirements:

1. **ResponseEntity(HttpStatus status):** Creates a new ResponseEntity with the specified HTTP status. The response body and headers will be empty.
2. **ResponseEntity(T body, HttpStatus status)**: Creates a new ResponseEntity with the specified body and HTTP status. The response headers will be empty.

3. **ResponseEntity(MultiValueMap<String, String> headers, HttpStatus status):** Creates a new ResponseEntity with the specified headers, body, and HTTP status.
4. **ResponseEntity(T body, MultiValueMap<String, String> headers, HttpStatus status):** Creates a new ResponseEntity with the specified body, headers, and HTTP status.
5. **ResponseEntity(T body):** Creates a new ResponseEntity with the specified body. The HTTP status will be set to 200 OK, and the response headers will be empty.
6. **ResponseEntity<T>:** A generic type constructor allowing you to specify the type of the body. This is used in conjunction with one of the above constructors to create an instance of ResponseEntity with a specific body type.

These constructors provide flexibility in creating ResponseEntity instances with different combinations of body, headers, and status codes, allowing you to customize HTTP responses according to your application's needs

We can use ResponseEntity like below in ProductController file

```
@getMapping("/product/{id}")|
public ResponseEntity<Product> getSingleProduct(@PathVariable("id") Long id){
   Product product;
   try{
           product = productService.getSingleProduct(id);
   }catch (Exception e) {
       return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
   }
       return new ResponseEntity<>(product, Httpstatus.NoT_FOUND);
}
```

Instead of simply returning a 404 status code, we can enhance it by including a message along with the status code. This way, instead of returning the product response directly from the getSingleProducts method, we'll create another DTO in dto package called ProductResponseSelf. we will create 2 fields product and message type of Product and String and we will assign it using constructor.

```
@Getter
@Setter
@AllArgsConstructor
public class ProductResponseSelf {

   private Product product;
   private String message;
}
```

After creating the DTO, we'll return it from the product controller's getSingleProduct method by creating an object and passing the product along with a message. In case of an error, we'll wrap this DTO and the status code in a ResponseEntity within the catch block. After the catch block, we'll return a ResponseEntity with the first parameter being a new ProductResponseSelf containing the product and a success message, and the second parameter being the status code.

```
@GetMapping("/product/{id}")
public ResponseEntity<ProductResponseSelf> getSingleProduct(@PathVariable("id") Long id){
   Product product;
   try{
       product = productService.getSingleProduct(id);
   }catch (ProductNotPresentException e){
       ProductResponseSelf productResponseSelf = new ProductResponseSelf(null, "Product
Doesn't exist");
       return new ResponseEntity<>(
               productResponseSelf, HttpStatus.NOT_FOUND);
   }catch (ArithmeticException e){
       ProductResponseSelf productResponseSelf = new ProductResponseSelf(null, "Something
went wrong");
       return new ResponseEntity<>(
               productResponseSelf, HttpStatus.INTERNAL_SERVER_ERROR);
   }

   return new ResponseEntity<>(new ProductResponseSelf(product, "Success"), HttpStatus.OK);
}
```

Currently, in the controller, we catch every type of exception and return it as a 404. However, this approach isn't correct because the exception could be something specific like a null pointer. Instead, we want to throw our custom error. To do this, we'll create an exceptions package and inside it, create a class called ProductNotPresentException.

Note: Currently, the third-party service doesn't provide an error status code. That's why we're using ResponseEntity. If the API starts returning status codes, we'll need to handle them directly.

```
public class ProductNotPresentException extends Throwable {
}
```

use this exception in a controller catch block like

```
}catch (ProductNotPresentException e){
}catch (ArithmeticException e){
}
```

==

Handling Exceptions

==

There's still an issue where we're adding catch blocks in every method, which isn't ideal. To address this, we'll introduce an exception handler. First, we remove the try-catch blocks from the controller methods and add the exception in the method signature, so it's thrown directly to the UI. Then, we'll use **@ExceptionHandler(ProductNotPresentException.class)** to handle this specific exception. We'll write a method below it to specify what to return when this exception occurs.

```
@ExceptionHandler(ProductNotPresentException.class)
public ResponseEntity<ProductResponseSelf> hndleInvalidProduct(){
    ProductResponseSelf productResponseSelf = new ProductResponseSelf(null, "Product not
found");
    return new ResponseEntity<>(productResponseSelf, HttpStatus.NOT_FOUND);
}
```

==

Controller Advice -> @RestControllerAdvice

==

Now, after separating business logic and exception handling, it's cleaner, but we can further improve it by managing exceptions in a separate class instead of in every controller. This is where Controller Advice comes in. We'll create a separate package called controlleradvice and within it, a class named ProductControllerAdvice. We'll move all the exception-handling logic from the controllers to this class. Above the class name, we'll add the annotation @RestControllerAdvice. That's it. Whenever any controller throws these errors, the flow will come here. It will act like a default exception handler. whenever any exception in any controller occurs it will run this method first.

```
@RestControllerAdvice
public class ProductControllerAdvice {

   @ExceptionHandler(ProductNotPresentException.class)
   public ResponseEntity<ProductResponseSelf> handleInvalidProduct(){
       ProductResponseSelf productResponseSelf = new ProductResponseSelf(null,
               "Product Doesn't exist from contorller advice ");
       return new ResponseEntity<>(
               productResponseSelf, HttpStatus.NOT_FOUND);
   }

   @ExceptionHandler(ArithmeticException.class)
   public ResponseEntity<ProductResponseSelf> handleArithmaticException(){
       ProductResponseSelf productResponseSelf = new ProductResponseSelf(null,
               "Something went wrong from contorller advice ");
       return new ResponseEntity<>(
               productResponseSelf, HttpStatus.INTERNAL_SERVER_ERROR);
   }
}
```

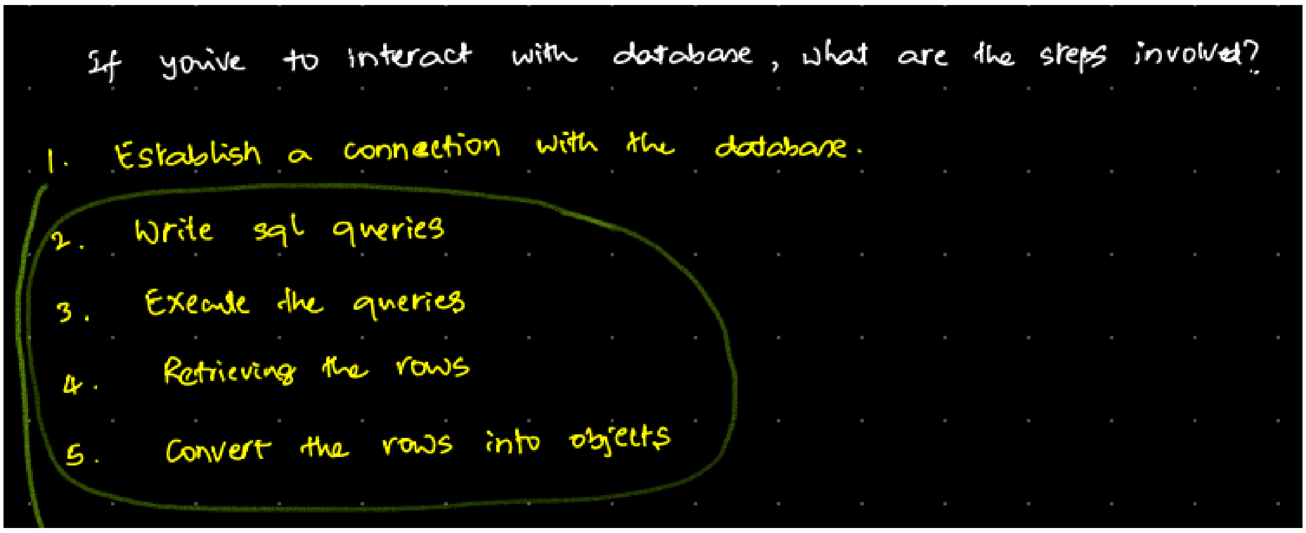
If we need to override any exception handling logic, we can simply write the same logic in the specific controller where we want the override to occur.

```
 @ExceptionHandler(ArithmeticException.class)
   public ResponseEntity<ProductResponseSelf> handleArithmaticException(){
       ProductResponseSelf productResponseSelf = new ProductResponseSelf(null,
               "Something went wrong from contorller advice ");
       return new ResponseEntity<>(
               productResponseSelf, HttpStatus.INTERNAL_SERVER_ERROR);
   }
```

==

Introduction to DB interaction in spring boot

==



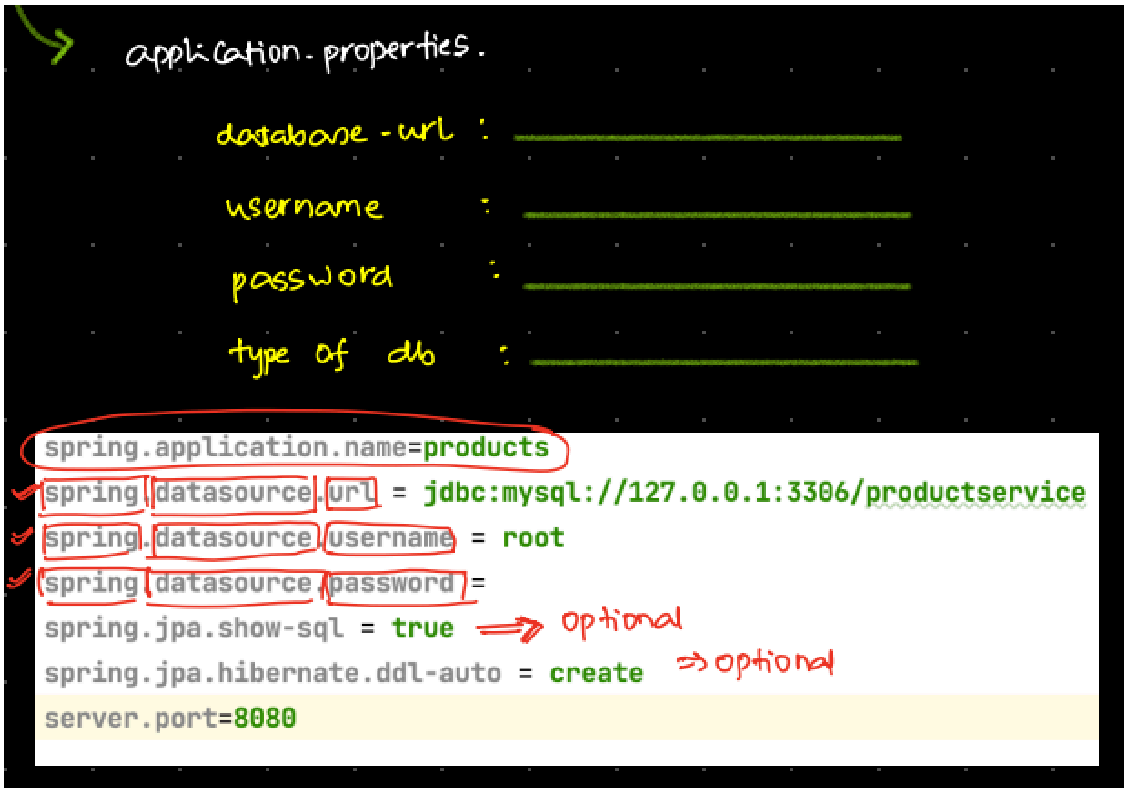
If you've to interact with database, what are the steps involved?

1. Establish a connection with the database.
2. Write sql queries
3. Execute the queries
4. Retrieving the rows
5. Convert the rows into objects

==

Understanding application.properties

==



application.properties.

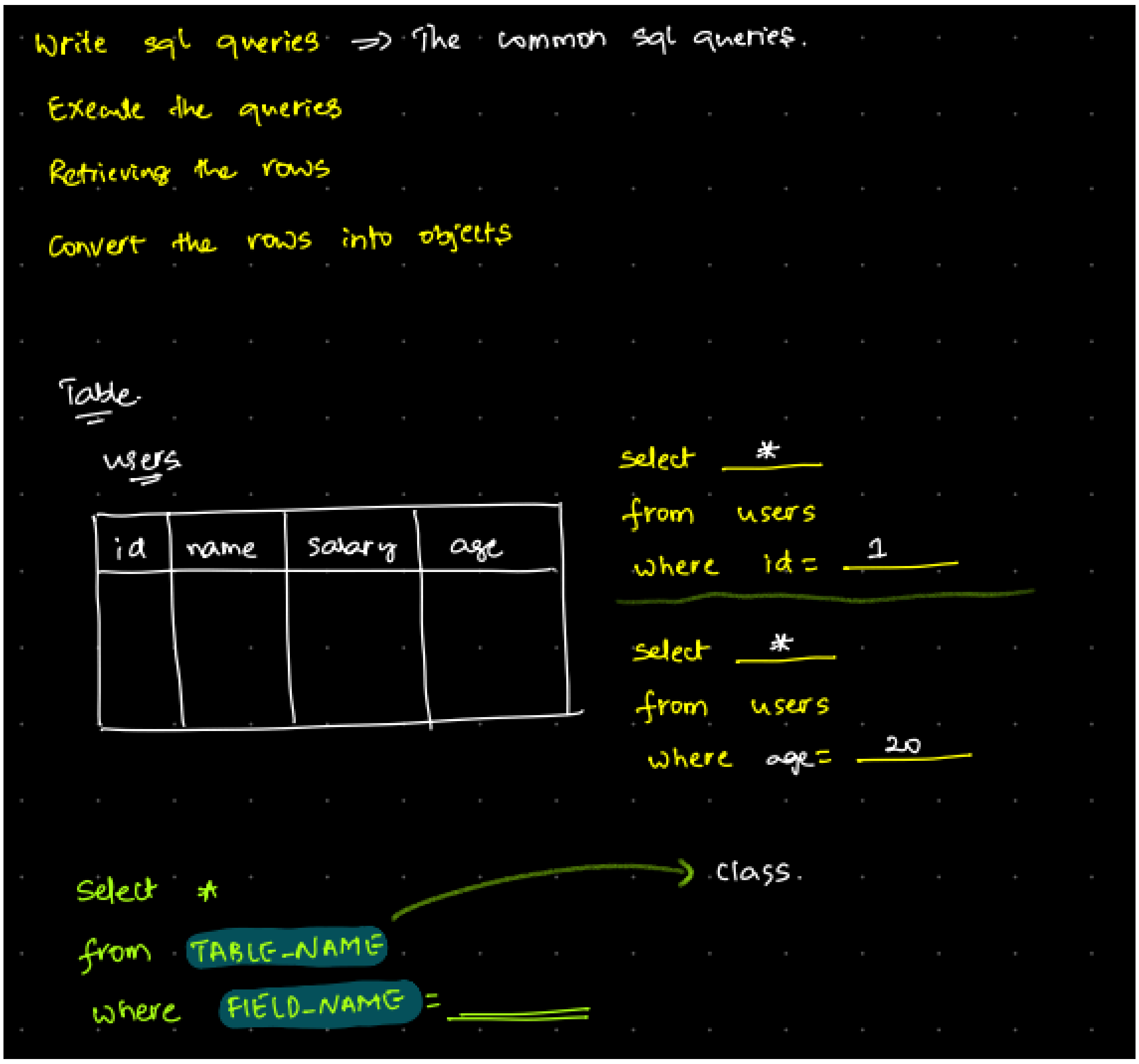
database-url : ______

username : ______

password : ______

type of db : ______

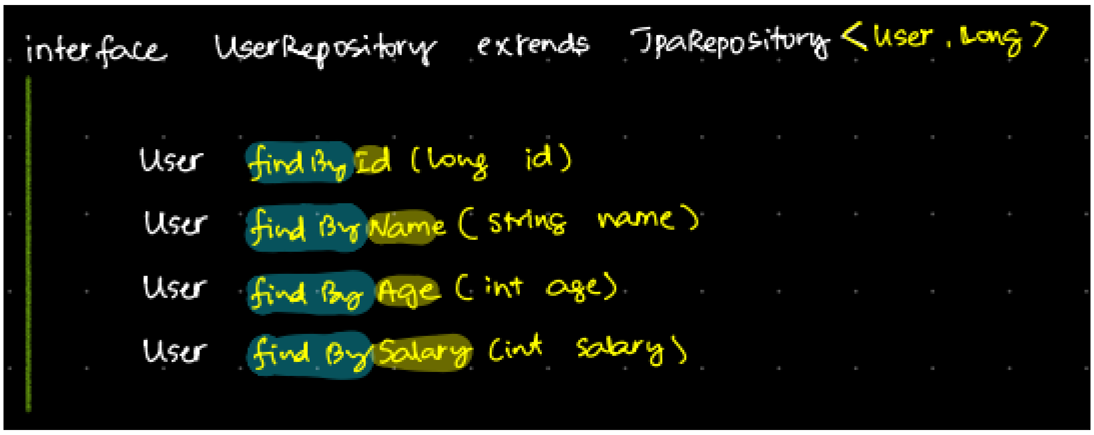
```
spring.application.name=products
spring.datasource.url = jdbc:mysql://127.0.0.1:3306/productservice
spring.datasource.username = root
spring.datasource.password =
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = create
server.port=8080
```



==
How JPA can write implementations for any table and any column
==

When interacting with a database using queries, we often notice common patterns that can be generalized. Instead of writing the same query repeatedly, we can use JPARepository to address this issue.

To utilize JpaRepository, we extend it in our repository interface and provide two parameters: the class name and the primary key type, such as <User, Long>. By doing this, we gain access to default methods that allow us to fetch data easily.



```
interface UserRepository extends JpaRepository<User, Long>

    User findById (Long id)
    User findByName (String name)
    User findByAge (int age)
    User findBySalary (int salary)
```

==

Spring Data JPA Hands on

==

Next, create a package named Repositories and within it, an interface called ProductRepository. Annotate it with @Repository and extend JpaRepository<Product, Long>.

After that, in the ProductController, create an instance of ProductRepository using @Autowired. Then, in each method, start by calling the required method using ProductRepository, such as findById.

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

==

How to write custom methods in Repository

==

Spring JPA Repository Documentation
https://docs.spring.io/spring-data/jpa/docs/1.6.0.RELEASE/reference/html/jpa.repositories.html

If we want to use any custom method like findByDescription or any other operation, we have to define its signature in the repository interface. The method signature should follow the same naming convention, such as Product findByDescription(String description).

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

   Product findByDescription(String description);
   Product findByName(String name);
   Product findByPrice(Float price);
}
```

Now implement this in the controller

```
@GetMapping("/products/search")
public Product getProductsByName(@RequestParam("name") String name){
   return productRepository.findByName(name);

}
```

@RequestParam -> It is a Spring annotation used to bind a web request parameter to a method parameter in a controller method. It allows you to extract values from the request URL's query parameters and pass them as arguments to your controller methods.

@PathVariable("id") -> It is a Spring annotation used to extract values from the URL's path and map them to method parameters in a controller method. It allows you to retrieve dynamic

parts of the URL and use them as arguments in your controller methods. In this specific example, "id" is the name of the path variable that you want to extract from the URL.
The **main difference between @RequestParam and @PathVariable** is in how they extract information from the request:

1. @RequestParam: This annotation extracts data from the query parameters in the URL. Query parameters appear after the "?" in a URL and are in the form of key-value pairs (e.g., ?key1=value1&key2=value2). You use @RequestParam to access these parameters.
2. @PathVariable: This annotation extracts data from the path of the URL. The path is the part of the URL that comes after the domain name. Path variables are dynamic segments of the URL's path, enclosed in curly braces (e.g., /users/{id}). You use @PathVariable to capture these dynamic parts of the URL.

In summary, @RequestParam deals with query parameters, while @PathVariable deals with dynamic parts of the URL's path.

Backend LLD-4: Repositories, UUIDs, Representing Inheritance - 6th April
- **UUID**
- **Problems with integer id - Can be guessed**
- **Problems with integer id - Distributed DB**
- **The idea behind UUID**
- **UUID how they're stored**
- **Representing Inheritance**
- **Single table**

UUID -> One solution is to use UUIDs (Universally Unique Identifiers) for IDs. UUIDs are long strings of characters that are guaranteed to be unique across different systems. They provide a way to generate IDs that don't rely on sequential numbers and are not easy to guess. This can help avoid data privacy issues associated with exposing IDs to the public. Additionally, UUIDs can handle large amounts of data without running into storage limitations like integer IDs

In the table below, we see that using email as the primary key isn't a good idea. This is because emails can change, and storing them as strings takes up a lot of space and can be costly to search through.

One solution is to create a separate column for IDs using integers. However, there's a problem with this approach because storing long IDs, like those exceeding 2 billion, isn't possible.

Another solution is to use a long data type for IDs, which can handle large amounts of data. But there are issues with this too. Since IDs are numbers, there's a risk of exposing them to the public. If IDs are auto-incremented, it's easy to guess the next ID, which can lead to data privacy problems. However, if IDs aren't publicly visible, this might not be a concern.

UUID : Universally unique identifier

users

email	pass	phNo	name

Can email be pK => Yes

is it a good idea => No.

Issues

1. Email id can change
2. Datatype of email id is string.

Solⁿ: have a separate id column.

1. integer => 4 Bytes , $2^{32} = 2$ Billion.

   Sometimes not enough.

2. long => 8 Bytes , $2^{64} \simeq 10^{18}$, mostly fine.

Problem 1: What if 'id' needs to be exposed in url.

ex: flipkart.com /search / products/1 ; GET

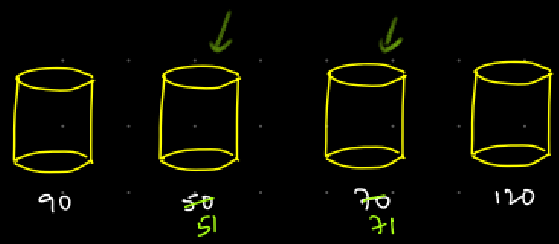
flipkart.com /search / products/2 ; GET

flipkart.com /search / products/3 ; GET

Data violation problem.

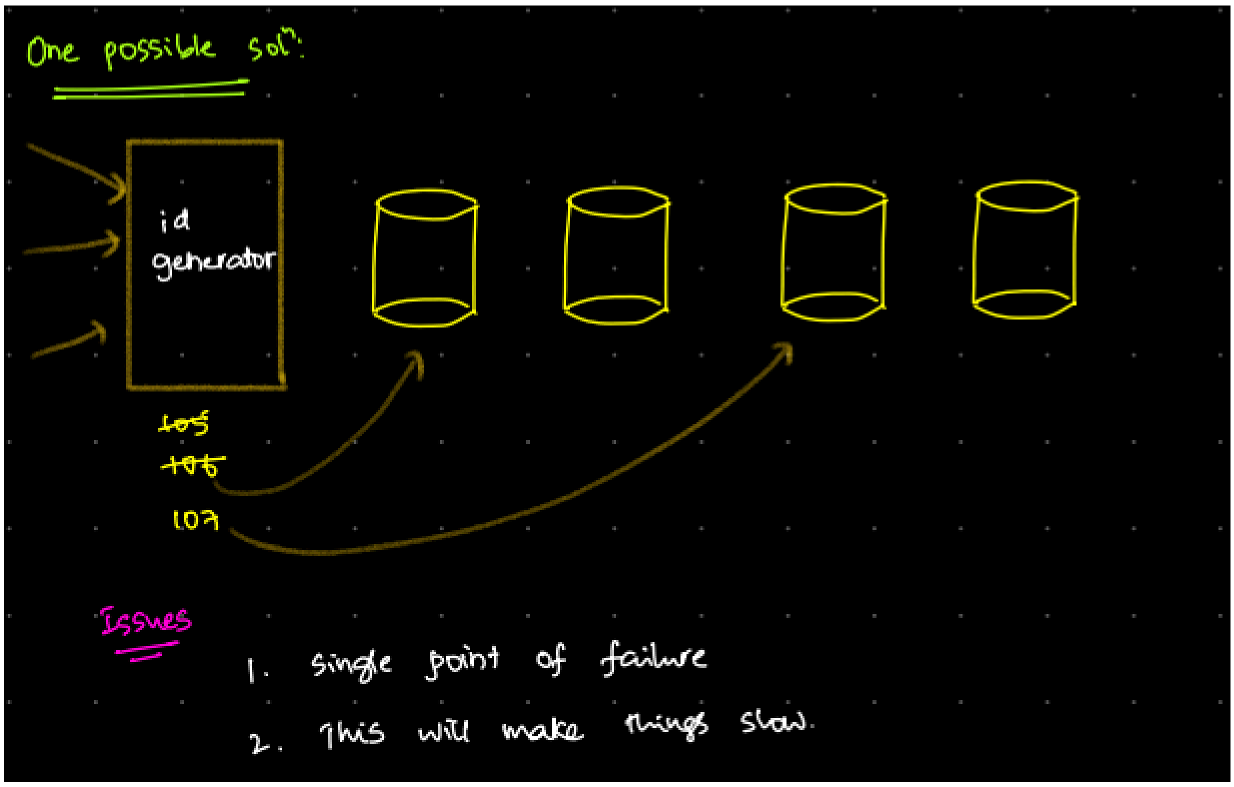
' If your id is not publicly visible, then it's still fine.

Problem 2: What if we've distributed DB.

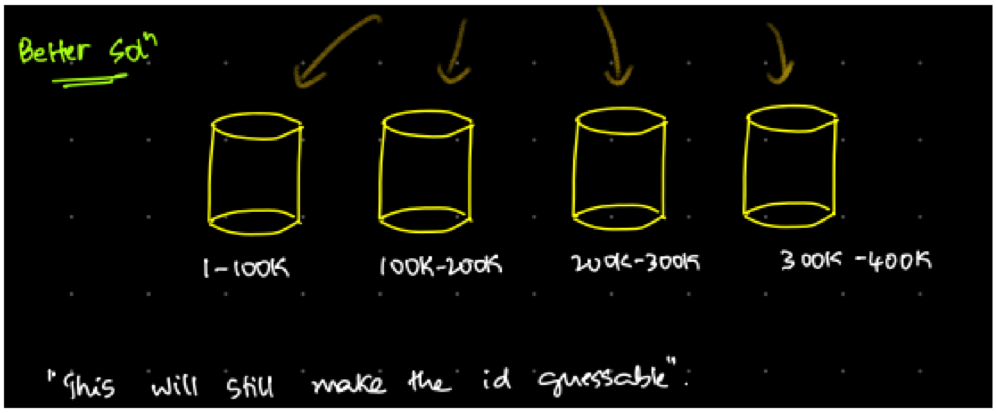


There can be a situation, where same id can be used.

Long data type can solve the problem, but using it in distributed systems, such as multiple databases, can be challenging to maintain unique IDs.



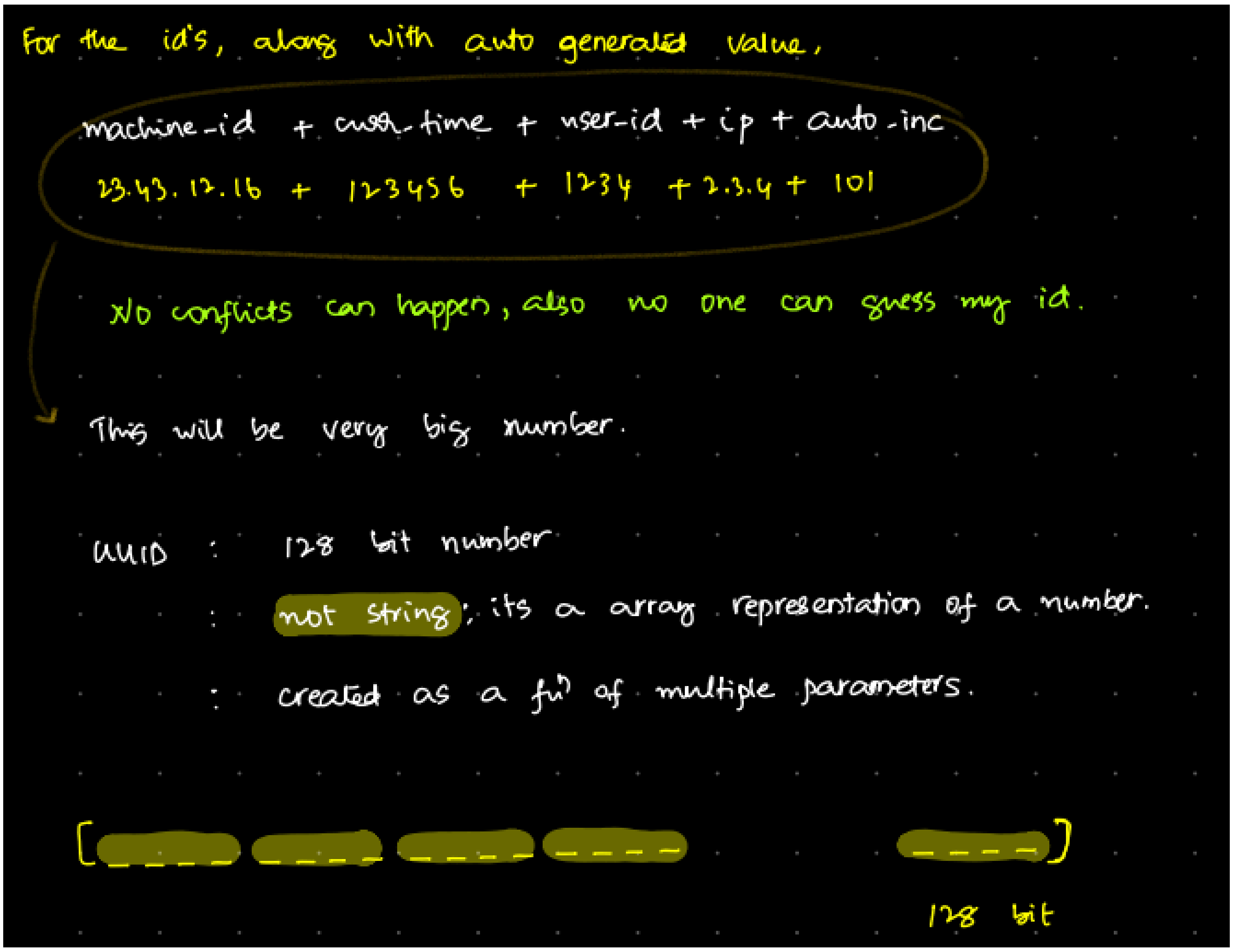
Another solution is to have a shared resource for generating database IDs. However, this approach comes with its own set of problems. One issue is the risk of a single point of failure, meaning if this shared resource fails, it can disrupt the entire system. Additionally, because we need to maintain a sequence of IDs, it can slow down operations as we have to take a lock, which can cause delays.



To solve the previous issue, we can assign a limit to each database to prevent overlaps. However, the problem of predictability still remains.

UUID -> uuidgenerator.net

To address this issue, we'll combine machine_id, curr_time, user_id, IP, and auto-increment together, forming what's known as a UUID (Universally Unique Identifier). This number is quite large and is stored using 128 bits. Essentially, a UUID is a unique 128-bit number generated from a specific combination of these elements and it is not a string it is array representation of number. It will store hexa-decimal number system.


For the id's, along with auto generated value,

machine-id + curr-time + user-id + ip + auto-inc

23.43.12.16 + 123456 + 1234 + 2.3.4 + 101

No conflicts can happen, also no one can guess my id.

This will be very big number.

UUID : 128 bit number

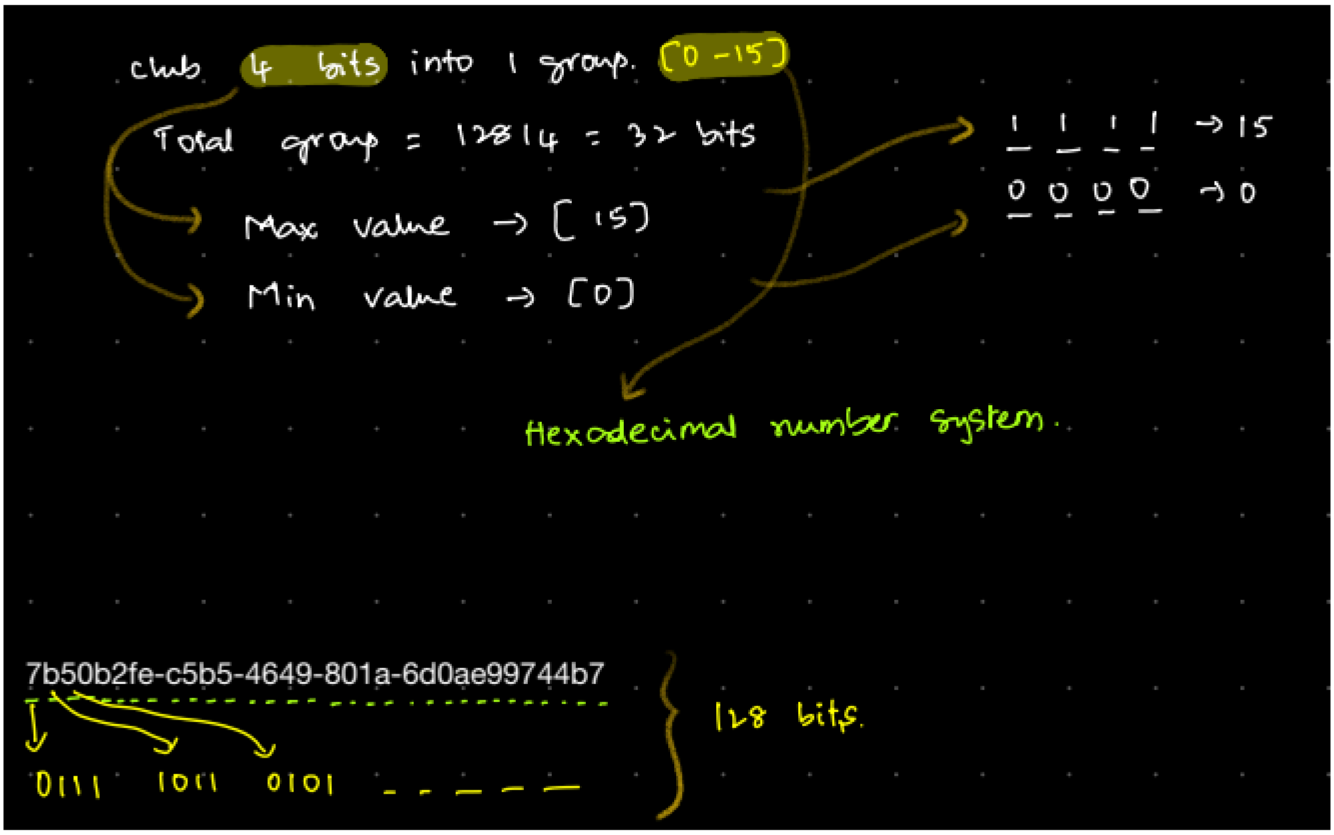
: not string; its a array representation of a number.

: created as a fn of multiple parameters.

[_]

128 bit

UUID how they're stored -> Representation of Hexa decimal number system


club 4 bits into 1 group. [0 - 15]

Total group = 128/4 = 32 bits

Max value -> [15]

Min value -> [0]

1 1 1 1 -> 15

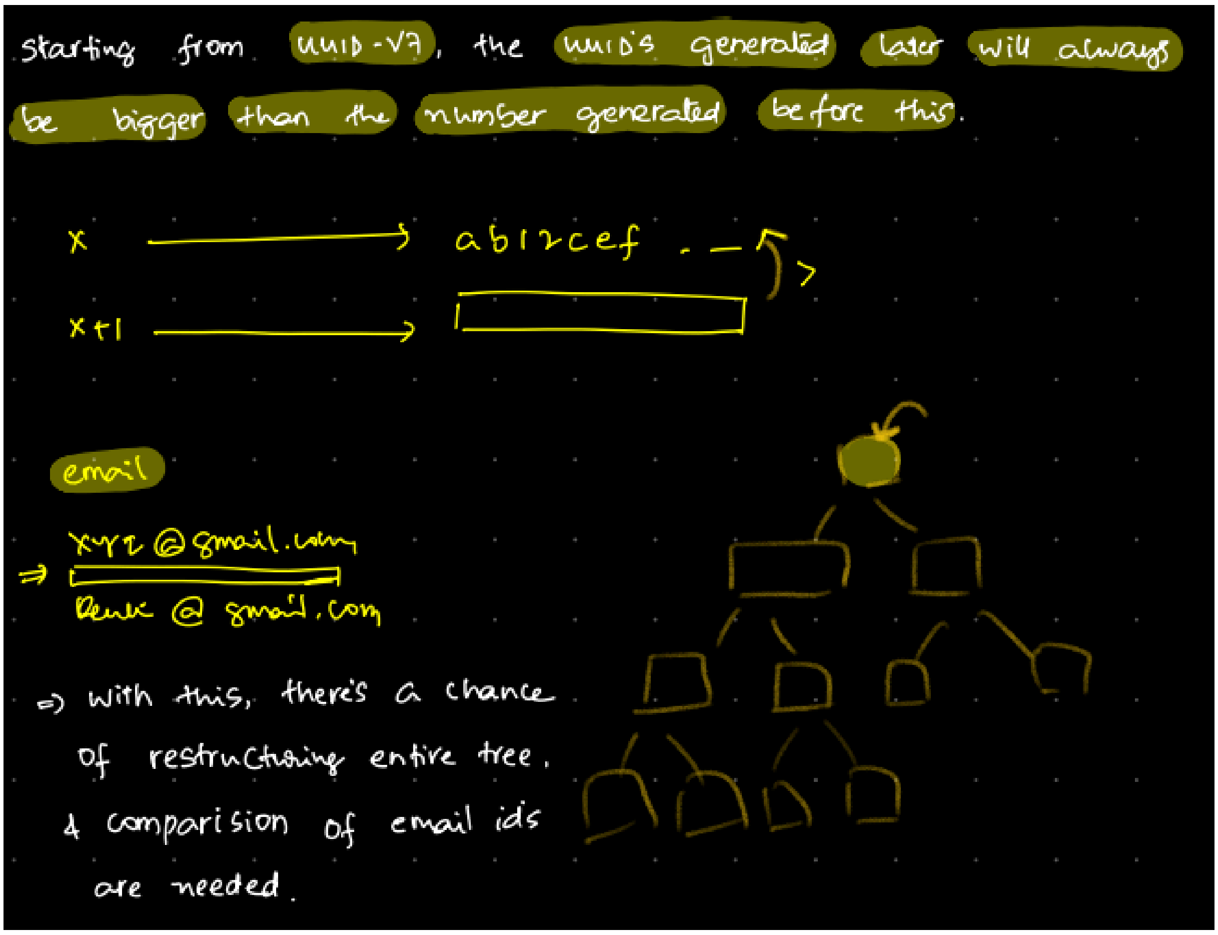
0 0 0 0 -> 0

Hexadecimal number system.

7b50b2fe-c5b5-4649-801a-6d0ae99744b7

0111 1011 0101 _ _ _ _ _

128 bits.

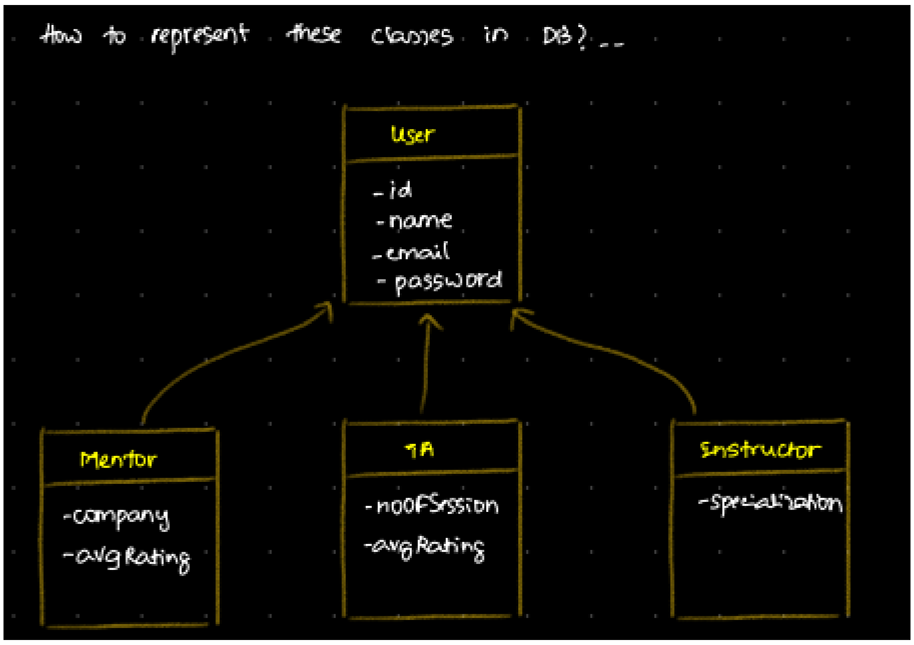

starting from UUID-V7, the UUID's generated later will always be bigger than the number generated before this.

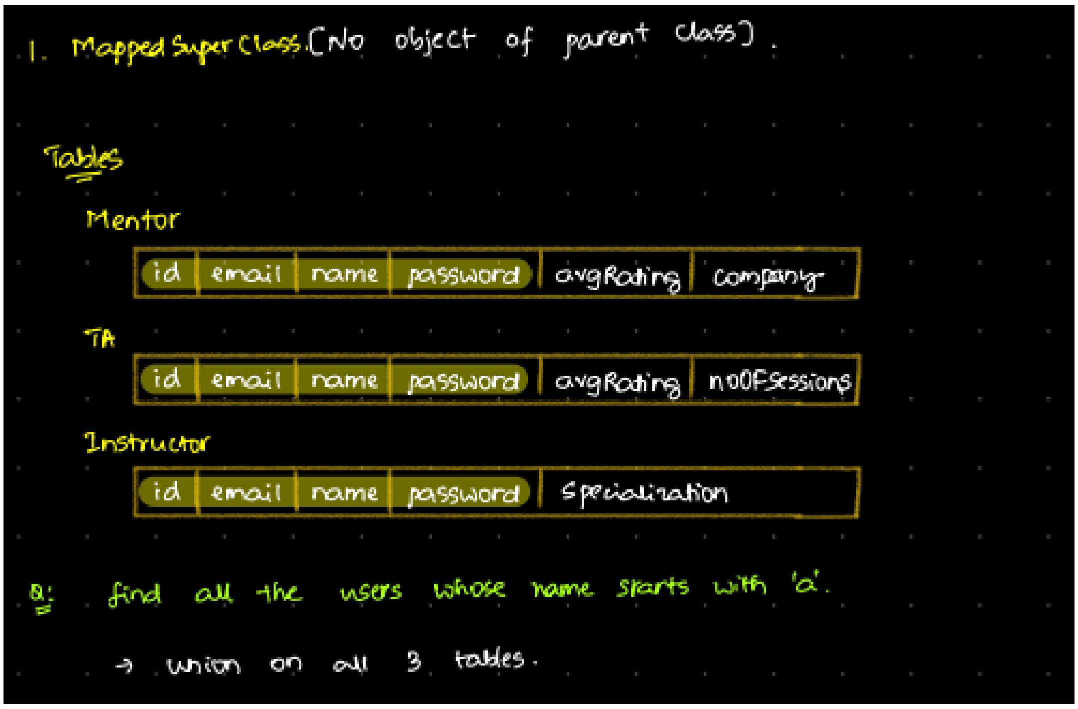
⇒ with this, there's a chance of restructuring entire tree, & comparision of email ids are needed.

As described above, after UUID V7, all numbers generated will be larger than those generated before. This is because to avoid rearranging IDs in the database, the ID column needs to be modified. However, this can pose issues if indexing is already in place.

Ways to represent inheritance

1. **@MappedSuperclass**: This annotation is used to designate a superclass whose mappings are applied to the entities that inherit from it. It allows you to define common fields and mappings in a superclass and have them inherited by multiple entity classes. it is used when we do not want to create super class object


How to represent these classes in DB?..



1. Mapped Super Class (No object of parent class).

Tables

Mentor

id	email	name	password	avgRating	company

TA

id	email	name	password	avgRating	noOfSessions

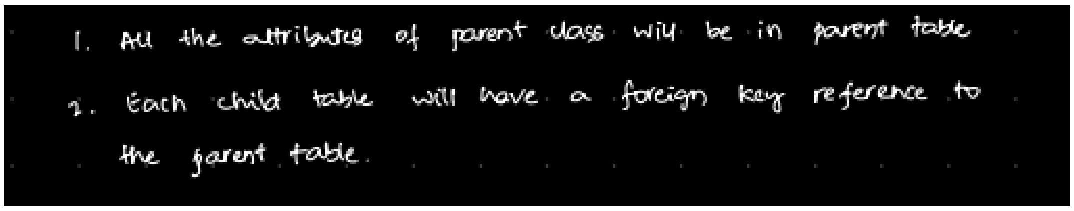
Instructor

id	email	name	password	specialization

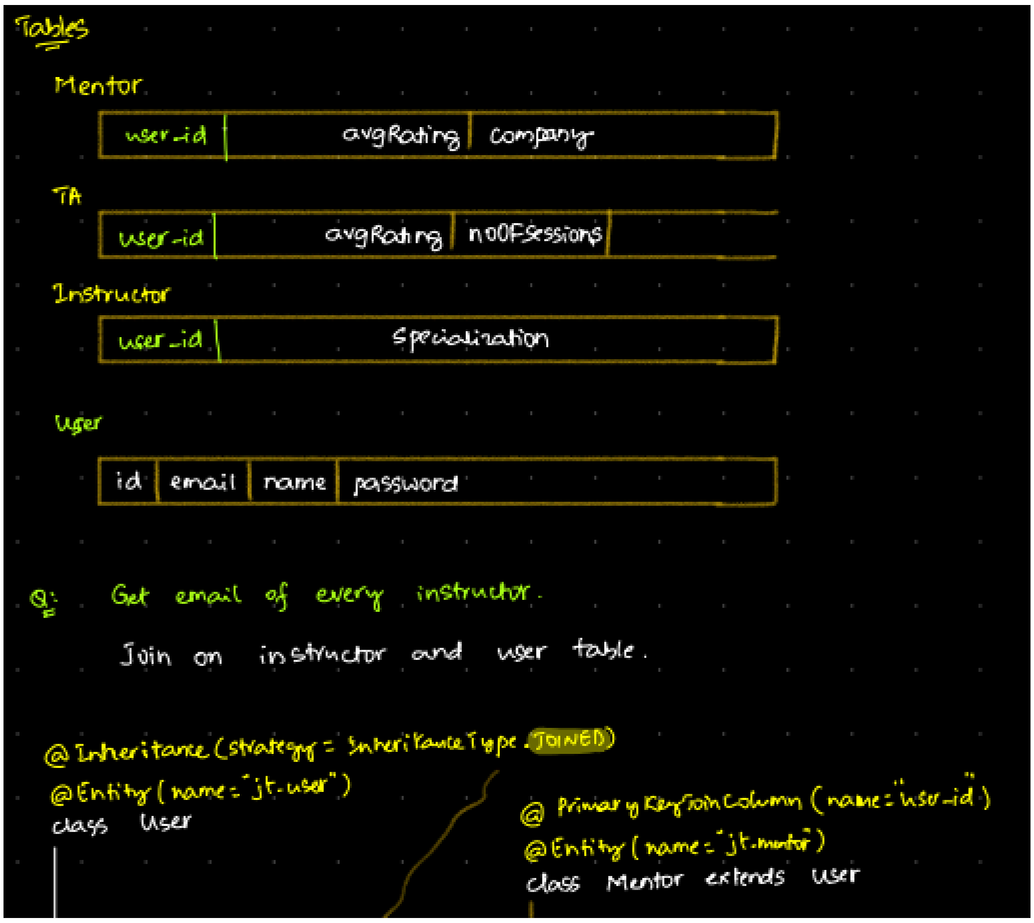
Q: find all the users whose name starts with 'a'.

→ union on all 3 tables.

2. **@JoinTable**: This annotation is used to specify the join columns and the name of the table when there is a Many-to-Many relationship between two entities. It helps in customizing the join table and its columns.



1. All the attributes of parent class will be in parent table
2. Each child table will have a foreign key reference to the parent table.



Tables

Mentor

user_id	avgRating	company

TA

user_id	avgRating	noOfSessions

Instructor

user_id	specialization

User

id	email	name	password

Q: Get email of every instructor.

Join on instructor and user table.

```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity(name="jt_user")
class User

@PrimaryKeyJoinColumn(name="user_id")
@Entity(name="jt_mentor")
class Mentor extends User
```

In the scenario described above, columns are not duplicated in each table; instead, they remain in the superclass table. To achieve this, we use the annotation **@Inheritance(strategy=InheritanceType.JOINED)** at the superclass level. and @Entity(name=”sc-user”) if we are using a reserved keyword like user class or anyothor Additionally, in the subclasses, we utilize the annotation above the subClass **@PrimaryKeyJoinColumn(name="user_id")**. We can name it anything **user_id** is the name of foreignkey here.

3. **@TablePerClass**: This annotation is used to implement inheritance mapping strategy where each subclass has its own table in the database. In this strategy, the superclass fields are duplicated in each subclass table and each record is in a separate table like Mentor, TA, Instructor will not be present in user table.

   "Table per Concrete Class" inheritance strategy in JPA is **@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS**). This annotation is typically applied at the superclass level to specify that each concrete subclass should have its own table in the database. Here we have table for user class also.

   Note: We can not use **@GeneratedValue(strategy = GenerationType.IDENTITY)** with Table_PER_CLASS. we have to chane it **@GeneratedValue(strategy = GenerationType.AUTO)**

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

@Entity(name="jt_user")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   public Long id;
   public String name;
   public String email;
   public  String password;
}
```

```
@Entity
@Getter
@Setter

public class Ta extends User {
   private int noOfSession;
   private int  avgRating;
}
```

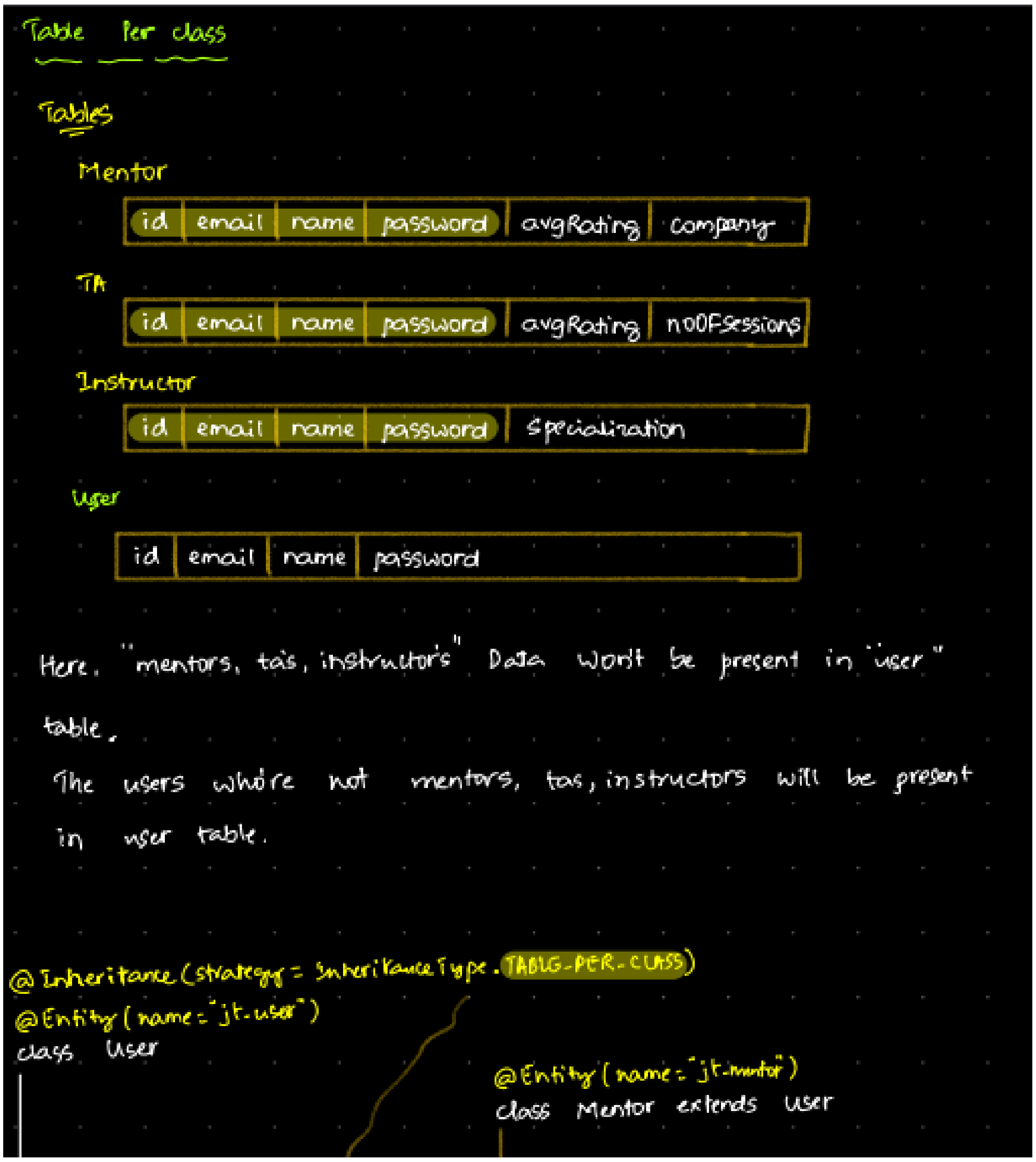


Table Per class

Tables

Mentor

id	email	name	password	avgRating	company

TA

id	email	name	password	avgRating	noOFSessions

Instructor

id	email	name	password	specialization

user

id	email	name	password

Here, "mentors, tas, instructor's" Data won't be present in "user" table.

The users who're not mentors, tas, instructors will be present in user table.

```
@Inheritance(strategy = InheritanceType.TABLE-PER-CLASS)
@Entity(name="jt-user")
class User

@Entity(name="jt-mentor")
class Mentor extends User
```

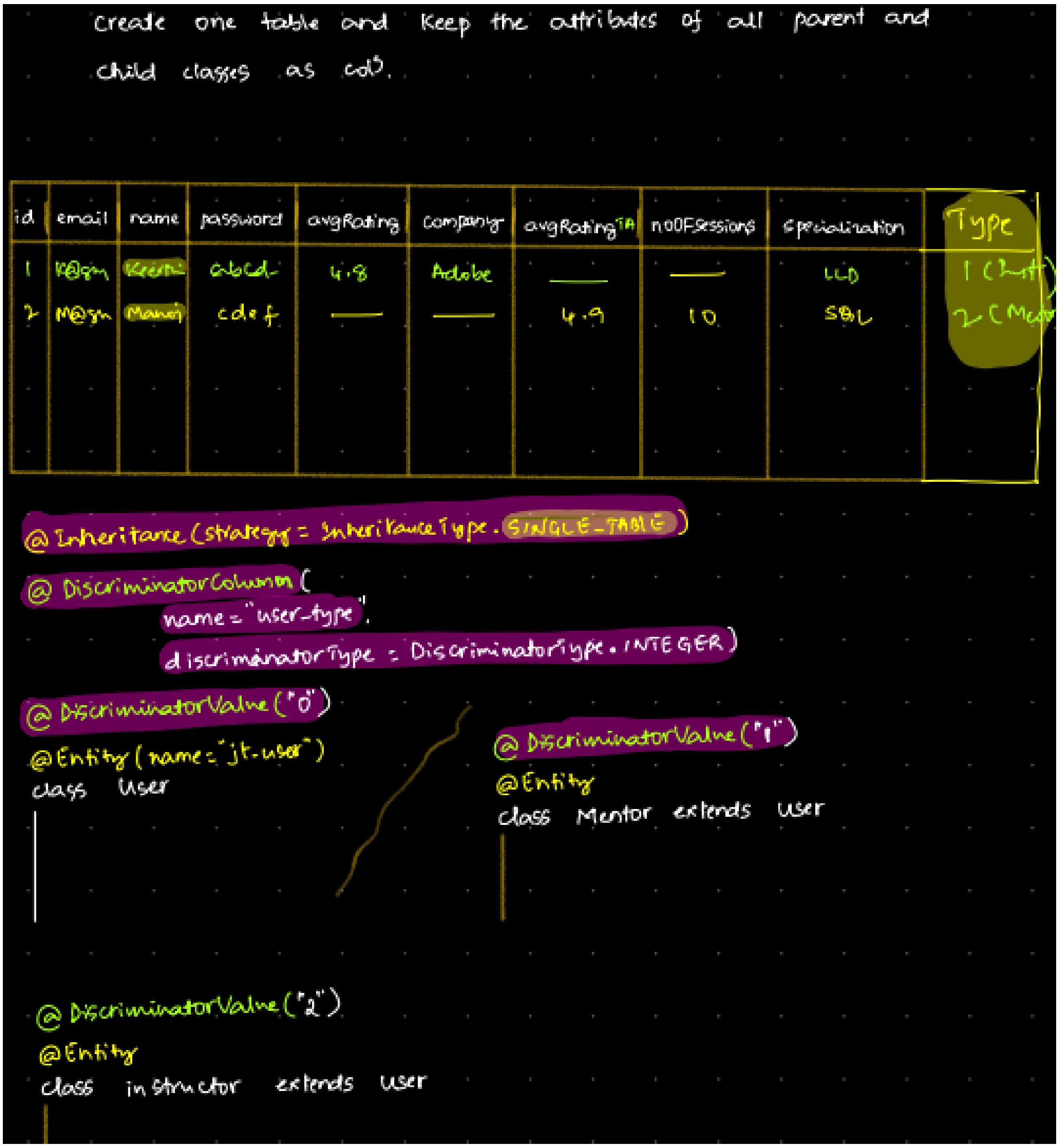
4. **@SingleTable**: This annotation is used to implement inheritance mapping strategy where all classes in the inheritance hierarchy are mapped to a single table in the database. Discriminator column is used to differentiate between the types of entities stored in the table.

   With the single table strategy, all types of objects in the inheritance hierarchy are stored in one big database table. This makes it really fast to run queries that involve different types of objects.

   But there's a downside: all the attributes of every object type have to fit into the same table. So, if some attributes only belong to certain types of objects, the rest of the table cells for other types are left empty. This can cause problems with data integrity because you can't enforce "must have a value" rules on attributes that only apply to some objects. And that could make your database administrator a bit nervous.

When you're defining the subclasses, make sure to add something called @DiscriminatorValue. This little addition tells the system how to tell the different subclasses apart when it's storing them in the database. It's like giving each subclass a special label.

Now, in some systems, like Hibernate, if you forget to add this label, it will use the name of the subclass as a default label. But, if you want your code to work smoothly across different systems, it's better to be explicit and provide the label yourself.



Create one table and Keep the attributes of all parent and child classes as col's.

id	email	name	password	avgRating	company	avgRatingTA	noOFSessions	specialization	Type
1	K@gm	Keerthi	abcd	4.8	Adobe	—	—	LLD	1 (Inst)
2	M@gm	Manoj	cdef	—	—	4.9	10	SBL	2 (Me

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
        name = "user_type",
        discriminatorType = DiscriminatorType.INTEGER)
@DiscriminatorValue("0")
@Entity(name = "jt_user")
class User

@DiscriminatorValue("1")
@Entity
class Mentor extends User

@DiscriminatorValue("2")
@Entity
class instructor extends User
```

```java
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "user_Type")
@Entity(name="jt_user")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   public Long id;
   public String name;
   public String email;
   public  String password;
}
```

```java
@Entity
@Getter
@Setter
@DiscriminatorValue("1")

public class Ta extends User {
   private int noOfSession;
   private int  avgRating;
}
```

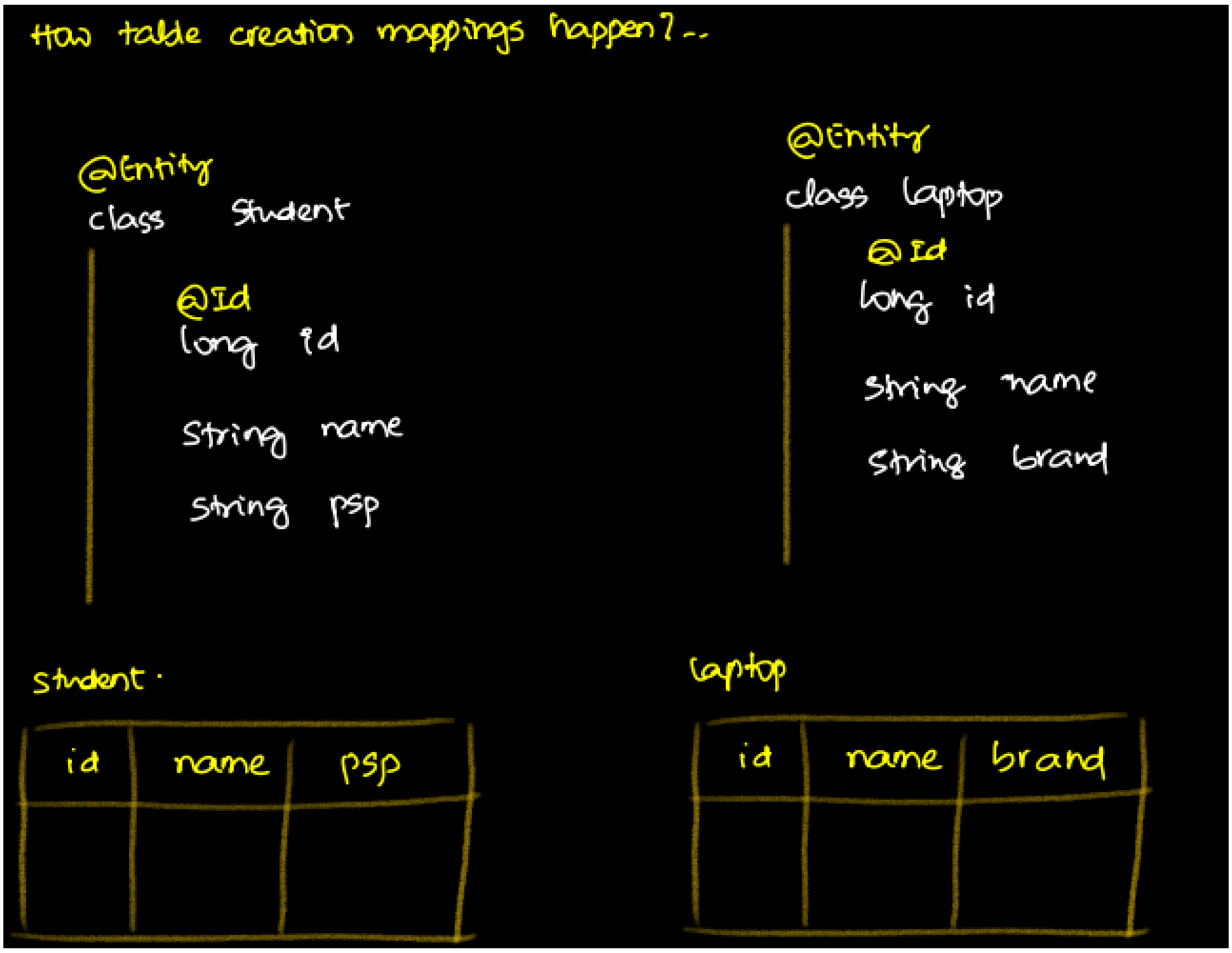
These annotations help in defining the structure of database tables and the relationships between entities in a Spring Boot application.

Backend LLD-4: JPA Queries, Cardinality Mappings - 13th April

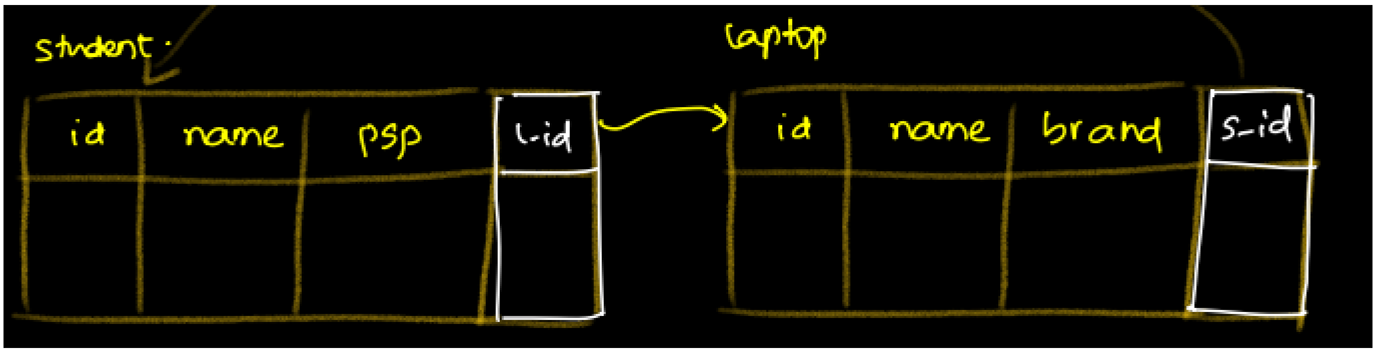
- **How table creation happens ?**
- **Types of quries (Inside repository)**

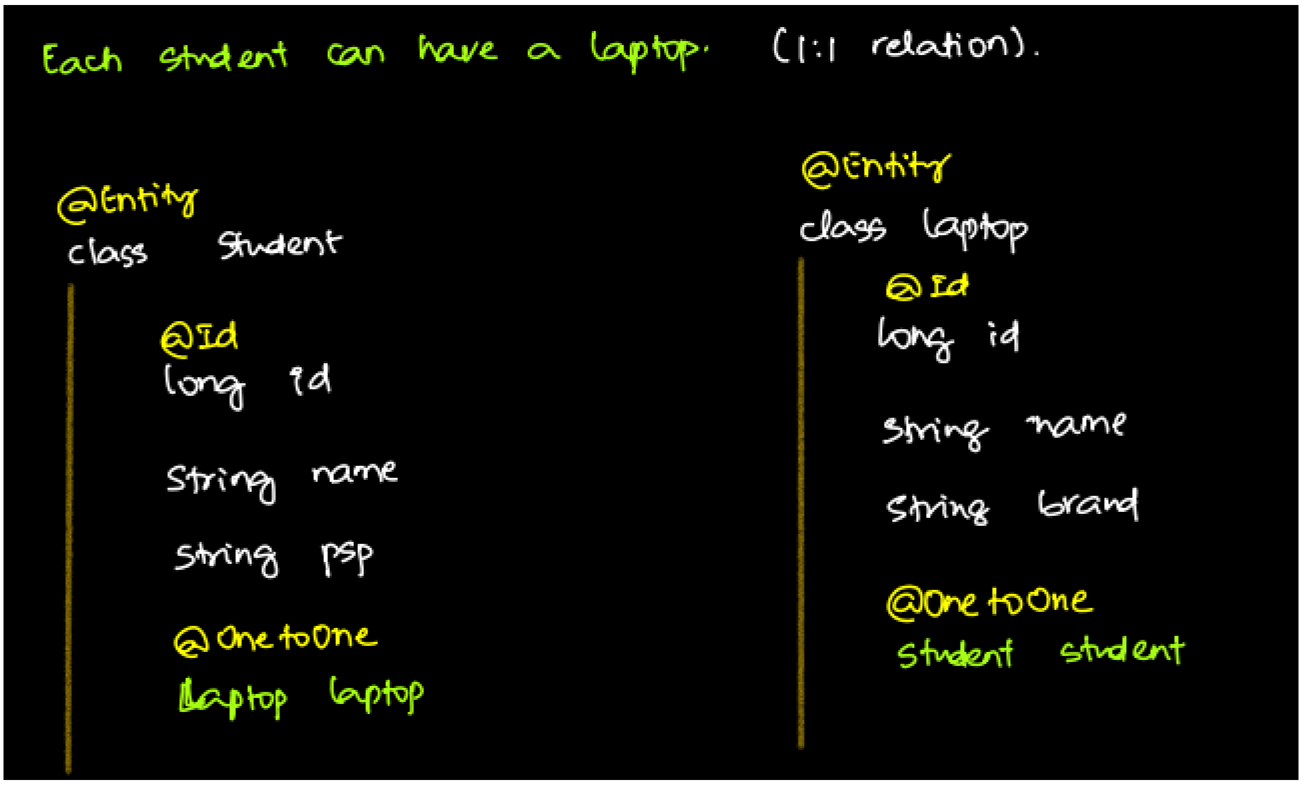
Basic mapping of table

When we want to connect a class to a table in a database, we use something called the @Entity annotation. This tells a tool called Hibernate how to link them together. It creates a table with the same name as the class, unless we specify a different name in the @Entity annotation.

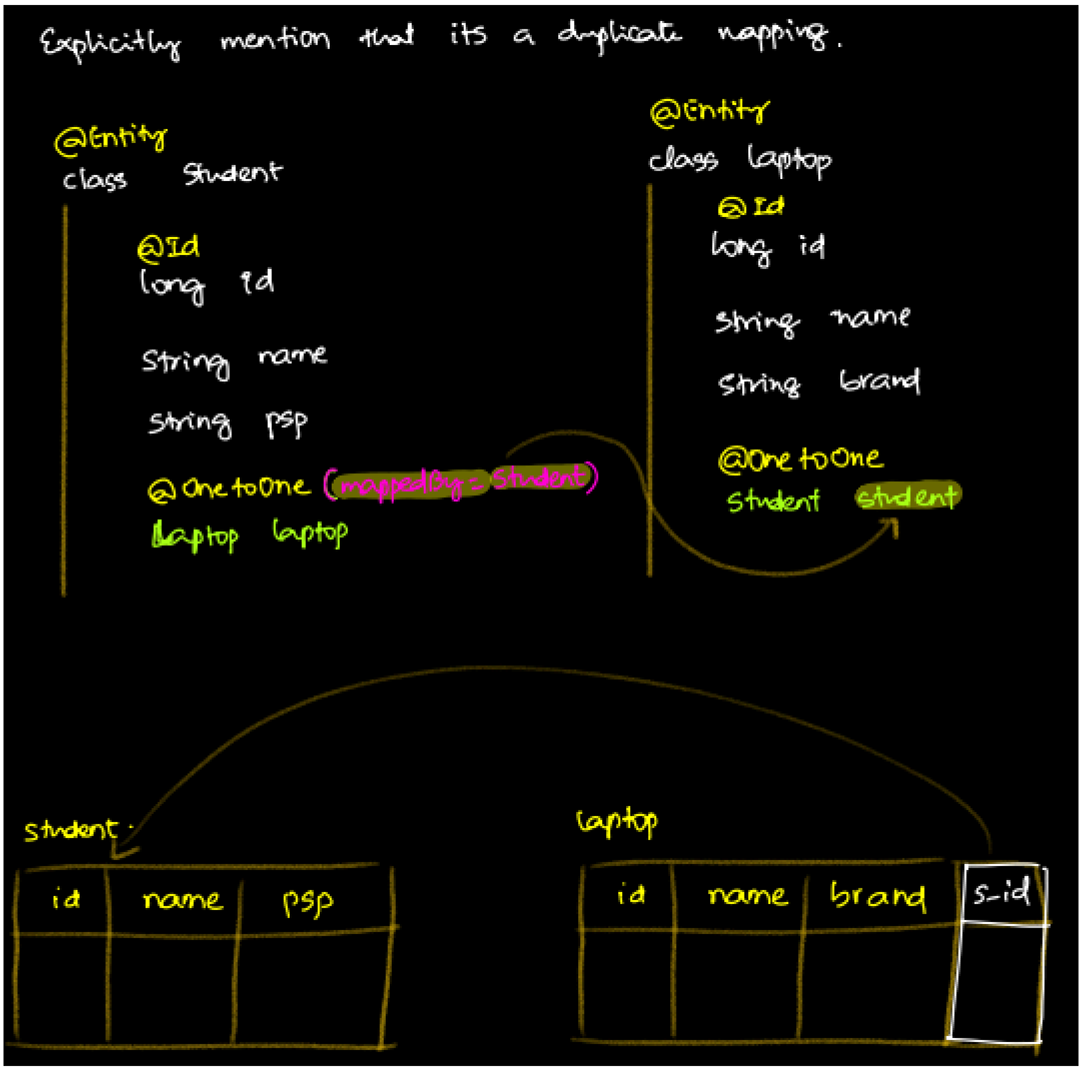


If we're using one class to refer to another in our code, we need to tell Hibernate about it using special codes like @OneToOne or @OneToMany. For example, if both a Student and a Laptop have a special connection called OneToOne, Hibernate will create a unique identification number (id) for each of them.



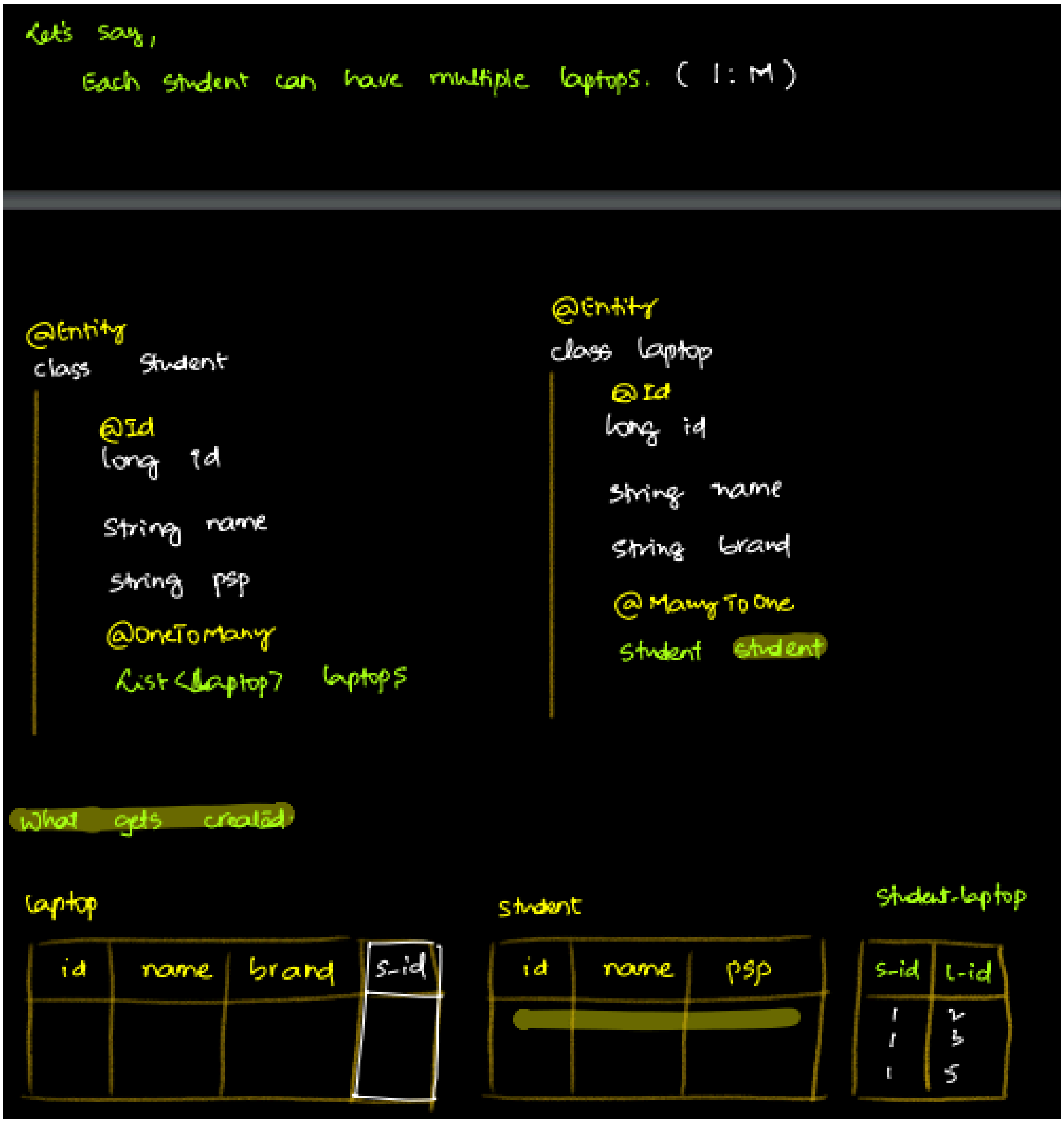


To avoid this We just need to say which class is not required to create column using by using @OneToMany(mappedBy=Student), and Hibernate will handle the rest.



If each student can own several laptops, we'll make a list of laptops within the student class. This sets up a "one-to-many" relationship, meaning one student can have many laptops, but each laptop can only belong to one student.

When we use @OneToMany in the student class and @ManyToOne in the laptop class, Hibernate adds a new column called "s_id" in the laptop table. It also creates a new table called student_laptop to keep track of this relationship because we are storing lis of laptop in student class and we can't store a list of IDs directly in the database.



Let's say,

Each student can have multiple laptops. (1 : M)

```
@Entity
class Student
    @Id
    long id

    String name

    String psp

    @OneToMany
    List<Laptop> laptops
```

```
@Entity
class Laptop
    @Id
    long id

    String name

    String brand

    @ManyToOne
    Student student
```

What gets created

laptop

id	name	brand	s_id

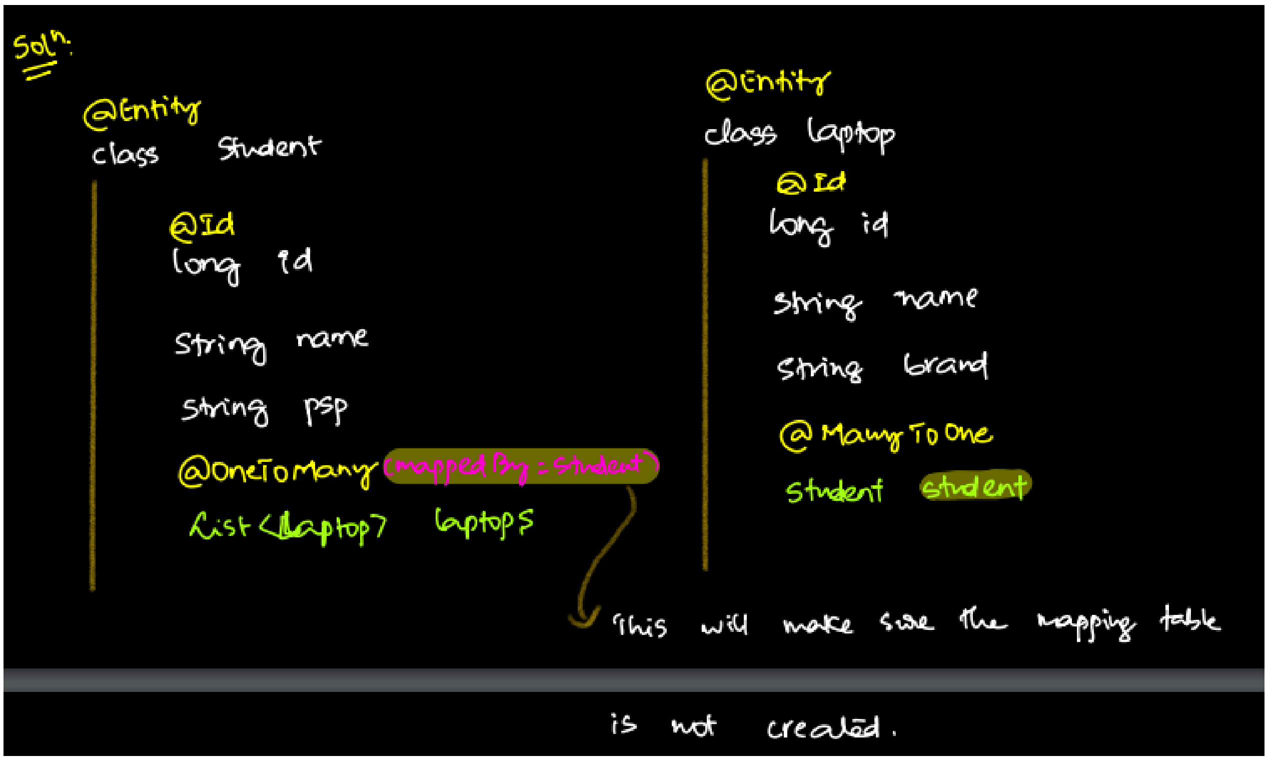
student

id	name	psp

Student_laptop

s_id	l_id
1	2
1	3
1	5

s_id is required but student_laptop table is extra for us to remove this we just simply add @OneToMany(mappedBy=”student”) n student clss just above the list.

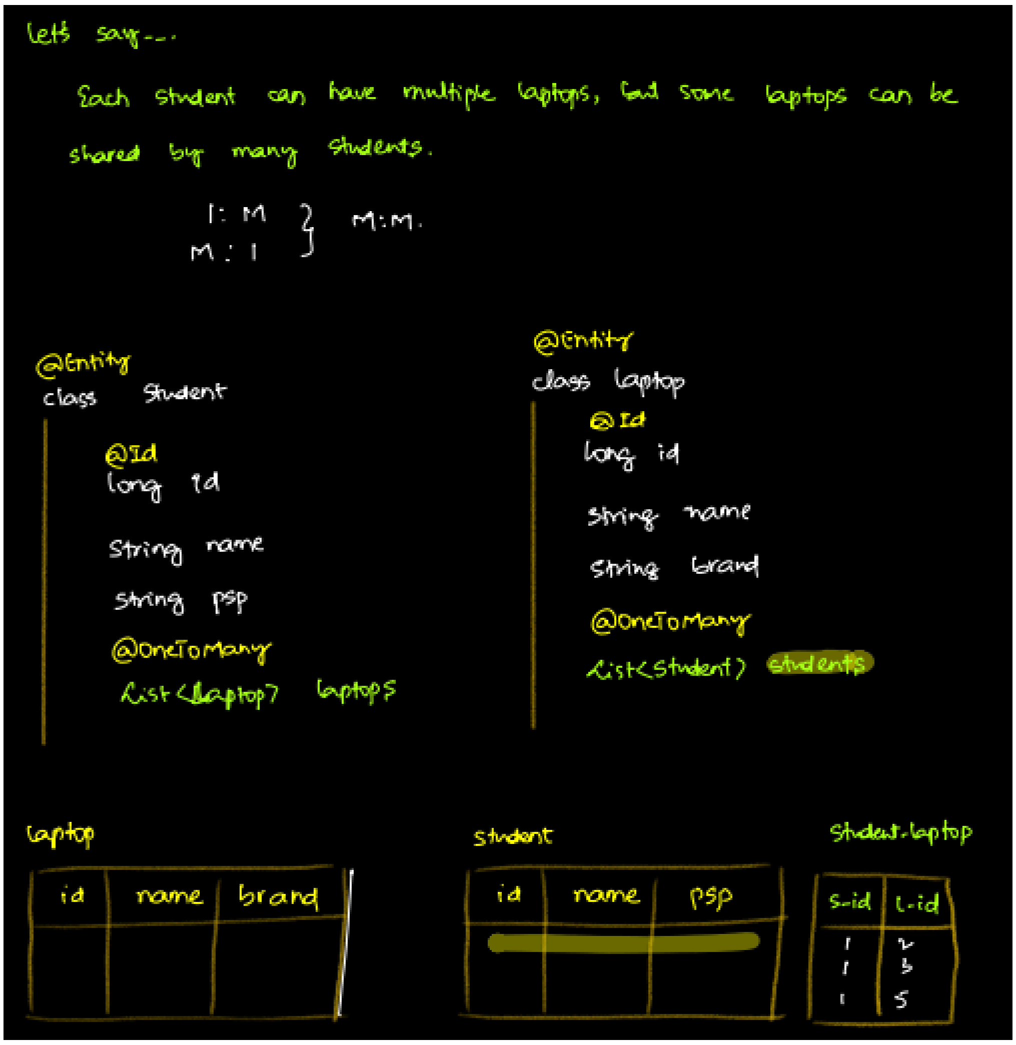


Soln:

```
@Entity
class Student
    @Id
    long id

    String name

    String psp

    @OneToMany(mappedBy = student)
    List<Laptop> laptops
```

```
@Entity
class laptop
    @Id
    long id

    String name

    String brand

    @ManyToOne
    Student student
```

This will make sure the mapping table is not created.



Let's say...

Each student can have multiple laptops, and some laptops can be shared by many students.

```
 1 : M  }
 M : 1  }  M:M.
```

```
@Entity
class Student
    @Id
    long id

    String name

    String psp

    @OneToMany
    List<Laptop> laptops
```

```
@Entity
class laptop
    @Id
    long id

    String name

    String brand

    @OneToMany
    List<Student> students
```

laptop

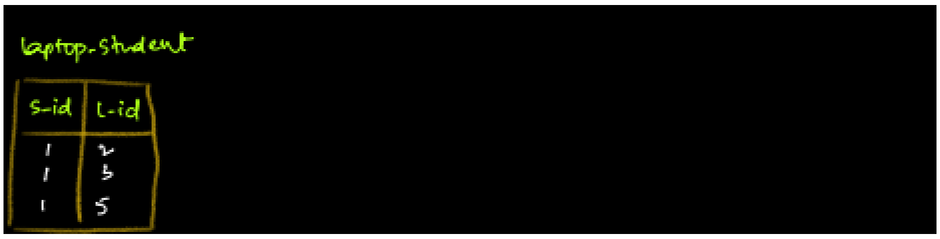
id	name	brand

student

id	name	psp

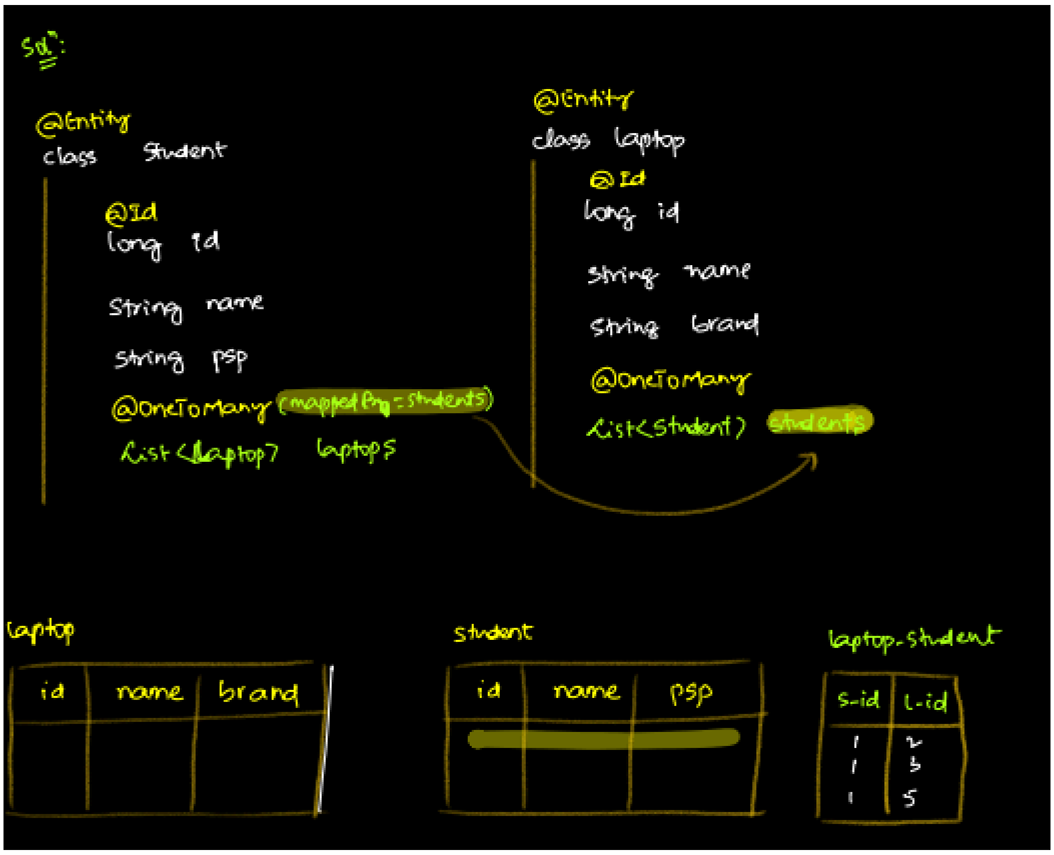
student_laptop

s-id	l-id
1	2
1	3
1	5



laptop_student

s-id	l-id
1	2
1	3
1	5


Sql:

```
@Entity
class   Student

    @Id
    long  id

    String  name

    String  psp

    @OneToMany (mappedBy = students)
    List<Laptop>   laptops
```

```
@Entity
class  laptop
    @Id
    long  id

    String  name

    String  brand

    @OneToMany
    List<Student>  students
```

laptop

id	name	brand

student

id	name	psp

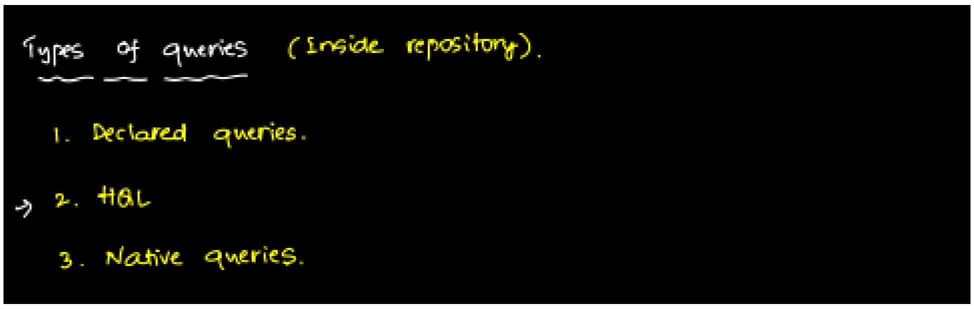
laptop-student

s-id	l-id
1	2
1	3
1	5

==

Types of quries (Inside repository)

==


Types of queries (Inside repository).

1. Declared queries.
2. HQL
3. Native queries.

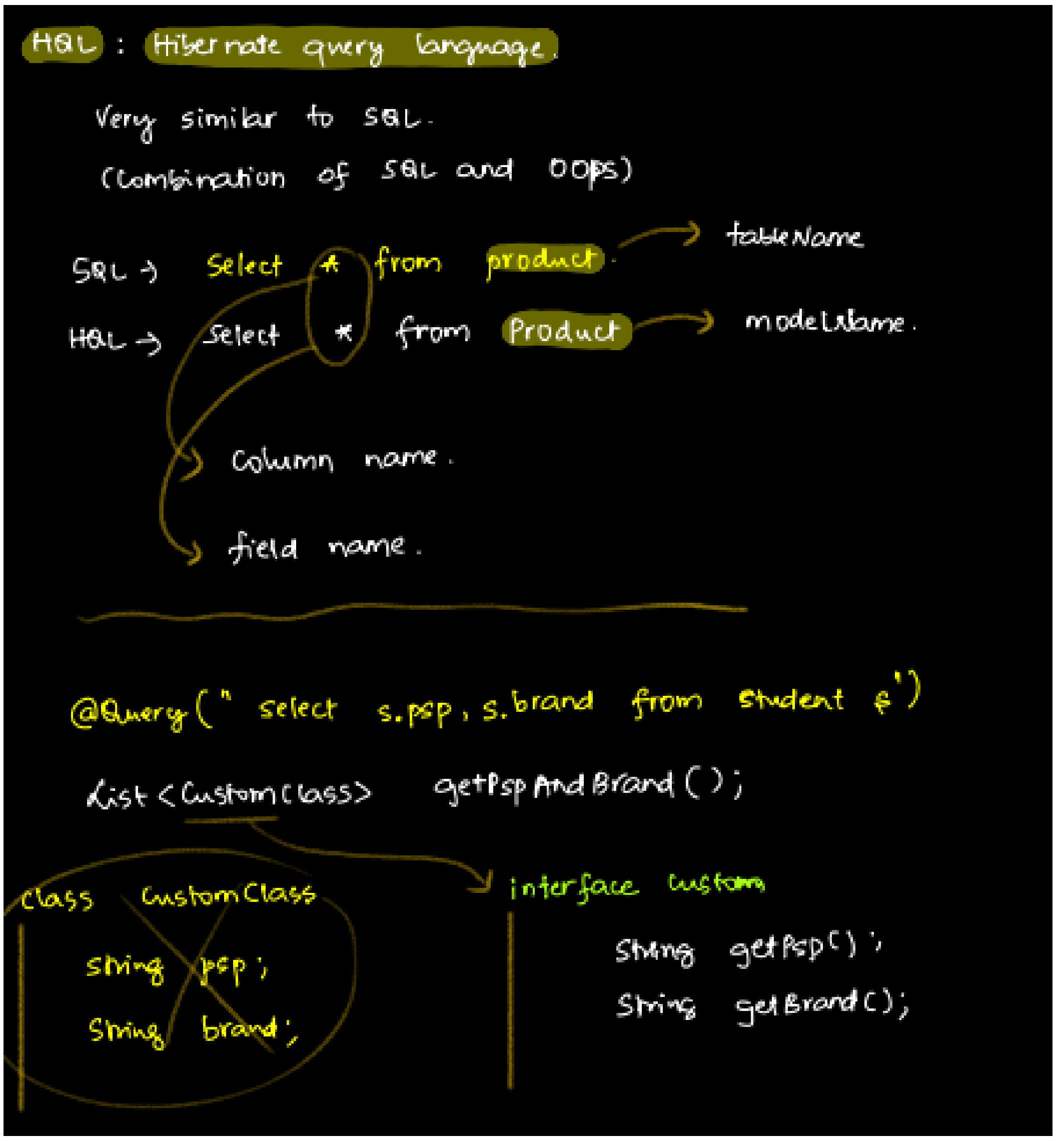
Basic Queries (Declared quries): Simple queries based on method names derived from the method signature. For example, findByFirstName(String firstName).

HQL: Hibernate Query Language (HQL) is a powerful object-oriented query language used in Hibernate, which is an ORM (Object-Relational Mapping) framework for Java. In Spring Boot applications, you can use HQL for more complex data retrieval operations when working with JPA repositories.

HQL is like SQL but uses class names instead of table names and field names instead of column names. It's useful for writing complex queries and works independently of the database. If we need data from multiple classes, we can create a custom class, instantiate it, and call it in HQL.

HQL is similar to SQL, but instead of operating on tables and columns, it operates on persistent objects and their properties. Some key points about HQL:

1. **Object-Oriented:** HQL queries are written using the entity and property names rather than table and column names.
2. **Supports Joins:** HQL supports various types of joins including inner joins, outer joins, and cross joins.
3. **Named Parameters:** HQL supports named parameters, which provide better readability and flexibility in parameter binding.
4. **Entity Relationships:** HQL allows you to navigate through entity relationships (e.g., associations) in your queries.
5. **Aggregation Functions:** HQL supports aggregation functions like SUM, COUNT, AVG, etc.

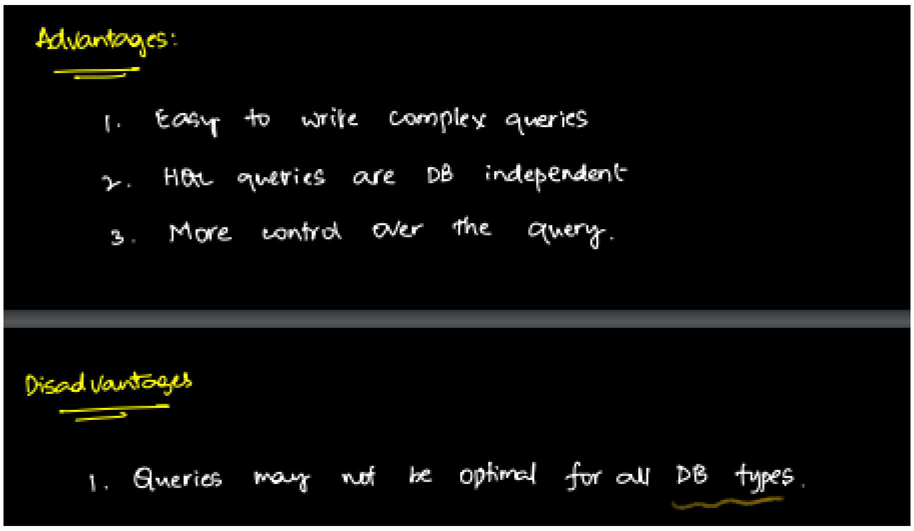

HQL : Hibernate query language
Very similar to SQL.
(Combination of SQL and OOPs)
SQL → Select * from product → tableName
HQL → Select * from Product → modelName.
Column name.
field name.

```
@Query(" select s.psp, s.brand from student s")
List<CustomClass> getPspAndBrand();

class CustomClass
    String psp;
    String brand;

interface custom
    String getPsp();
    String getBrand();
```

Using interfaces in HQL allows for polymorphic queries, where you can retrieve entities based on their common interface rather than their concrete class.

Who worked on the ORM (Object-Relational Mapping) made a minor adjustment to it. They requested interfaces instead of classes because if you use a class, it can potentially be altered using getter and setter methods. When retrieving data from the database, we typically don't expect to change it. To ensure this, they suggest passing interfaces and using abstract methods instead of class fields also keep naming convention like for psp write **String getPsp()**. This approach helps maintain data integrity and security.

Note: If we're writing a class and we need some fields or required fields from another table, it's advisable to create a new interface in the DTO (Data Transfer Object) package. Then, we can use that interface as the return type in the repository. This approach helps maintain separation of concerns and ensures clarity in data access operations

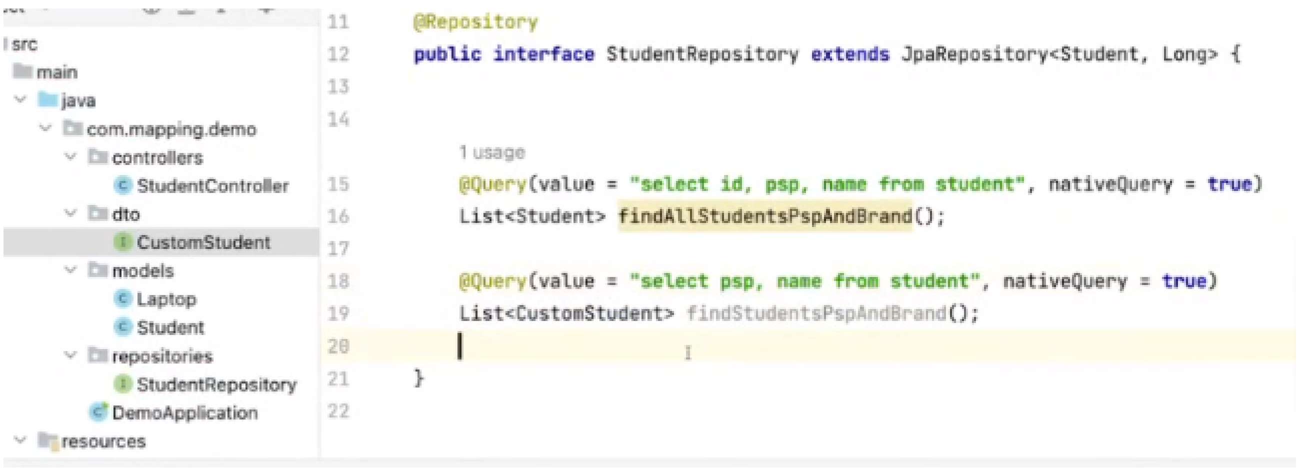


Advantages:

1. Easy to write complex queries
2. HQL queries are DB independent
3. More control over the query.

Disadvantages

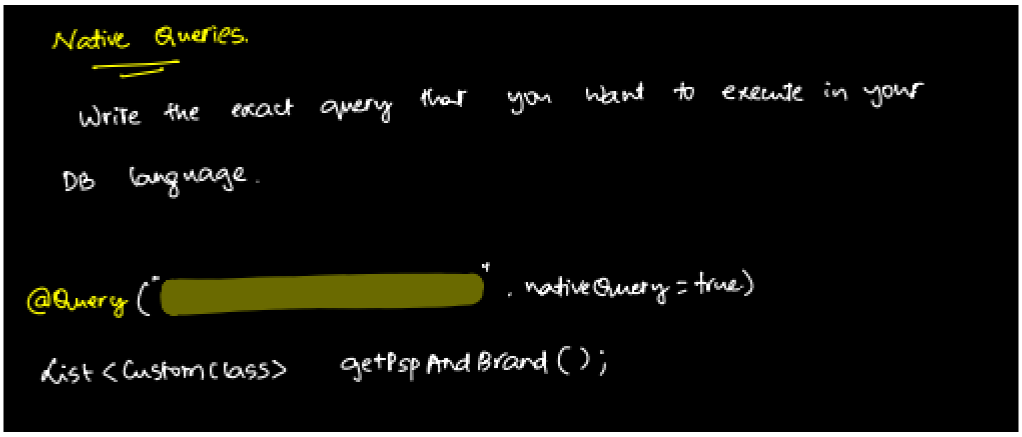
1. Queries may not be optimal for all DB types.



```
src
  main
    java
      com.mapping.demo
        controllers
          StudentController
        dto
          CustomStudent
        models
          Laptop
          Student
        repositories
          StudentRepository
        DemoApplication
    resources
```

```java
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {


    1 usage
    @Query(value = "select id, psp, name from student", nativeQuery = true)
    List<Student> findAllStudentsPspAndBrand();

    @Query(value = "select psp, name from student", nativeQuery = true)
    List<CustomStudent> findStudentsPspAndBrand();

}
```

Native Queries: SQL queries written directly in your repository interface using the @Query annotation with the nativeQuery attribute set to true.



Native Queries.

Write the exact query that you want to execute in your DB language.

```
@Query("[redacted]", nativeQuery = true)
List<CustomClass>    getPspAndBrand();
```

The problem with native query is it is Database dependent and we have to write different query for different database type like mysql and oracle. to mange this we will write different files for every database and as per requirement we will call it.

Backend LLD-4: Query Methods, Fetch Type, Mode, Schema Versioning - April 17

- **Fetch Types**
- **Mode**
- **Schema Vrsioning**

Fetch Types -> In Hibernate, the default fetch type for associations is also **FetchType.LAZY**, similar to JPA. This means that by default, associated entities are loaded lazily when they are accessed for the first time.

However, you can also specify FetchType.EAGER to override the default behavior and eagerly fetch associated entities every time the parent entity is loaded.

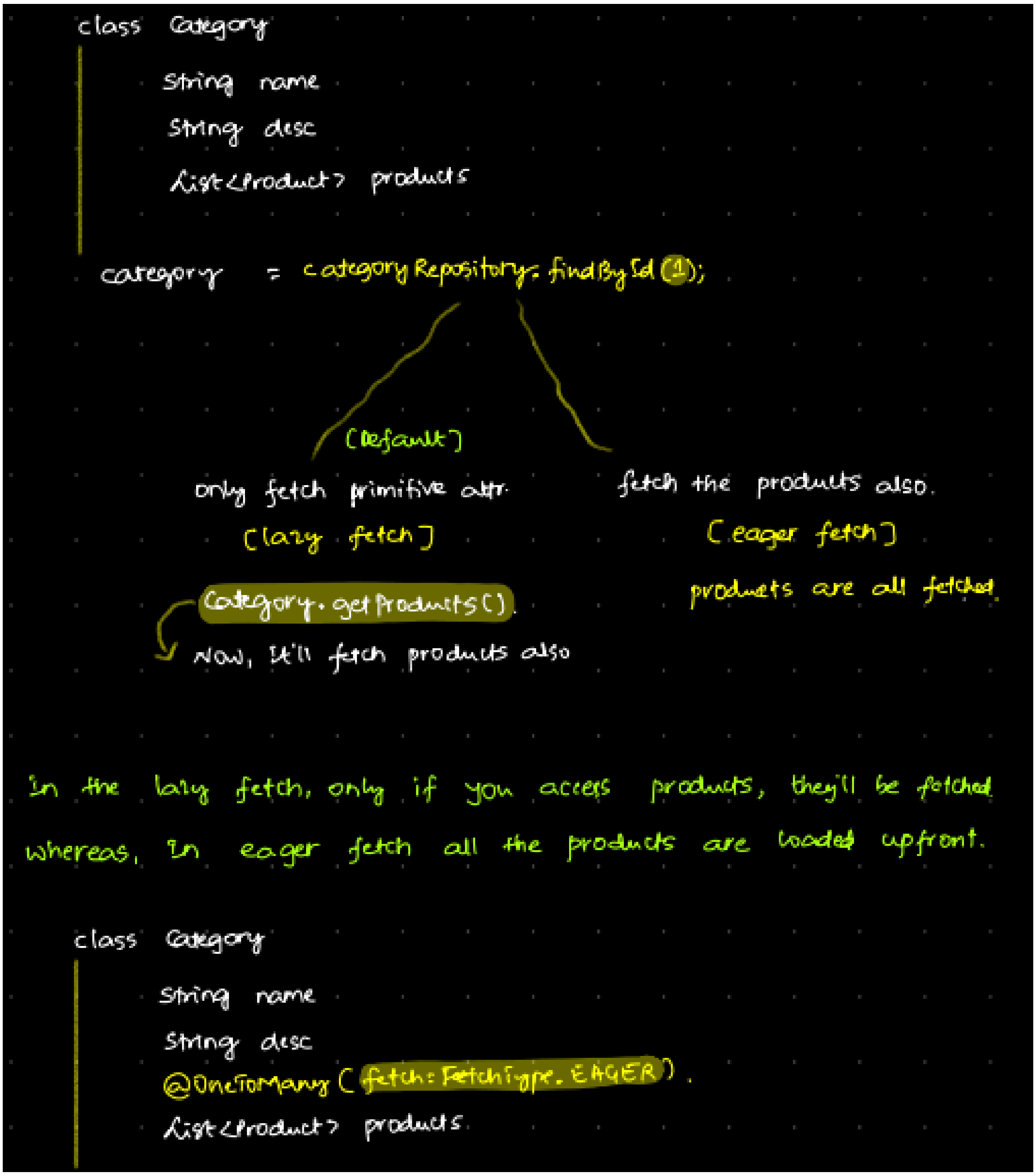
Here's an example of how you can specify fetch types in Hibernate using annotations:

```
@Entity
public class Category {
   // Other fields and annotations

   @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
   private List<Product> products;

   // Getters and setters
}

@Entity
public class Product {
   // Other fields and annotations

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "category_id")
   private Category category;

   // Getters and setters
}
```

In this example, the **@OneToMany** association from Category to Product specifies FetchType.LAZY, meaning that when you fetch a Category entity, the associated Product entities will not be loaded automatically.

Similarly, the **@ManyToOne** association from Product to Category also specifies **FetchType.LAZY**, ensuring that when you fetch a Product entity, the associated Category entity will not be loaded automatically.

You can adjust the fetch type according to your application's requirements to optimize performance and avoid unnecessary data loading.



```
class Category
    String name
    String desc
    List<Product> products

category = categoryRepository.findById(1);
```

(default)
only fetch primitive attr.
[lazy fetch]

Category.getProducts()
Now, it'll fetch products also

fetch the products also.
[eager fetch]
products are all fetched.

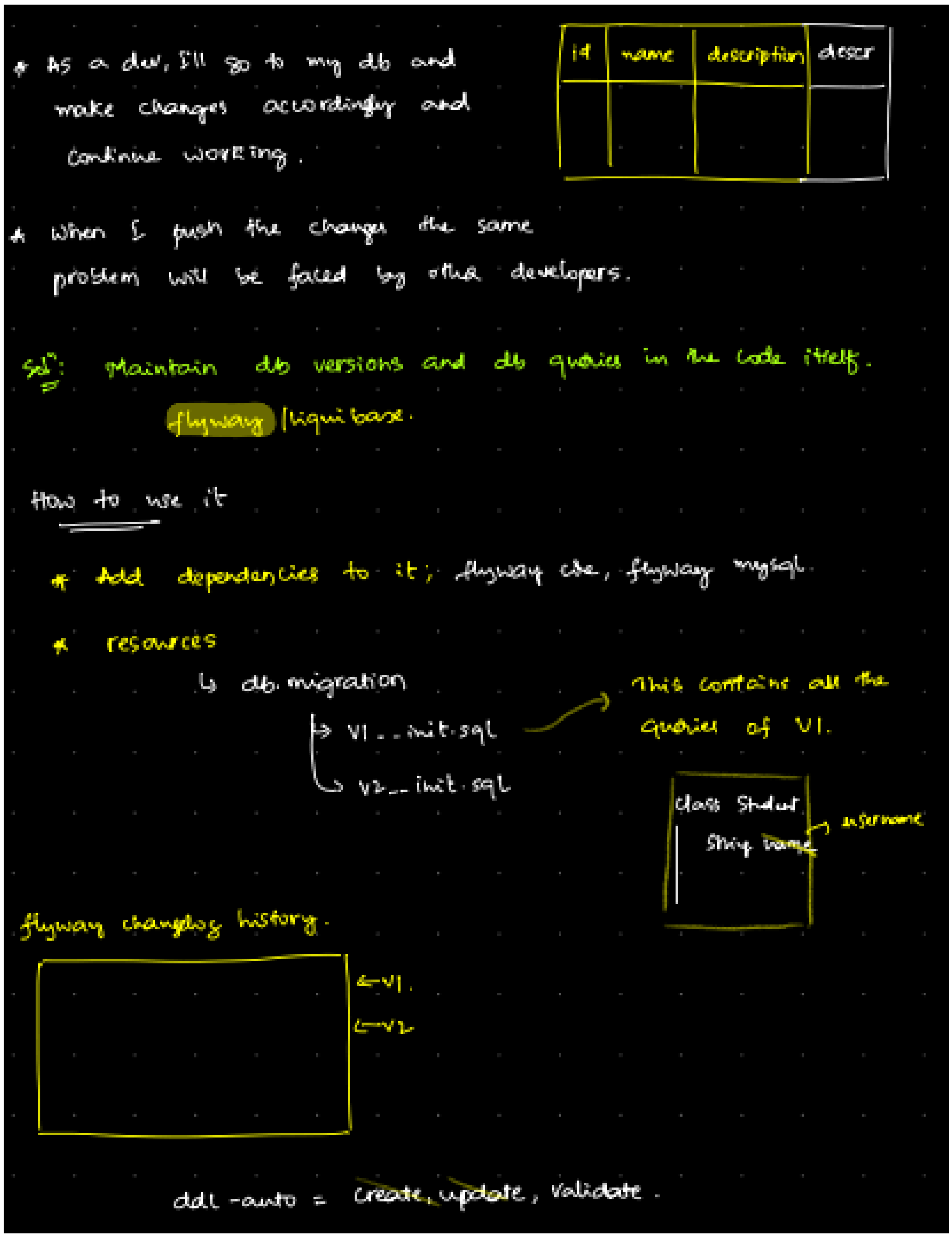
In the lazy fetch, only if you access products, they'll be fetched whereas, in eager fetch all the products are loaded upfront.

```
class Category
    String name
    String desc
    @OneToMany(fetch=FetchType.EAGER)
    List<Product> products
```

Schema Vrsioning -> Database versioning is necessary because when we make changes to our code, like renaming a property in a class, the database might not automatically update to match these changes. For example, if we change "description" to "descr" in our code, Hibernate won't know about this change and might add a new column to the table instead of changing the existing one.

If we manually change the database and then share our code with other developers, they would have to make the same changes to their databases. This can cause inconsistency and errors. To solve this, we need a way to manage these database changes within our codebase.

That's where Flyway comes in. It helps us keep track of database changes over time. Every time we make a change to the database structure, we create a new database version file. Flyway keeps track of these versions and ensures that everyone's database stays consistent with the codebase. and we achieve this using **Flyway core** and **flyway mysql**



* As a dev, I'll go to my db and make changes accordingly and continue working.

* When I push the changes the same problem will be faced by other developers.

Sol: Maintain db versions and db queries in the code itself.

flyway / liquibase.

How to use it

* Add dependancies to it; flyway core, flyway mysql.

* resources
    ↳ db.migration
        ↳ V1__init.sql → This contains all the queries of V1.
        ↳ V2__init.sql

ddl-auto = ~~create~~, ~~update~~, validate.

Step 1: Add Flyway to Your Project
You can add the Flyway dependency to your project's build configuration file. For Maven, add the following to your pom.xml:

```
<!-- https://mvnrepository.com/artifact/org.flywaydb/flyway-core -->
<dependency>
   <groupId>org.flywaydb</groupId>
   <artifactId>flyway-core</artifactId>
   <version>10.11.0</version>
</dependency>
```

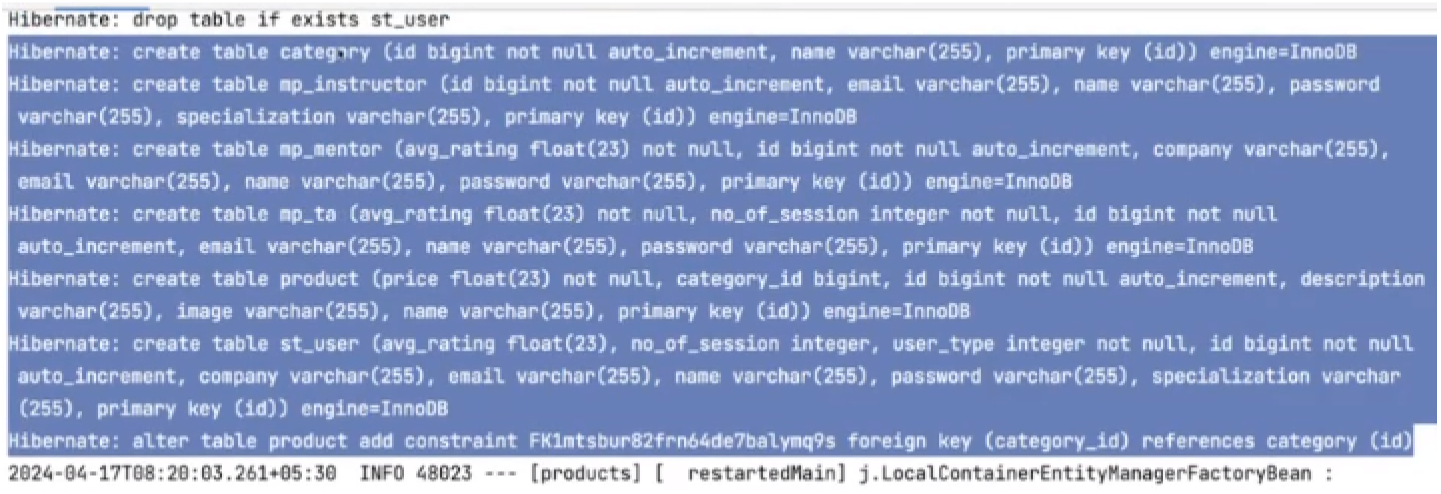
```
<!-- https://mvnrepository.com/artifact/org.flywaydb/flyway-mysql -->
<dependency>
<groupId>org.flywaydb</groupId>
<artifactId>flyway-mysql</artifactId>
<version>10.11.0</version>
</dependency>
```

Install dependency by click refresh button on right side

Step 2: Create Flyway Migration Scripts
Create SQL migration scripts following Flyway's naming convention. These scripts should be placed in the **db/migration directory** within your project's **resources** folder. Each script should have a name that starts with a version prefix followed by a description.

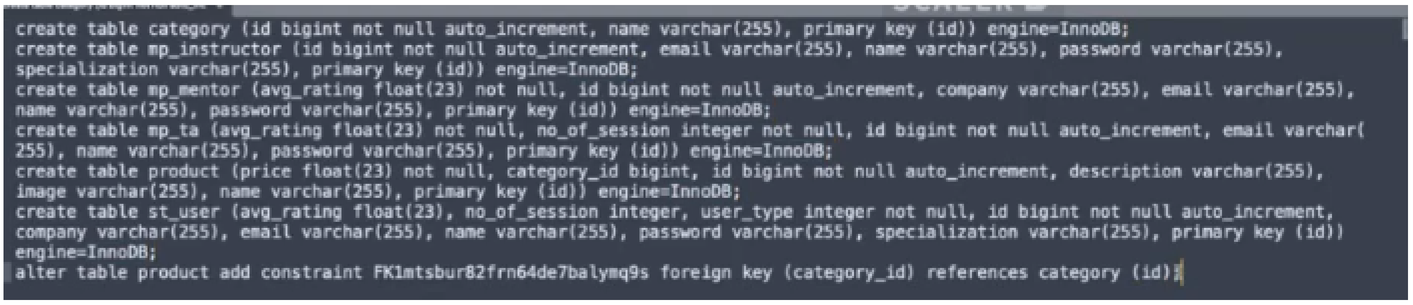
```
V1__Initial_schema.sql
V2__Add_table.sql
V3__Modify_table.sql
```

sql queries from console



```
Hibernate: drop table if exists st_user
Hibernate: create table category (id bigint not null auto_increment, name varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table mp_instructor (id bigint not null auto_increment, email varchar(255), name varchar(255), password
 varchar(255), specialization varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table mp_mentor (avg_rating float(23) not null, id bigint not null auto_increment, company varchar(255),
 email varchar(255), name varchar(255), password varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table mp_ta (avg_rating float(23) not null, no_of_session integer not null, id bigint not null
 auto_increment, email varchar(255), name varchar(255), password varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table product (price float(23) not null, category_id bigint, id bigint not null auto_increment, description
 varchar(255), image varchar(255), name varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table st_user (avg_rating float(23), no_of_session integer, user_type integer not null, id bigint not null
 auto_increment, company varchar(255), email varchar(255), name varchar(255), password varchar(255), specialization varchar
 (255), primary key (id)) engine=InnoDB
Hibernate: alter table product add constraint FK1mtsbur82frn64de7balymq9s foreign key (category_id) references category (id)
2024-04-17T08:20:03.261+05:30  INFO 48023 --- [products] [  restartedMain] j.LocalContainerEntityManagerFactoryBean :
```

intial queries after filter



```
create table category (id bigint not null auto_increment, name varchar(255), primary key (id)) engine=InnoDB;
create table mp_instructor (id bigint not null auto_increment, email varchar(255), name varchar(255), password varchar(255),
specialization varchar(255), primary key (id)) engine=InnoDB;
create table mp_mentor (avg_rating float(23) not null, id bigint not null auto_increment, company varchar(255), email varchar(255),
name varchar(255), password varchar(255), primary key (id)) engine=InnoDB;
create table mp_ta (avg_rating float(23) not null, no_of_session integer not null, id bigint not null auto_increment, email varchar(
255), name varchar(255), password varchar(255), primary key (id)) engine=InnoDB;
create table product (price float(23) not null, category_id bigint, id bigint not null auto_increment, description varchar(255),
image varchar(255), name varchar(255), primary key (id)) engine=InnoDB;
create table st_user (avg_rating float(23), no_of_session integer, user_type integer not null, id bigint not null auto_increment,
company varchar(255), email varchar(255), name varchar(255), password varchar(255), specialization varchar(255), primary key (id))
engine=InnoDB;
alter table product add constraint FK1mtsbur82frn64de7balymq9s foreign key (category_id) references category (id);
```

Copy this all query in v1 file at first time and in next time just write require query in next version like **ALTER TABLE table_name RENAME description To descr;**

Note -> To create the first SQL file, we usually need to recreate the entire database and manually copy all the SQL queries from the console into a file named "V1_initial_schema". However, if we want to automate this process, we can utilize the JPA Buddy plugin available in IntelliJ Ultimate version.

Step 3: Configure Flyway
In your Spring Boot application, configure Flyway by providing the necessary properties in your application.properties use

```
spring.jpa.hibernate.ddl-auto = validate
```

Step 4: Integrate with Hibernate
Hibernate will automatically detect the changes made to the database schema by Flyway during the application startup. Ensure that your Hibernate configuration is correctly set up to work with the database schema.

After run it first time it will create **flyway_schema_history**