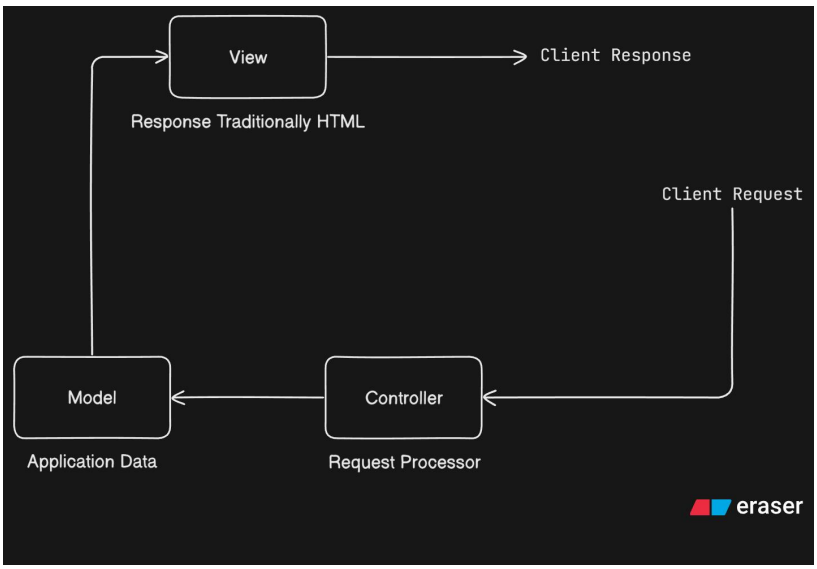
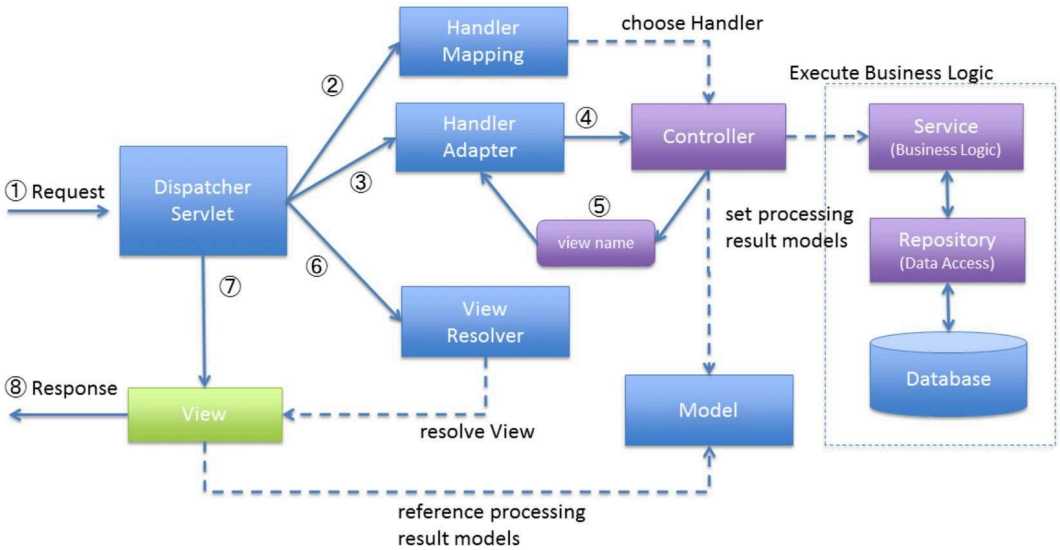
MVC Architecture



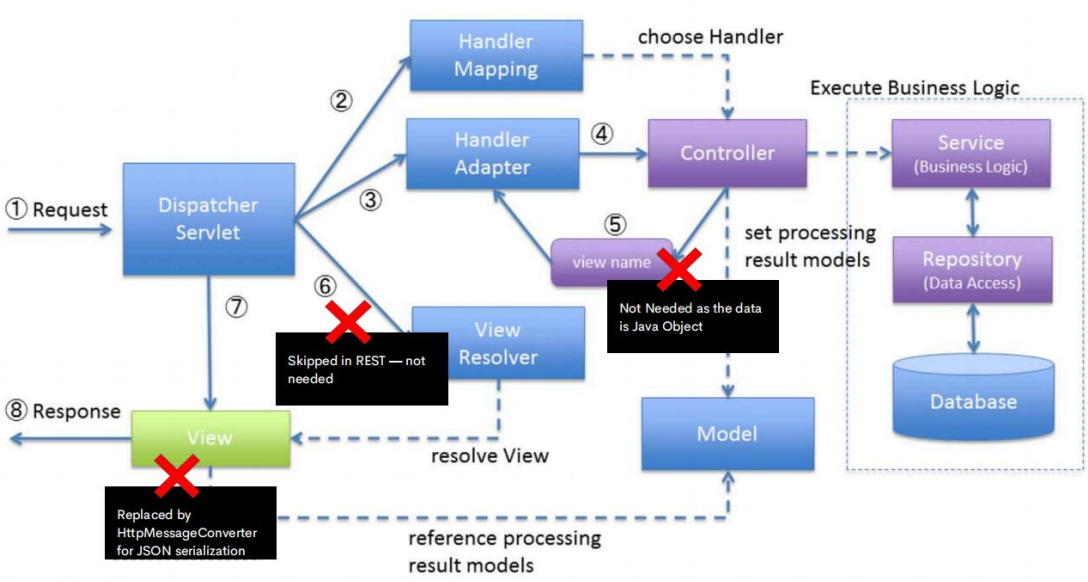
- Controller
  - Different APIs (Get, Post etc) will be having different controller
- Model
  - Handling the data
  - Talking to Database
  - In case of Java, Model is present in the Object
- View
  - HTML or JSON
  - Send the data to the client

- **Spring MVC** (Legacy)



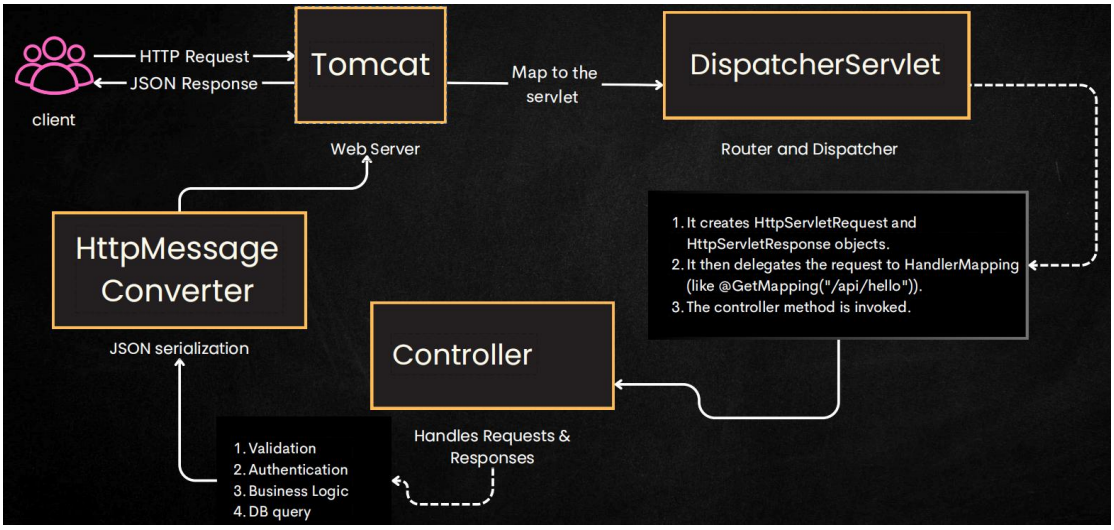
- View Resolver will take a look at the Dispatcher Servlet that which kind of user has requested and then it converts the Model to that format i.e. XML, HTML, JSON etc.

➢ **Spring MVC** (Modern)

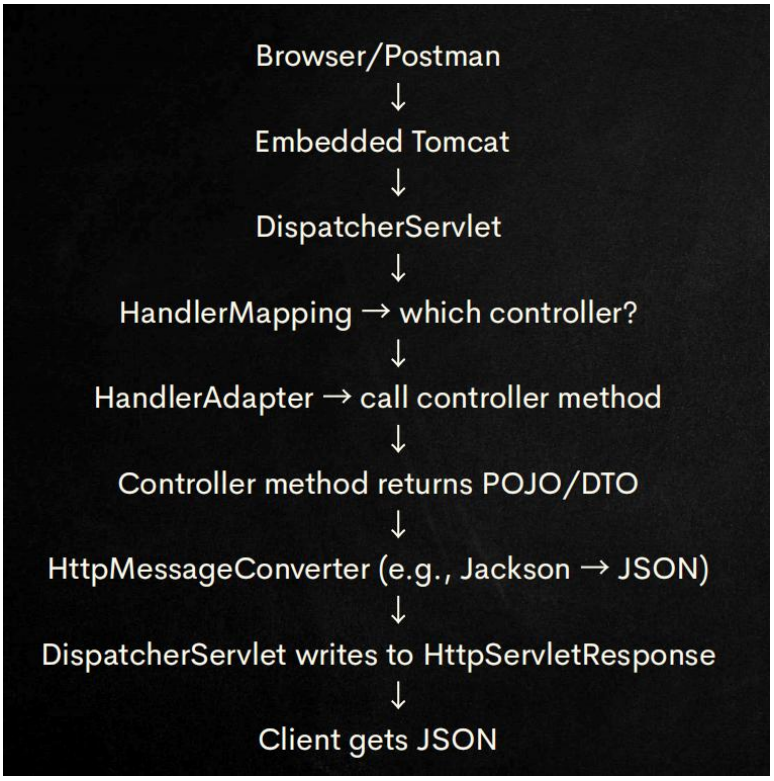


- In recent times, JSP is not used.
- React or Angular etc are used to create Frontend and use Spring to create JSON.

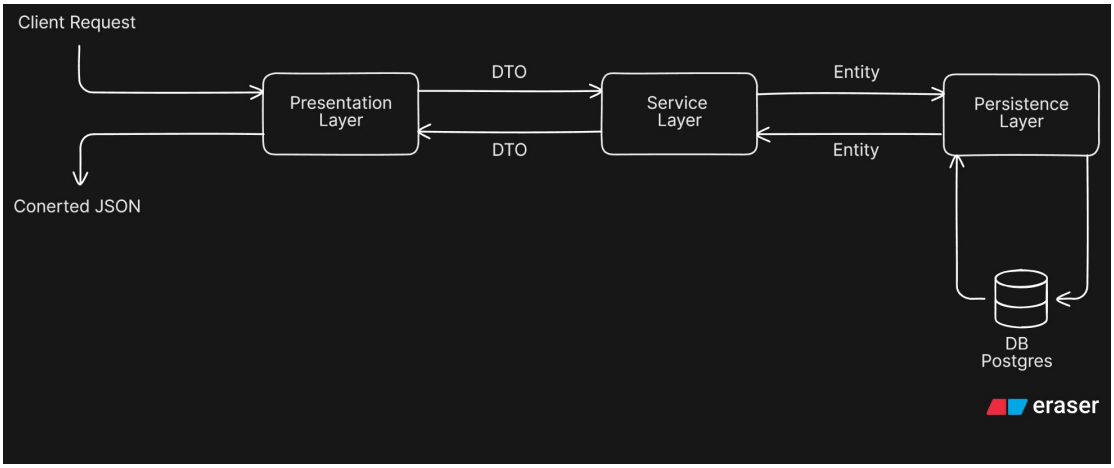
➢ **How does a Web Server works in Spring Boot?**



- The client request cannot go to the **Dispatcher Servlet** directly as it's a Java/Spring entity and not able to handle http request directly.
- So the request go to a **web server** (tomcat, jetty etc.. Spring Boot uses Tomcat by default) and then it transfer those requests to **Dispatcher Servlet**.

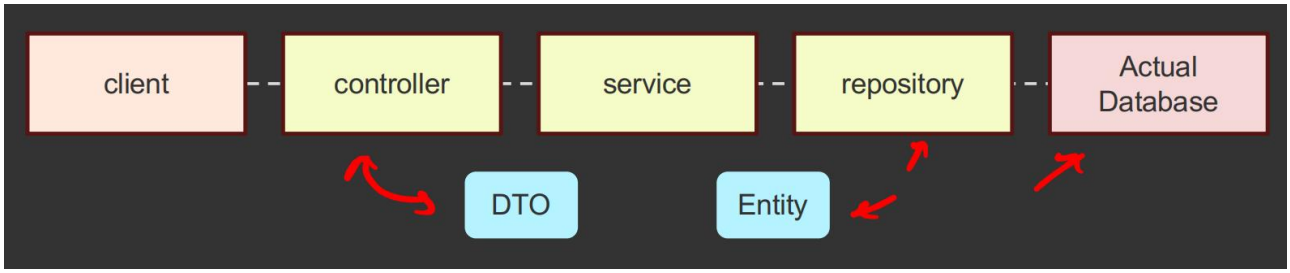
- We can have multiple requests like *profile request* or *admin request* or *reels request* etc etc. So these routing part is done in here i.e. **Dispatcher Servlet**.
  - 2 Objects i.e. **HttpServletRequest** and **HttpServletResponse** are created in this part only.
  - In that **HttpServletRequest** contains everything like headers, query, ids params.
  - After doing the things, we can update the **HttpServletResponse** and return it as the response.
- 

- 3 layered architecture



- **Layers**
  - **Presentation Layer** *(Handle Request and Response)*
    - Request from client will be handled by this
    - Communication between *Presentation* and *Service* layer happens with transfer of **DTO** *(Data Transfer Object)*
    - Lets say user is trying to login, so payload of the request may be containing username, password, email etc.. those things will be converted to an Object (DTO) and sent to the *Service* Layer.
  - **Service Layer** *(Business Logic)*
    - It'll check the things like is the user is valid with the help of Persistence Layer.
    - Service and Persistence layer have conversation by transferring *Entity*.
  - **Persistence Layer** *(Data Access Layer)*
    - It has only the access to database.
    - So, *Service* layer verifies the things with the help of this *Persistence* layer only.
- In simple terms: A DTO is the structure received from (or sent to) the user in API requests/responses, and an Entity is the structure used to store data in the Database."

Presentation Layer

- 

- **@RestController**
  - It marks a class as a REST controller and automatically applies *@ResponseBody* to every method.
  - ```
    @RestController  no usages
    public class EmployeeController {
    ```
  - You can see in the **@RestController**, it has one annotation called **@ResponseBody**
    - ```
      @Controller
      @ResponseBody
      public @interface RestController {
          @AliasFor(
                  annotation = Controller.class
      ```
    - It ensures the returned Java object is converted to JSON/XML and written to the HTTP response body using message converters.
  - To convert incoming JSON/XML to Java objects, Spring uses **@RequestBody**, not **@ResponseBody**.
  - If you write **@Controller** instead of **@RestController** then it'll think "user is the name of a view template (i.e. User.jsp, User.html) and it'll try to load the UI page and fail".
    - To make it return JSON then you need to write **@ResponseBody** in each method.
- **NOTES**
  - Spring MVC provides an annotation-based programming model where **@Controller** and **@RestController** components use annotations to express request mappings, request input, exception handling, and more.
  - The **@RestController** annotation is a shorthand for **@Controller** and **@ResponseBody**, meaning all methods in the controller will return JSON / XML directly to the response body.

- With **@RestController**, we don't need to explicitly write **@ResponseBody** for each methods to make it return *JSON*. (But with **@Controller** we need to write explicitly).

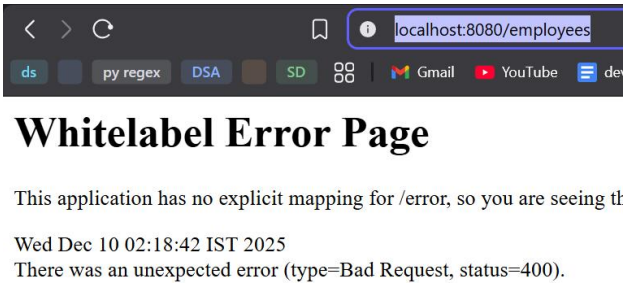
- **Request Mappings**
  - You can use the **@RequestMapping** annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.
  - **@GetMapping**, **@PostMapping** ..etc are there for **GET, POST, ..**etc type of requests.
- See the below code

```
@RestController  no usages
public class EmployeeController {

    @GetMapping(path = "/getSecretMessage")  no usages
    public String getMySuperSecretMessage() {
        return "Secret Message: abcd87957hw4895";
    }
}
```

  - Now this path **"/getSecretMessge"** is exposed.
  - But the question is how was it become live just by writing the code here? We didn't even used this method anywhere to make it live.
    - So, as we discussed earlier, Component Scan happens for all the files present on the same or child directories where the **main** method file is placed.
    - **@RestController** contains **@Controller** inside it; and **@Controller** contains **@Component** inside it.
    - It'll check for all the *annotations* and makes Bean out of those directly and use.
- To create the DTO, there is no specific annotations.

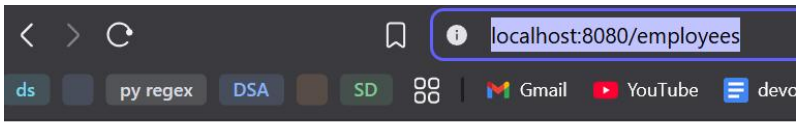
```
public class EmployeeDTO {  no usages
    private Long id;  no usages
    private String name;  no usages
    private String email;  no usages
    private Integer age;  no usages
    private LocalDate dateOfJoining;  no usages
    private Boolean isActive;  no usages
}
```

- See the below use cases to pass **params** in URL
  - ```
    @GetMapping("/employees/{employeeId}/{someMore}")  no usages
    public EmployeeDTO getEmployeeById(@PathVariable String employeeId,
                                       @PathVariable String someMore) {
    ```
    - If you see here I have set the **argument** names in the method same as the **params** name in the URL.
    - If you give different names then it'll not work.
    - The fix for this i.e. if you want to give different argument names then use **@PathVariable("real-param-name") DataType newArgument**
    - ```
      @GetMapping("/employees/{employeeId}/{someMore}")  no usages
      public EmployeeDTO getEmployeeById(@PathVariable("employeeId") String id,
                                         @PathVariable("someMore") String more) {
      ```
  - You can see the output will be in JSON format even if we didn't explicitly convert this to JSON.
    - ```
      {
          "id": 89734394873849,
          "name": "Alok",
          "email": "something@gmail.com",
          "age": 24,
          "dateOfJoining": "2020-01-01",
          "active": true
      }
      ```
    - There is a library **jackson** which does this things.
- To get the **query** values, use the annotation **@RequestParam**
  - ```
    @GetMapping(path = "/employees")  no usages
    public String getAllEmployees(@RequestParam Integer age) {
    ```
  - The url should be like: **/employees?age=40** (something like this)
  - But if you open only **/employee** it'll not load the page
  - 

  - It is because the query param is mandatory here.
  - We need to make it optional. **@RequestParam(required = false)**

```
@GetMapping(path = "/employees")  no usages
public String getAllEmployees(@RequestParam(required = false) Integer age) {
```

- Now you can see the page is loading



Hi age = null

- Instead of writing **/employees** everywhere, we can set the path for **@RequestMapping** at the top level.

```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {
```

- Now you don't need to write **/employee** in every methods.

```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {

    @GetMapping   // here no need to define anything  no usages
    public String getAllEmployees(@RequestParam(required = false) Integer age) {
        return "Hi age = " + age;
    }

    @GetMapping("/{employeeId}")  no usages
    public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) {
        return new EmployeeDTO(employeeId);
    }
}
```

- Other mappings are also there: **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping**
- There are so many **@Request** annotations to get the **query, body, header** etc etc.

```
@ RequestParam or
@ RequestMapping
@ RequestBody org
@ RequestAttribut
@ RequestHeader o
@ RequestPart org
@ RequestScope or
```

- Fjfdlfd
- Fdf

---------- **SOME POINTS** ----------

- There is a dependency called **lombok** which provides **getters, setters, constructors** by default. No need of defining this by our own.

```
@Getter  7 usages
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class EmployeeDTO {
    private Long id;
    private String name;
```

- **spring.datasource**
  - It is a spring boot configuration property used to set up *database connection*.
  - It tells spring boot:
    - Which database to connect to
    - How to connect
    - Which driver to use
    - Database username & password.

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

- **Presentation Layer**, **Service Layer**, **Persistence Layer**
  - These 3 are just architectural patterns. It has nothing to do with **spring/spring boot** framework.
  - Spring Boot just makes it easy to implement these architectural patterns.
  - Spring Components for the Layer
    - **Presentation** -------- **Controller** (**@RestController**)
    - **Service** ---------- **Service** (**@Service**)
    - **Persistence** ---------- **Repository** (**@Repository**)

```
Controller → Service → Repository → JPA → Hibernate → JDBC → Driver → Database
```

  - This is how the work happens between each layers.

Level	What It Does	VERY Short Explanation
Repository (Spring Data)	Converts repository methods (`save`, `findAll`) into JPA API calls	"Method → JPA function"
JPA (Specification)	Provides API (`persist`, `find`, `remove`) but does NOT implement them	"Rules + interfaces only"
Hibernate (JPA Provider)	Implements JPA functions → generates SQL → talks to JDBC	"Real ORM + SQL generator"
JDBC	Sends SQL to the database and gets results	"Communication pipe"
Database Driver	Converts JDBC calls to DB-specific protocol	"Translator"
Database	Stores and returns actual data	"Storage engine"

- You call high-level functions *(save(), findAll(), findByName())*
- Spring Data + JPA + Hibernate automatically generate optimized SQL for you

NOTE (Very Very Important; workflow from Repository level to Database level)

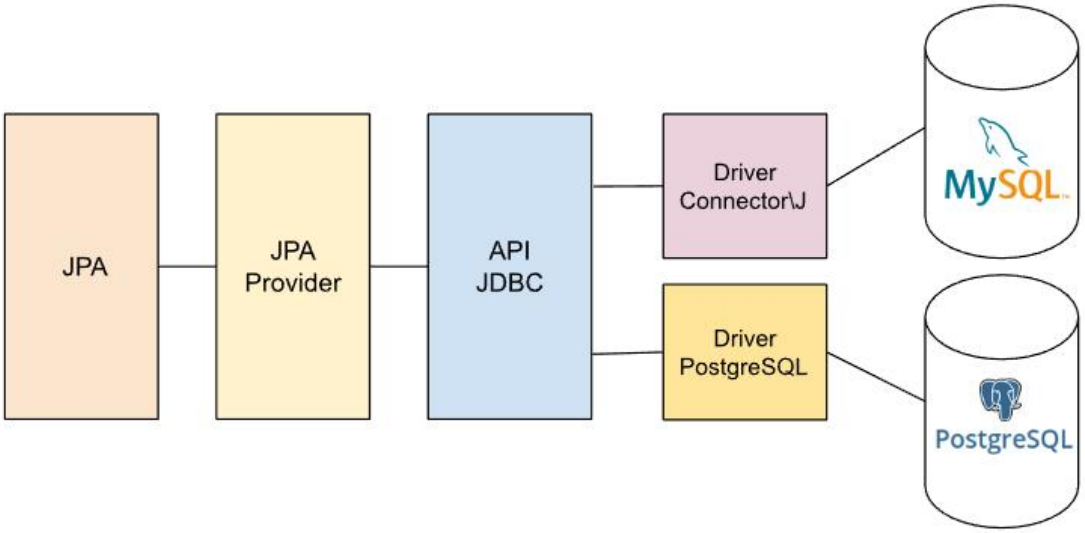
- There are **two** levels in Spring Data JPA:
  - Repository level
  - EntityManager (JPA) level.
- JPA itself consists of only **interfaces** (no implementation) and the **EntityManager** API.
- When we call a repository method like *findById*, Spring Data JPA creates a proxy implementation of the repository interface that contains implementations of high-level methods (findAll, findById, save, etc.).
  - **Proxy implementation** means Spring creates a class at runtime that acts as the real implementation of your repository interface — even though YOU never wrote that class.
  - It takes the implementations from a class called **SimpleJpaRepository** which has implementation of all the abstract methods of **JpaRepository** interface.
- That proxy internally delegates to the EntityManager, which triggers low-level JPA operations such as persist(), merge(), find(), and remove().
  - These methods are low level
  - It is there in between Repository and Hibernate level
- Up to this point, everything is at the JPA level (only rules + API, no actual SQL).
- Next, the call goes to the JPA provider (Hibernate).

- Hibernate implements JPA and converts entity operations into actual SQL queries.

- After Hibernate generates SQL, it sends the SQL to JDBC.
- JDBC is responsible for:
  - creating/maintaining a secure connection to the database, and acting as the transport layer to send Hibernate's SQL to the database driver.
- Finally, the database driver sends the SQL to the database server for execution.

-
- F
- F
- F
- F
- F

Persistence Layer



-
- **JPA**: Java Persistence API, nowadays its called ***"Jakarta Persistence API"***
- Databases are also some servers but they hold some unique logic so that they can hold some data and query through it.
- Databases provide **JDBC** *(Java Database Connection)* drivers. Java uses JDBC to communicate with the database using those drivers.
- JPA is a specification. Hibernate is the most commonly used JPA provider that actually performs ORM **(object–relational mapping)**.
  - **Object-relational mapping** means *JPA* works with Java *objects*, the *database* works with *tables*, and the *JPA provider* (like Hibernate) maps the Java *objects* to the database *tables* and vice versa.
  - We can imagine JPA as a set of rules, and a JPA provider as the implementation of those rules.
  - JPA defines interfaces and abstract classes that describe how entities should be mapped, persisted, and managed.
  - But JPA itself does not contain actual persistence logic.
  - A JPA provider (like Hibernate) implements those rules and performs the real work such as generating SQL, mapping objects to database tables, managing relationships, and handling transactions.
- **JPQL**
  - It is an object-oriented query language used in JPA to query entities, NOT database tables.
  - You write queries on entity classes, not table names
  - You use field names, not column names

- It works on Java objects, not database records
- *The JPA provider (Hibernate) converts JPQL → SQL internally*

➢ In simple terms:

- **JPA** = What to do (not how to do it)
- **JPS Provider (Hibernate)** = How to do it
- **JDBC** = The pipe through which SQL is sent to the DB
- **Driver** = Translator for a specific database
- **Database** = The storage engine

➢ **@Entity** is used to define a class as an entity.

- ```
  @Entity  no usages
  public class EmployeeEntity {
  ```
- It'll just tell the Spring Data JPA (Hibernate) that "this is the Java class. You need to convert it to table and store in the database".
- ```
  @Entity  no usages
  @Table(name = "employee")
  public class EmployeeEntity {
  ```
  - If you don't give that **@Table** then it'll create the table with name **EmployeeEntity**

➢ There are annotations to define the SQL things. For example:

- ```
  @Id  no usages
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  ```
- Here **@Id** means ***Primary Key*** in SQL.
- **@GeneratedValue** with **strategy** means the strategy of incrementing the **Id**.
- Different databases like mysql, postgres are having different type of strategy. So I selected *AUTO* so that it'll work everywhere.

➢ After creating the **Entity**, you need to create the **Repository**.

- The annotation will be **@Repository**.
- ```
  @Repository  no usages
  public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {
  }
  ```
- You need to extend **JpaRepository** (which is an interface) which is having all the *definitions of methods* like findAll, findAllById etc etc.
- Basically **JpaRepository** provides set of **CRUD** operations and query methods.
  - Generic Interface

- Predefined Methods
- Custom Queries

- Here the type will be like:
  - 1st one is type of **entity**
  - 2nd one is type of **primary key (@Id)**
- The **@Repository** is containing **@Component** so **bean** will automatically be created by Spring.

```
@GetMapping("/{employeeId}")  no usages
public EmployeeEntity getEmployeeById(@PathVariable(name = "id") Long id) {
    return employeeRepository.findById(id).orElse( other: null);
}

Change signature
@GetMapping   // here no need to define anything  no usages
public List<EmployeeEntity> getAllEmployees() {
    return this.employeeRepository.findAll();
}

@PostMapping  no usages
public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity inputEmployee) {
    return this.employeeRepository.save(inputEmployee);
}
```

- Created some controller methods inside the **EmployeeController** class using **EmployeeEntity**.
- But this shouldn't be done; Repository part and Controller part should not be mixed up.
- I have used the dependency **h2** which is used to simulate database things without using a real database.
- If **Entity** class has **private variables** and you have not defined **Getters** and **Setters** then you will not get the result because it'll not be able to get those values.

- You can see in the terminal
  - `Added connection conn0: url=jdbc:h2:mem:0703e0a3-b9ec-498e-902e-8abbb19c1115 user=SA`
  - In between the url link, there is something **"mem"** ; it means it'll sotre the entries in the memory till the application is running.
- **spring.jpa.hibernate.ddl-auto** tells Hibernate how to handle your database tables when the application starts.

Service

- It will be having access to both **DTO** and **Entity**.
- As it'll fetch the things from **Repository**, so it'll be in the form of **Entity**. But whatever it'll return, that will be going to **Controller** layer, so that return type should be of type **DTO** (not Entity).
  - To convert this, we can get the values of Entity one by one, then create one DTO type of object and return.
  - Or we can take help of the dependency: **"modelmapper"**
- As we need to convert **Entity** to **DTO** or vice-versa, so **ModelMapper** object will be used multiple times.
  - Its better to create one **Bean** of that and use that everywhere.
  - So, create one class **MapperConfig** and write the **@Configuration** and **@Bean** there.
  -
```
@Configuration  4 usages
public class MapperConfig {

    @Bean  no usages
    public ModelMapper getModelMapper() {
        return new ModelMapper();
    }
}
```
  - Now make sure, **Controller** part will be using **DTO**; **Repository** part will be using **Entity**; and **@Service** will be used to map those 2 properly depending upon the usage.
  -
```
@Service  3 usages
public class EmployeeService {
    private final EmployeeRepository employeeRepository;  4 usages
    public final ModelMapper mapper;  5 usages

    public EmployeeService(EmployeeRepository employeeRepository, ModelMapper mapper) {  no us
        this.employeeRepository = employeeRepository;
        this.mapper = mapper;
    }

    public EmployeeDTO getEmployeeById(Long id) {  1 usage
        EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse( other: null);
        return mapper.map(employeeEntity, EmployeeDTO.class);
```
- F
- F

- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- Ff
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
-