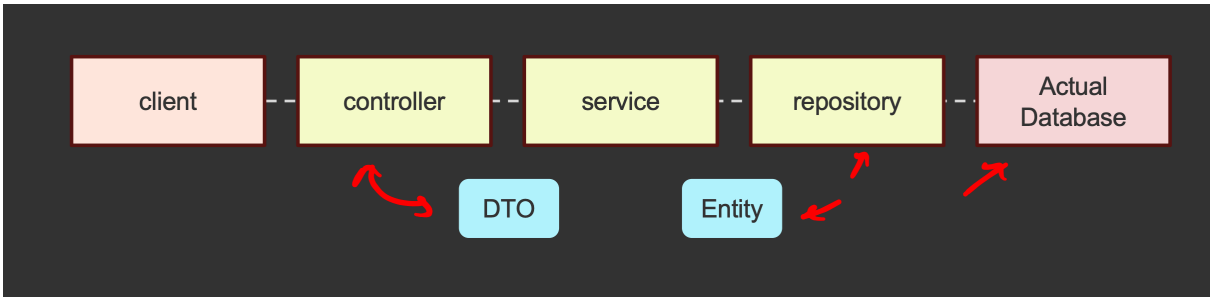
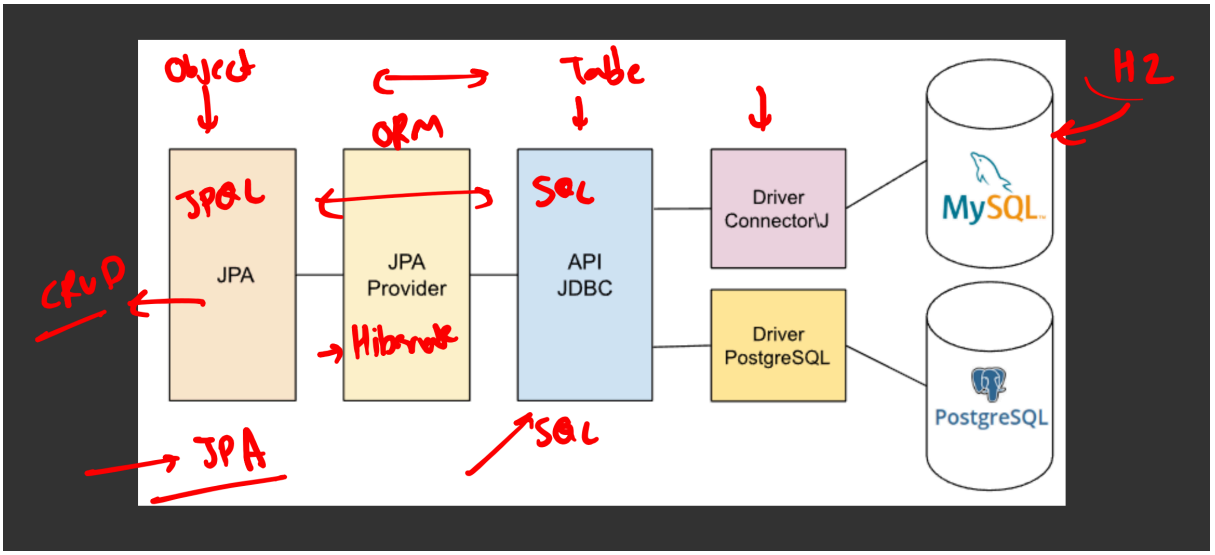
# Persistence Layer

## Spring Boot Web Project Structure




## Java Persistence API - JPA




## Spring H2 Database

Spring Boot makes it very easy to set up an in-memory H2 database for development and testing purposes. H2 is a lightweight, fast, and opensource relational database engine that can run in both in-memory and persistent modes.

Dependency
`<groupId>com.h2database</groupId>`

## Entity Annotation

The `@Entity` annotation in Spring and Java Persistence API (JPA) is used to mark a class as a persistent entity, meaning it represents a table in a relational database. This is a fundamental part of the ORM (Object-Relational Mapping) paradigm, where Java objects are mapped to database tables.

Key Points of `@Entity`:

- Class-Level Annotation
- Primary Key
- Automatic Table Mapping

## JPARepository Interface

The JPARepository interface in Spring Data JPA provides a set of CRUD (Create, Read, Update, Delete) operations and query methods for interacting with the database.

Key Points of CrudRepository:

- Generic Interface
- Predefined Methods
- Custom Queries

Let’s decalre an Entity class that is a reflection of the “table” with columns we will have at the persistence layer.

```
@Entity
@Table(name = "employees")
public class EmployeeEntity {
    private Long id;
```

```
    private String name;
    private String email;
    private Integer age;
    private LocalDate dateOfJoining;
    private boolean active;
}
```

We define the fields that are part of it and make it as Bean available for the spring boot to create/update in database when we launch the application. We are required by nature to define which is the field of `@Id` else that will not work

```
@Entity
@Table(name = "employees")
public class EmployeeEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    private String email;

    private Integer age;

    private LocalDate dateOfJoining;

    private boolean active;

}
```

Since the value will be of a sequence that gets auto-incremented we let the framework decide whichever is best choice based on the connected DB. We will add lombok dependency that will give extensions of getters/setter and more,

```
<dependency>
    <groupId>org.projectlombok</groupId>
```

```
    <artifactId>lombok</artifactId>
</dependency>
```

For time being, lets also have an in-memory database called H2 that words well for the simple CRUD operations we are planning to make.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

that makes the definition of entity more easy to work with and we would have something like below,

```
@Entity
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "employees")
public class EmployeeEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private Integer age;
    private LocalDate dateOfJoining;
    private boolean active;
}
```

Once we have the table we are also required to connect this spring boot application to actual database via spring JPA (Jakarta Persistence API)

```
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEnti
```

```
ty, Long>{}
```

Doing this gives us a repository Bean that have in-built CRUD operations and more can be developer defined with JPQL as custom implementations. To acheive this we also need some extra dependencies,

```
<dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
  </dependencies>
```

Now that the DB layer is completed, lets add the config and then go towards the controller layer. At the config, we must have the following

```
spring.h2.console.enabled=true

spring.datasource.username=<db_username>
spring.datasource.password=<db_password>

spring.datasource.url=jdbc:h2:file:/Users/path/to/db
spring.jpa.hibernate.ddl-auto=update
```

Once these are defined we have records persisted to a file on disk which acts as DB and on restart we can have them and not loose. Moving on to the controller layer, we need to make endpoints that support these repository activities,

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeRepository employeeRepository;

        // constructor based dependency injection
    public EmployeeController(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

        // get a single record by id
    @GetMapping(path = "/{employeeId}")
    public EmployeeEntity getEmployeeById(@PathVariable Long employeeI
d) {
        return employeeRepository.findById(employeeId).orElse(null);
    }

        // get all records of an entity/table
    @GetMapping
    public List<EmployeeEntity> getAllEmployees() {
        return employeeRepository.findAll();
    }

        // add a new value to the entity/table
    @PostMapping
    public EmployeeEntity createNewEmployee(@RequestBody EmployeeEnt
ity inputEmployee) {
        return employeeRepository.save(inputEmployee);
    }
}
```