SPRING BOOT

Introduction

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible

Spring initializer

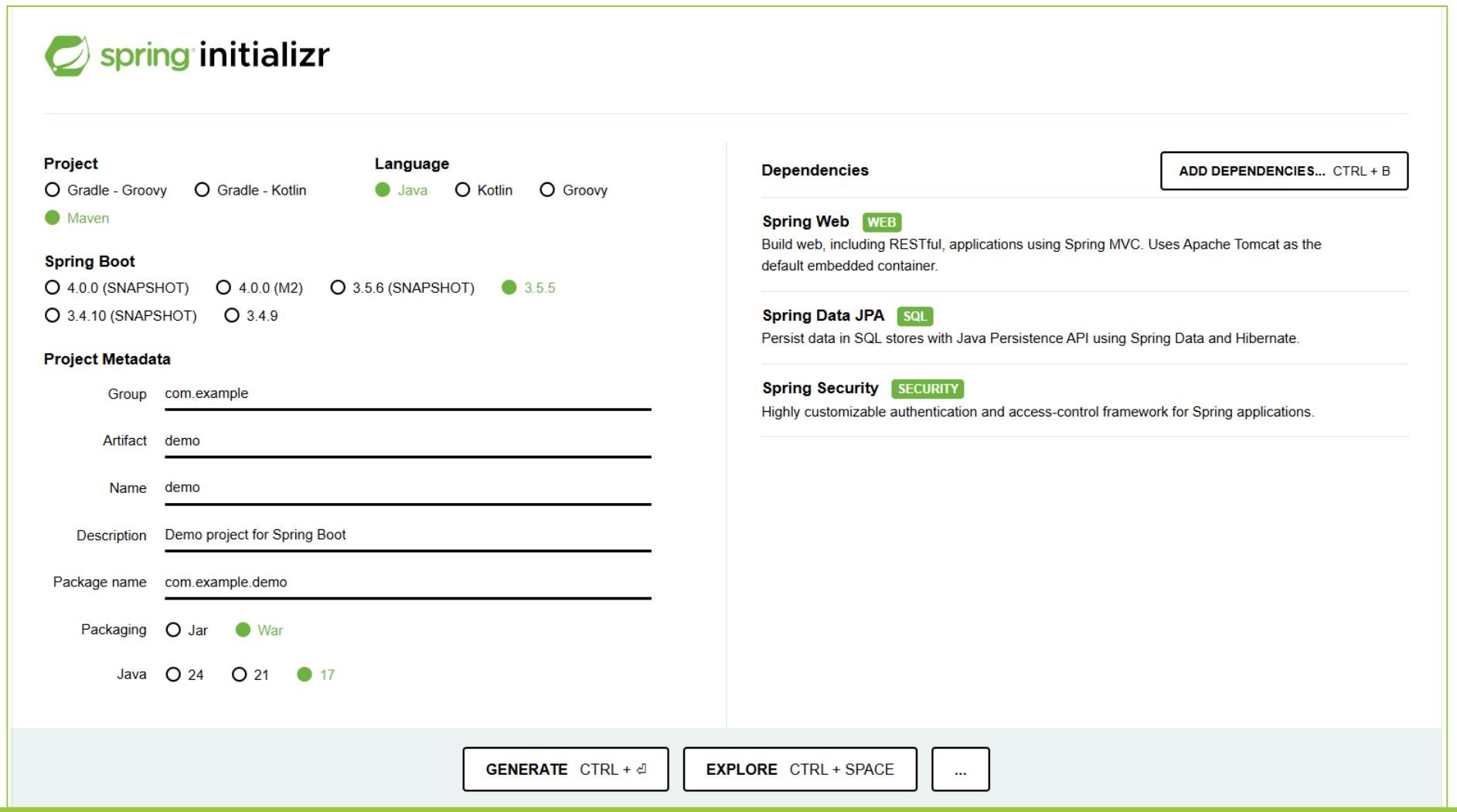
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration
- Quickly create a starter Spring project
- Select your dependencies
- Creates a Maven/Gradle project
- Import the project into your IDE: Eclipse, IntelliJ, NetBeans etc …

Spring initializer

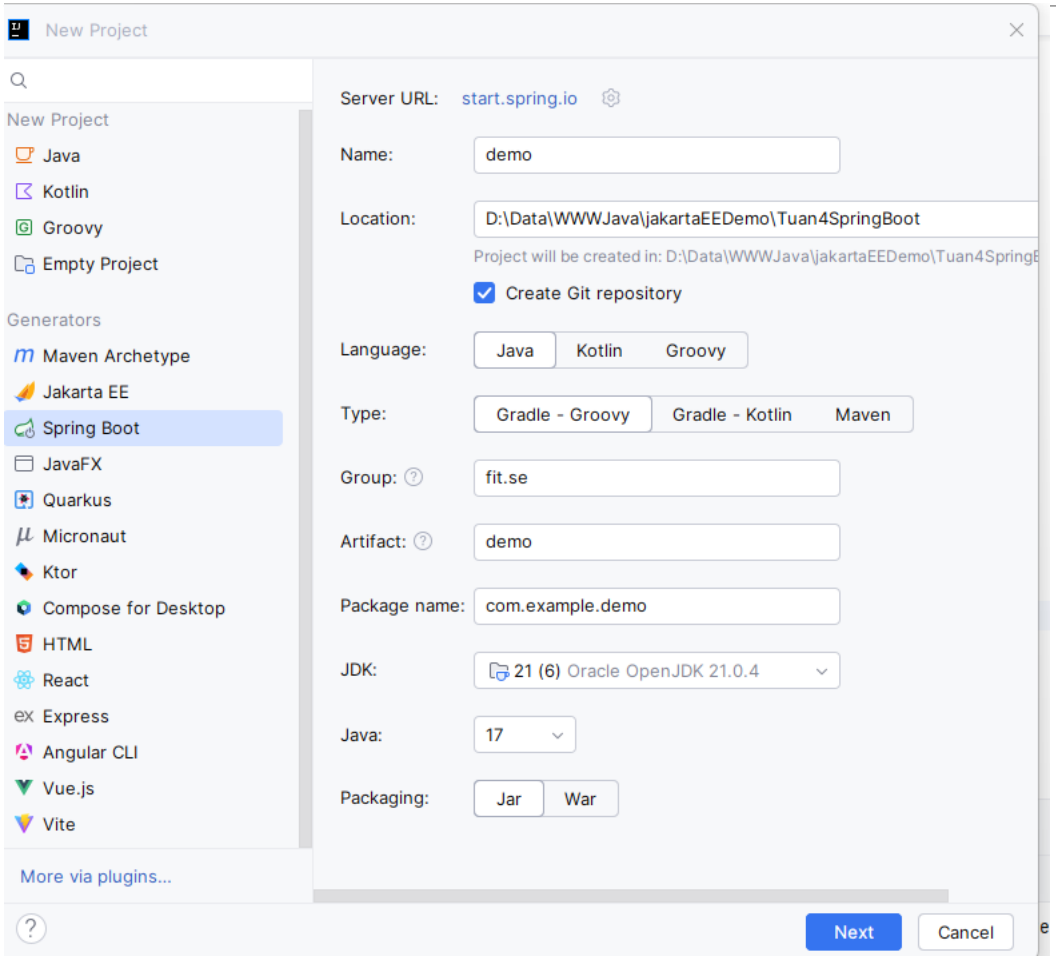
There are 3 main ways:

1. Using Spring Initializr Web (start.spring.io)
2. Using Spring Initializr directly in IntelliJ IDEA
3. Manually creating a Maven/Gradle project and adding Spring Boot

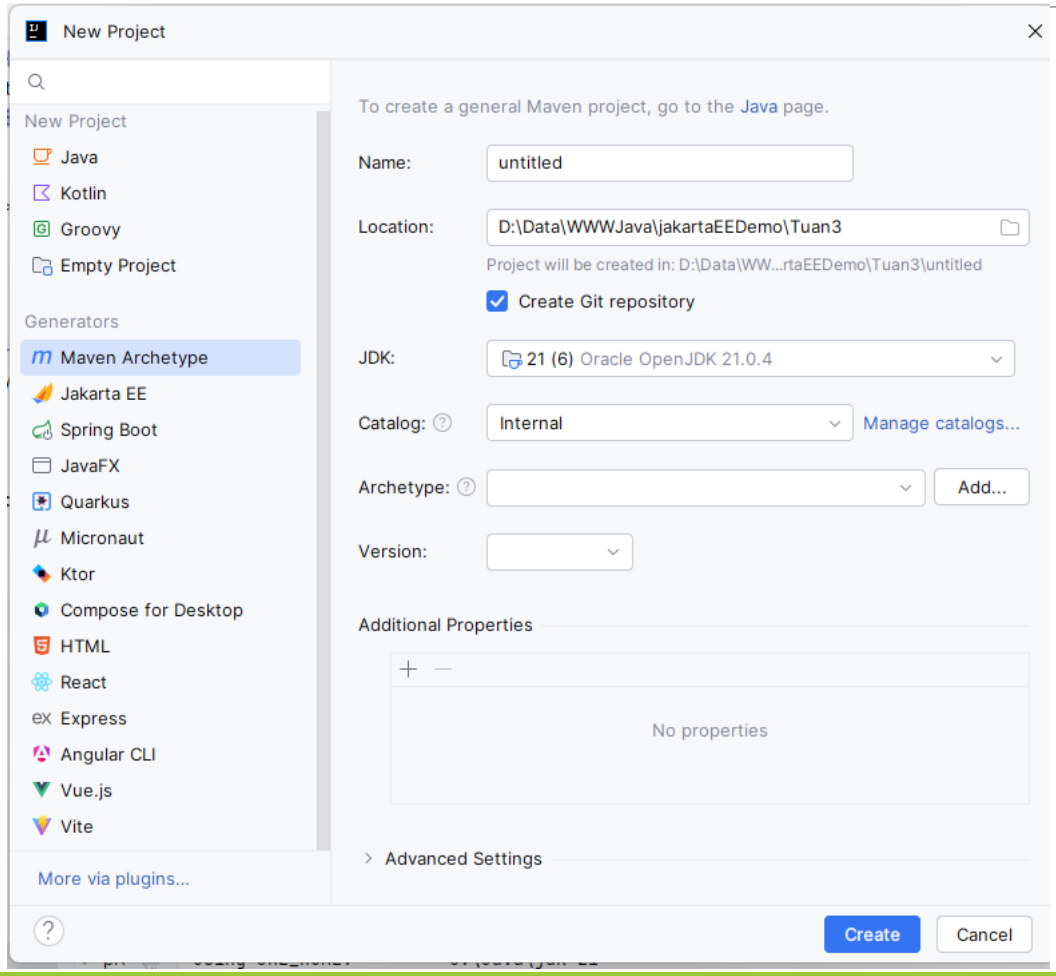
Spring initializer - start.spring.io



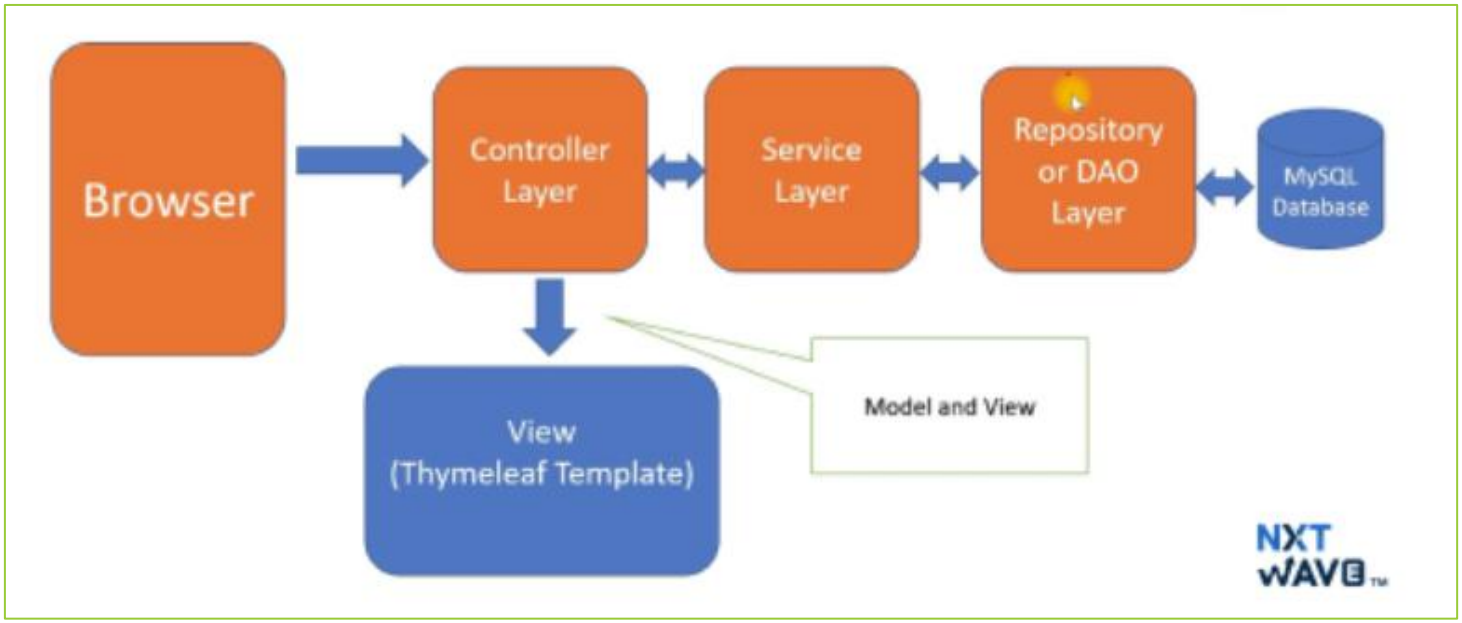
Spring initializer - IntelliJ



Spring initializer – IntelliJ (Maven)



Spring Boot Architecture



Dependencies

1. Core: spring-boot-starter
2. Web Applications (REST APIs / MVC): spring-boot-starter-web:
3. Data Access (JPA, JDBC, NoSQL):
   - spring-boot-starter-data-jpa
   - h2/mysql-connector-j/mssql-jdbc/spring-boot-starter-data-mongodb

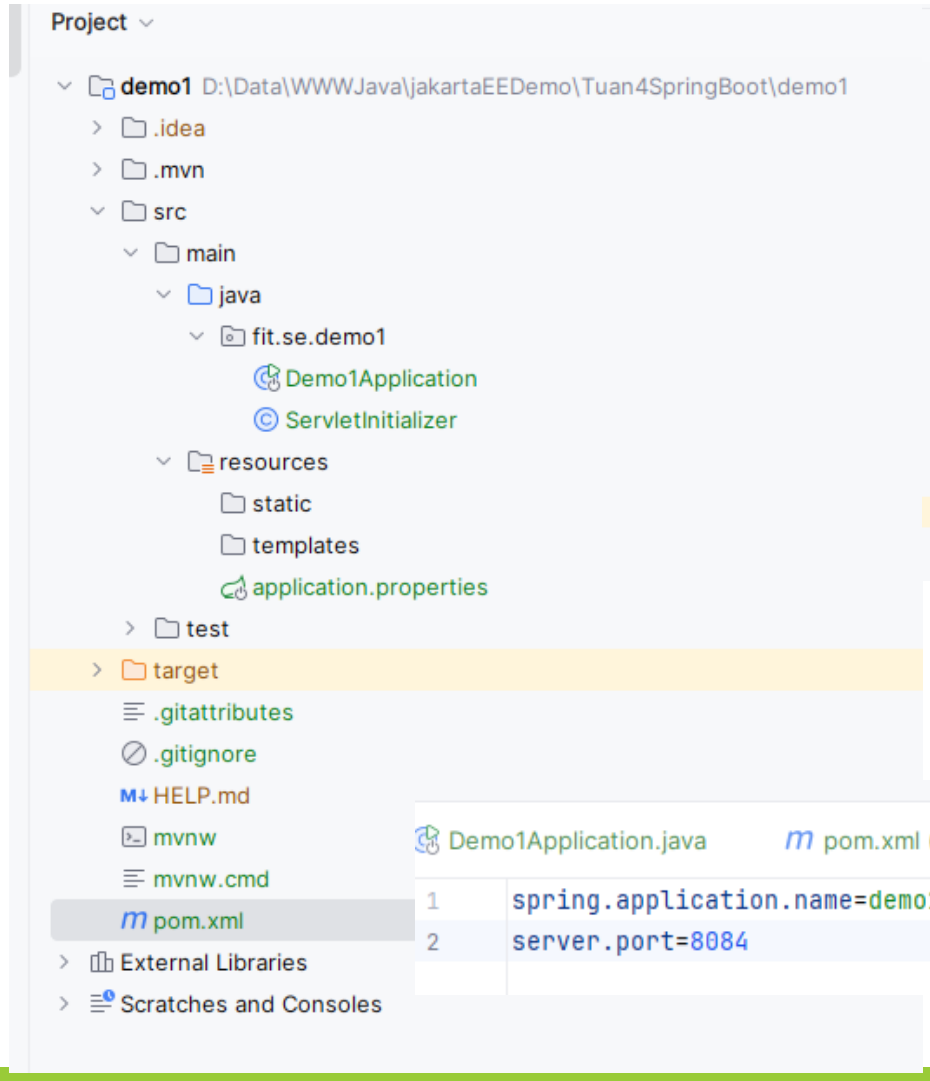
4. Template Engines (if building web pages): spring-boot-starter-thymeleaf

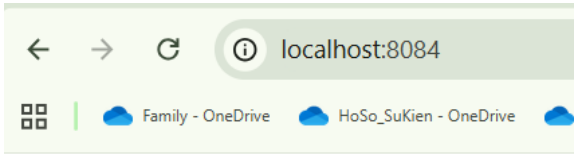
5. Utilities

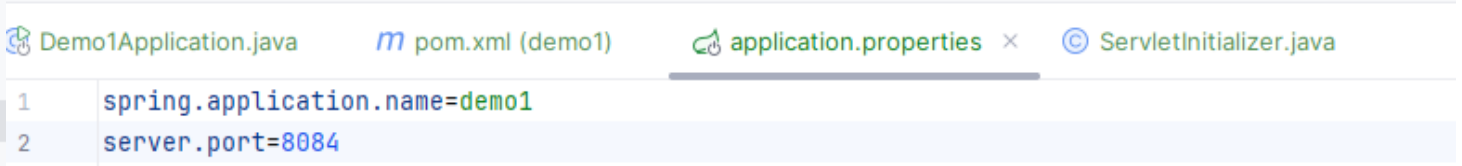
- Validation (JSR 380, Jakarta Validation API): spring-boot-starter-validation
- Lombok (to reduce boilerplate): Lombok
- DevTools (auto restart during development): spring-boot-devtools

6. Testing: spring-boot-starter-test

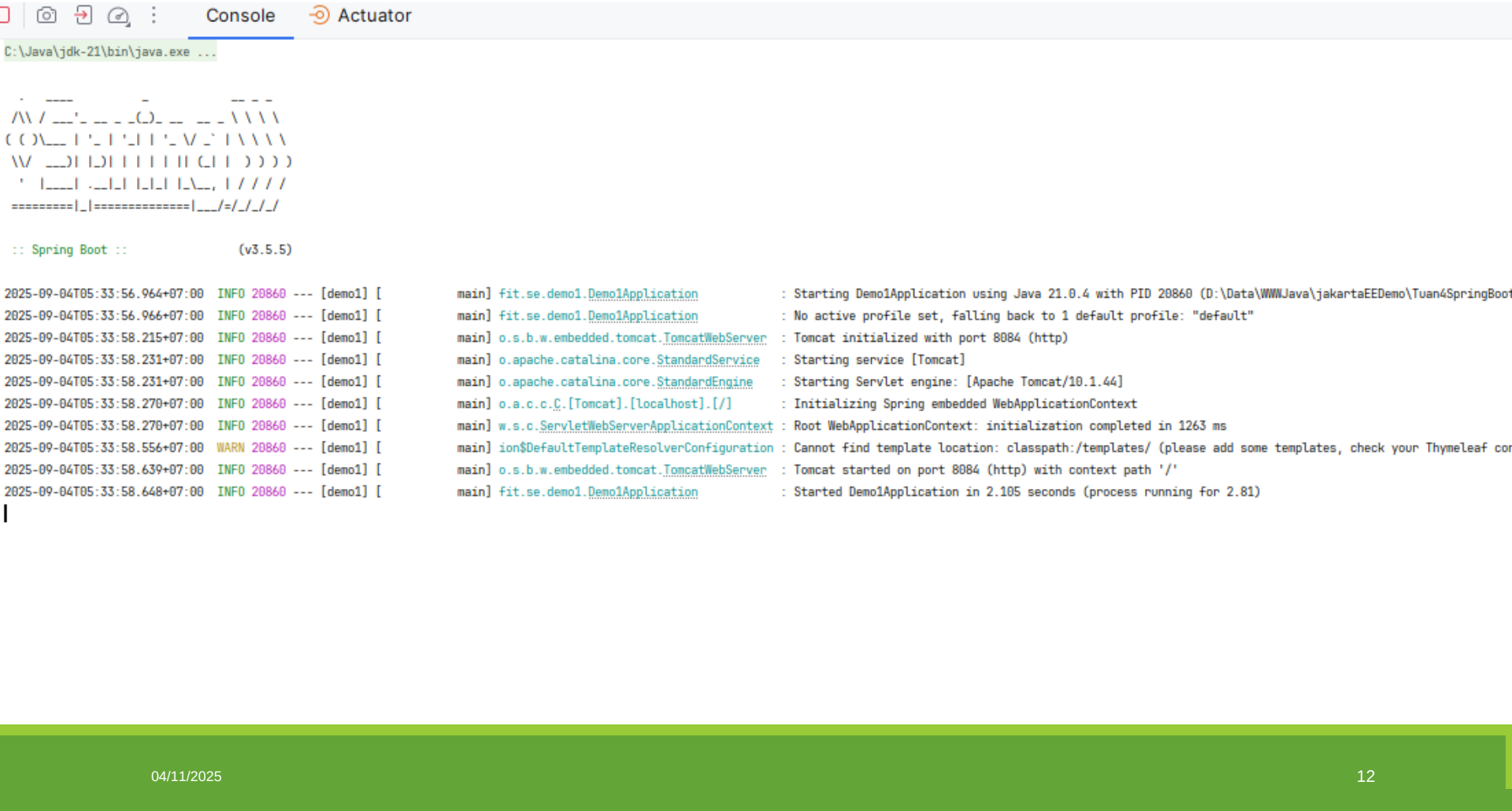
Project Structure







Running Example



Spring Data

Spring Data - Main modules

- Spring Data Commons - Core Spring concepts underpinning every Spring Data module.
- Spring Data JDBC - Spring Data repository support for JDBC.
- Spring Data R2DBC - Spring Data repository support for R2DBC.
- Spring Data JPA - Spring Data repository support for JPA.
- Spring Data KeyValue - `Map` based repositories and SPIs to easily build a Spring Data module for key-value stores.
- Spring Data LDAP - Spring Data repository support for Spring LDAP.
- Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.
- Spring Data Redis - Easy configuration and access to Redis from Spring applications.
- Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.
- Spring Data for Apache Cassandra - Easy configuration and access to Apache Cassandra or large scale, highly available, data oriented Spring applications.
- Spring Data for Apache Geode - Easy configuration and access to Apache Geode for highly consistent, low latency, data oriented Spring applications.

https://spring.io/projects/spring-data

Spring Data Commons

Spring Data Commons is part of the umbrella Spring Data project that provides shared infrastructure across the Spring Data projects. It contains technology neutral repository interfaces as well as a metadata model for persisting Java classes.

Primary goals are:

- Powerful Repository and custom object-mapping abstractions
- Support for cross-store persistence

Spring Data – Common (cont.)

- Dynamic query generation from query method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code
- Easy Spring integration with custom namespace

Spring JDBC

Accessing data with “pure” JDBC

The Spring Framework provides extensive support for working with SQL databases, from direct JDBC access. Java’s javax.sql.DataSource interface provides a standard method of working with database connections. Use other objects for execute SQL command:

- Connection
- Statement/PreparedStatement
- ResultSet
- …

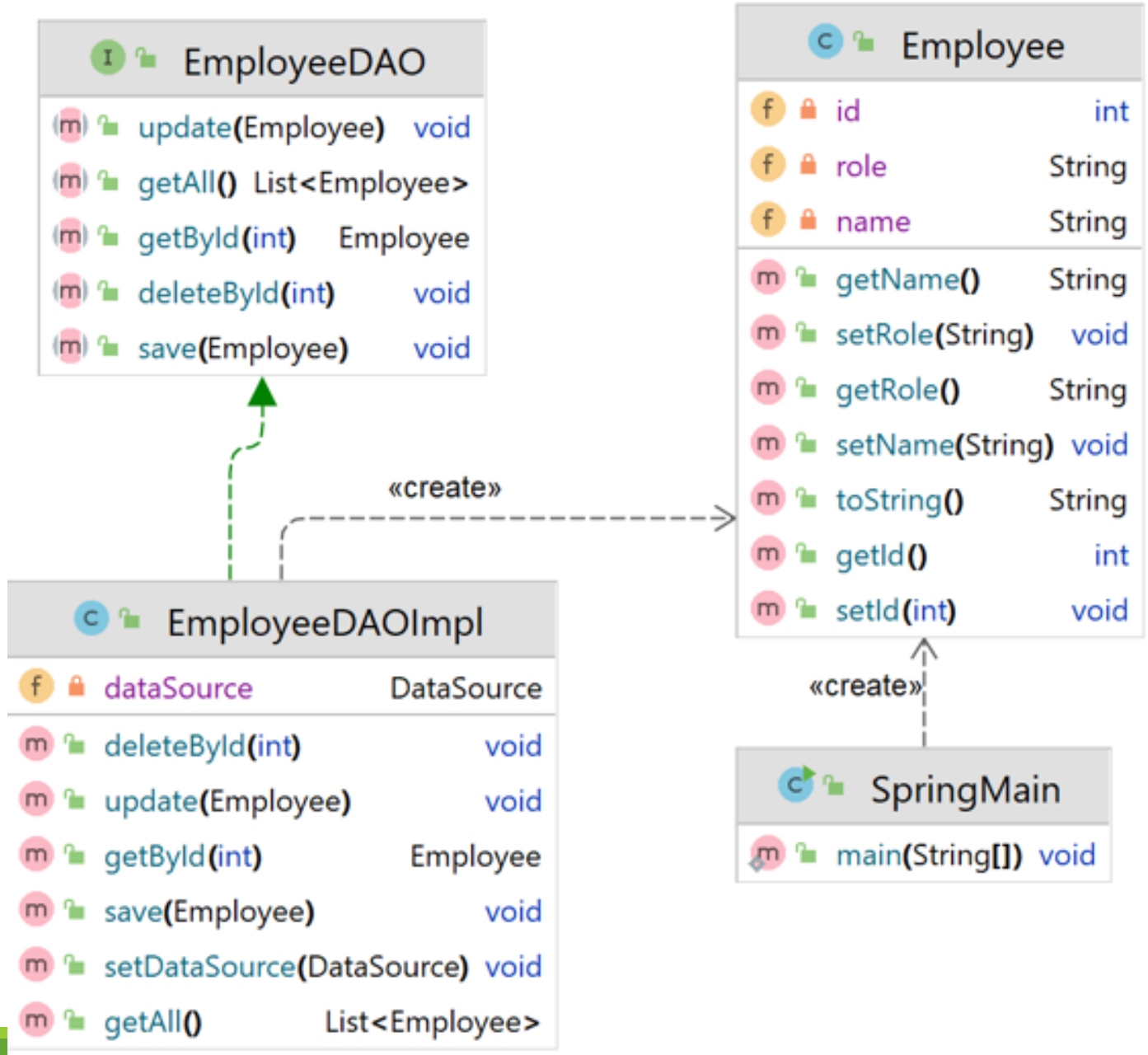
Configure DataSource:

main\resource\application.properties

application.properties

```
# MariaDB
spring.datasource.url=jdbc:mariadb://localhost:3306/sampledb
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create-drop
```

xample


EmployeeDAO
update(Employee) void
getAll() List<Employee>
getById(int) Employee
deleteById(int) void
save(Employee) void
Employee
id int
role String
name String
getName() String
setRole(String) void
getRole() String
setName(String) void
toString() String
getId() int
setId(int) void
«create»
EmployeeDAOImpl
dataSource DataSource
deleteById(int) void
update(Employee) void
getById(int) Employee
save(Employee) void
setDataSource(DataSource) void
getAll() List<Employee>
«create»
SpringMain
main(String[]) void

Example

Step 0. Create Spring Boot Project

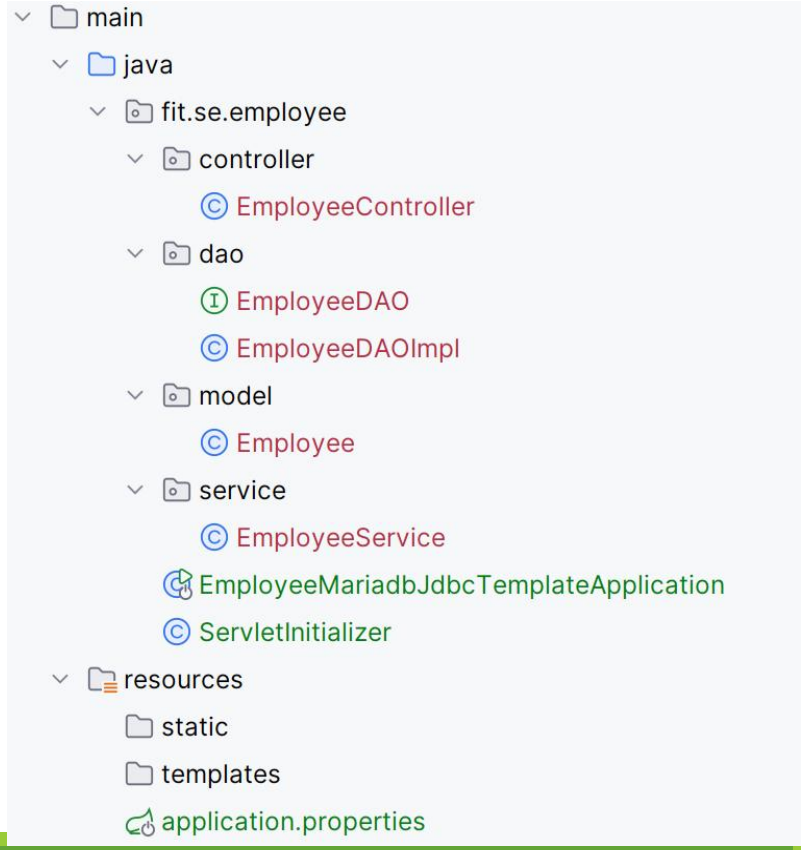
Spring Boot

Dependencies to include:

Spring Web

Spring JDBC

MariaDB Driver



Step 1: Config Datasource

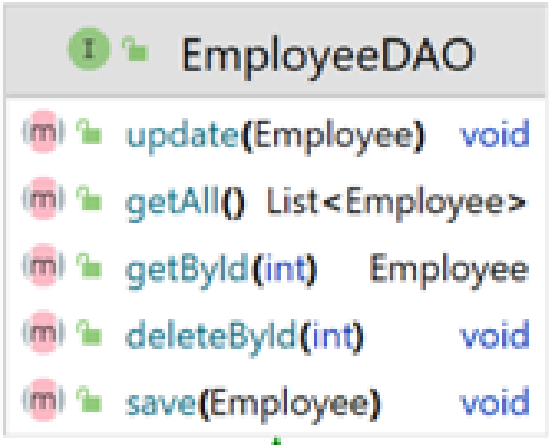
```
spring.datasource.url=jdbc:mariadb://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver

spring.sql.init.mode=always
spring.sql.init.platform=mariadb
```

Controller → Service → EmployeeDAO (interface) → EmployeeDAOImpl (execute SQL) → Database

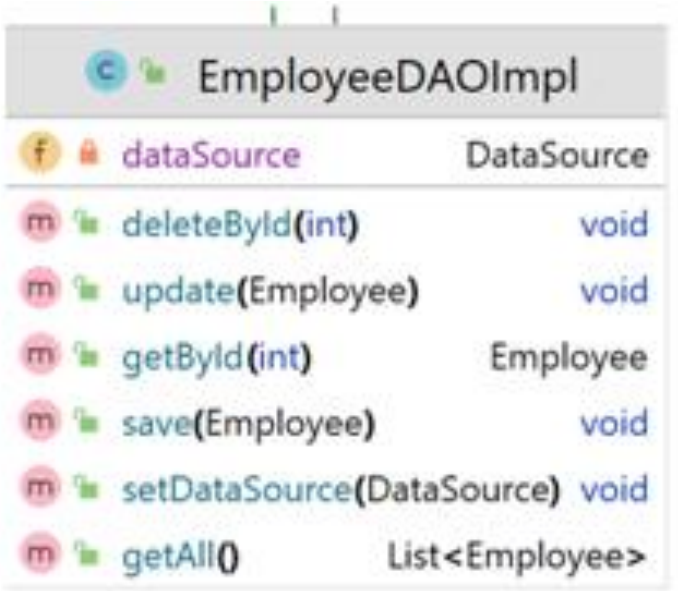
Step 2: Create Class Employee

Step 3: Create ***interface*** EmployeeDAO

```
public interface EmployeeDAO {  4 usages   1 implementation
    void save(Employee employee);  1 usage   1 implementation
    void update(Employee employee);  1 usage   1 implementation
    Employee getById(int id);  1 usage   1 implementation
    List<Employee> getAll();  1 usage   1 implementation
    void deleteById(int id);  1 usage   1 implementation
}
```



Step 4: Create Class EmployeeDAOImpl



Repository Layer

- @Repository – Data access component (DAO)
- @RepositoryRestResource – Expose repository as REST API (Spring Data REST)
- @Query – Define custom queries (JPQL/SQL)

```java
@Repository
public class EmployeeDAOImpl implements EmployeeDAO {
    private JdbcTemplate jdbcTemplate;  6 usages

    public EmployeeDAOImpl(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    private RowMapper<Employee> rowMapper = new RowMapper<>() {  2 usages
        @Override  no usages
        public Employee mapRow(ResultSet rs, int rowNum) throws SQLException, SQLException {
            return new Employee(
                    rs.getInt( columnLabel: "id"),
                    rs.getString( columnLabel: "name"),
                    rs.getString( columnLabel: "role")
            );
        }
    };
```

Service Layer

```
@Service  3 usages  new *
public class EmployeeService {
    private final EmployeeRepository repo;  8 usages

    public EmployeeService(EmployeeRepository repo) {  new *
        this.repo = repo;
    }

    public List<Employee> getAll() {  1 usage  new *
        return repo.findAll();
    }

    public Employee getById(int id) {  1 usage  new *
        return repo.findById(id).orElse( other: null);
    }

    public Employee save(Employee e) {  2 usages  new *
        return repo.save(e);
    }
```

(Step 4) Using JdbcTemplate

This is the central class in the JDBC core package.

It simplifies the use of JDBC and helps to avoid common errors.

It executes core JDBC workflow, leaving application code to provide SQL and extract results.

It executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package.

Using JdbcTemplate

Spring's JdbcTemplate class is auto-configured, and you can @Autowire them directly into your own beans.

In case of using named parameter, you should use the NamedParameterJdbcTemplate class

Query row(s) with Rows Mapper

```
private RowMapper<Employee> rowMapper = new RowMapper<>() {  2 usages
    @Override  no usages
    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException, SQLException {
        return new Employee(
                rs.getInt( columnLabel: "id"),
                rs.getString( columnLabel: "name"),
                rs.getString( columnLabel: "role")
        );
    }
};
```

Query row(s) with Rows Mapper

```
@Override  1 usage
public Employee getById(int id) {
    String sql = "SELECT * FROM employees WHERE id=?";
    return jdbcTemplate.queryForObject(sql, rowMapper, id);
}
```

```
@Override  1 usage
public List<Employee> getAll() {
    String sql = "SELECT * FROM employees";
    return jdbcTemplate.query(sql, rowMapper);
}
```

Query with BeanPropertyMapper

Using BeanPropertyMapper object will save you a lot of time.

```
// READ
public Employee findById(int id) {  no usages
    String sql = "SELECT * FROM employees WHERE id=?";
    return jdbcTemplate.queryForObject(sql,
            new BeanPropertyRowMapper<>(Employee.class),id);
}

// READ ALL
public List<Employee> findAll() {  no usages
    String sql = "SELECT * FROM employees";
    return jdbcTemplate.query(sql, new BeanPropertyRowMapper<>(Employee.class));
}
```

(Step 4) – CRUD Respository

```
public interface EmployeeRepository extends CrudRepository<Employee, Integer>,
        EmployeeRepositoryCustom {
    List<Employee> findByDepartmentId(Integer departmentId);  no usages  new *
}
```

CRUD Repository **Query Methods**

PersonRepository with query methods

```
interface PersonRepository extends PagingAndSortingRepository<Person, String> {

    List<Person> findByFirstname(String firstname);                                  ❶

    List<Person> findByFirstnameOrderByLastname(String firstname, Pageable pageable); ❷

    Slice<Person> findByLastname(String lastname, Pageable pageable);                ❸

    Page<Person> findByLastname(String lastname, Pageable pageable);                 ❹

    Person findByFirstnameAndLastname(String firstname, String lastname);            ❺

    Person findFirstByLastname(String lastname);                                     ❻

    @Query("SELECT * FROM person WHERE lastname = :lastname")
    List<Person> findByLastname(String lastname);                                    ❼
    @Query("SELECT * FROM person WHERE lastname = :lastname")
    Stream<Person> streamByLastname(String lastname);                                ❽

    @Query("SELECT * FROM person WHERE username = :#{ principal?.username }")
    Person findActiveUser();                                                         ❾
}
```

Step 5: Create Class EmployeeController

Controller Layer

- @Controller – MVC controller, returns views
- @RestController – REST API controller, returns JSON/XML
- @RequestMapping – General request mapping for URI
- @GetMapping,
@PostMapping,
@PutMapping,
@DeleteMapping – HTTP-specific mappings

Controller Layer

- @Controller – MVC controller, returns views
- @RestController – REST API controller, returns JSON/XML
- @RequestMapping – General request mapping for URI

`@GetMapping` trong class này → thực tế URL là `/api/employees`.

`@GetMapping("/{id}")` → URL thực tế là `/api/employees/{id}`.

```java
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {
    private final EmployeeDAO employeeDAO;  6 usages
    public EmployeeController(EmployeeDAO employeeDAO) {
        this.employeeDAO = employeeDAO;
    }
    @GetMapping
    public List<Employee> getAll() {
        return employeeDAO.getAll();
    }

    @GetMapping("/{id}")
    public Employee getById(@PathVariable int id) {
        return employeeDAO.getById(id);
    }
```

```java
@PostMapping
public void create(@RequestBody Employee employee) {
    employeeDAO.save(employee);
}

@PutMapping("/{id}")
public void update(@PathVariable int id, @RequestBody Employee employee) {
    employee.setId(id);
    employeeDAO.update(employee);
}

@DeleteMapping("/{id}")
public void delete(@PathVariable int id) {
    employeeDAO.deleteById(id);
}
```

Common Annotations

- @Autowired – Dependency injection
- @Qualifier – Specify bean to inject
- @Value – Inject values from properties
- @Configuration – Class with Spring beans
- @Bean – Define a bean inside configuration
- @Component – Generic Spring bean (superclass of @Service, @Repository, @Controller)

Spring Data JPA

Introduction

Spring Data JPA is used to reduce the amount of boilerplate code required to implement the data access object (DAO) layer.

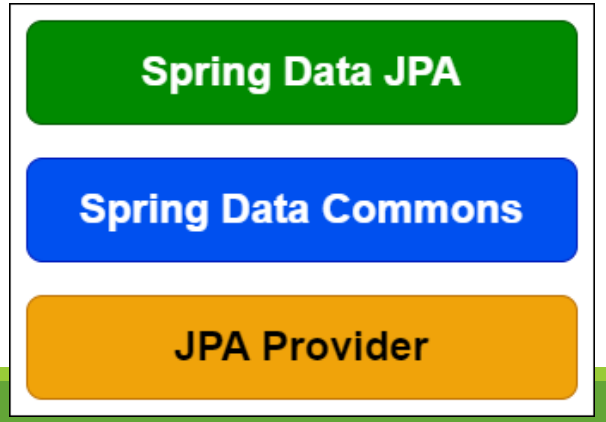
Spring Data JPA is not a JPA provider. It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider. If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following figure :

Spring Data JPA components

Spring Data JPA :- It provides support for creating JPA repositories by extending the Spring Data repository interfaces.

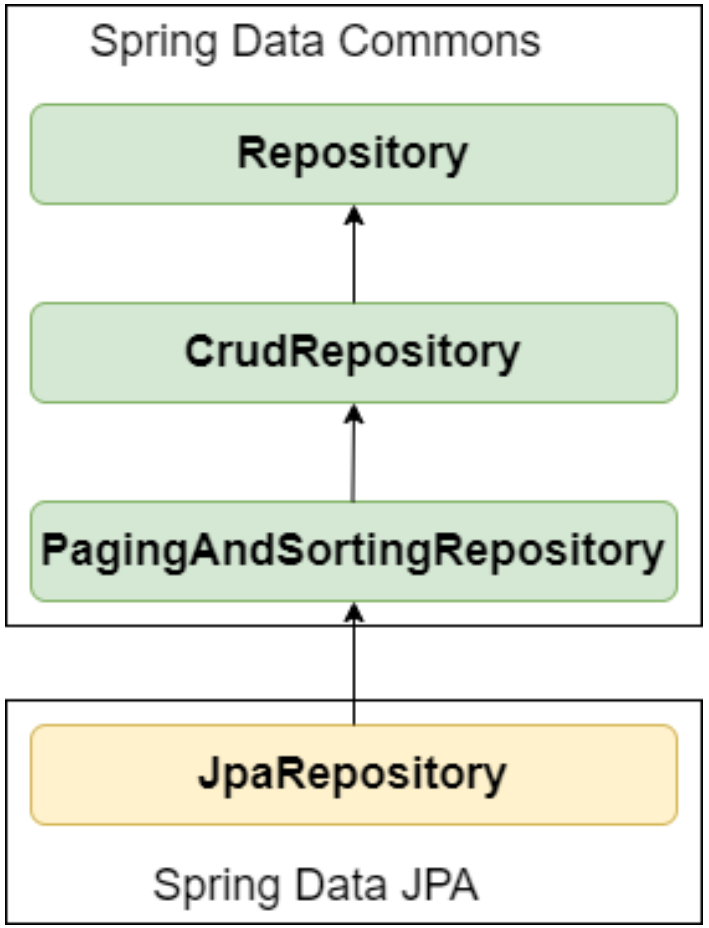
Spring Data Commons :- It provides the infrastructure that is shared by the datastore specific Spring Data projects.

JPA Provider :- The JPA Provider implements the Java Persistence API.



Spring Data Repositories Interfaces

The power of Spring Data JPA lies in the repository abstraction that is provided by the Spring Data Commons project and extended by the datastore specific sub projects.
We can use Spring Data JPA without paying any attention to the actual implementation of the repository abstraction, but we have to be familiar with the Spring Data repository interfaces. These interfaces are described in the following:



Spring Data Repositories

Spring Data Commons provides the following repository interfaces:

- `Repository` — Central repository marker interface. Captures the domain type and the ID type.
- `CrudRepository` — Interface for generic CRUD operations on a repository for a specific type.
- `PagingAndSortingRepository` — Extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction.
- `QuerydslPredicateExecutor` — *Interface to allow execution of QueryDSL Predicate instances. It is not a repository interface.*

Spring Data Repositories

Spring Data JPA provides the following additional repository interfaces:

- `JpaRepository` — JPA specific extension of Repository interface. It combines all methods declared by the Spring Data Commons repository interfaces behind a single interface.
- `JpaRepository` — *It is not a repository interface. It allows the execution of Specifications based on the JPA criteria API.*

Working with Spring Data Repositories

The central interface in the Spring Data repository abstraction is Repository. It takes the domain class to manage as well as the identifier type of the domain class as type arguments.

This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The CrudRepository and ListCrudRepository interfaces provide sophisticated CRUD functionality for the entity class that is being managed.

```java
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);      ❶

  Optional<T> findById(ID primaryKey); ❷

  Iterable<T> findAll();               ❸

  long count();                        ❹

  void delete(T entity);               ❺

  boolean existsById(ID primaryKey);   ❻

  // … more functionality omitted.
}
```

Setting up your project components

The JPA Provider implements the Java Persistence API.

- Hibernate (default JPA implementation provider)
- EclipseLink (you need config yourself)

Spring Data JPA hides the used JPA provider behind its repository abstraction.

Configure the DataSource

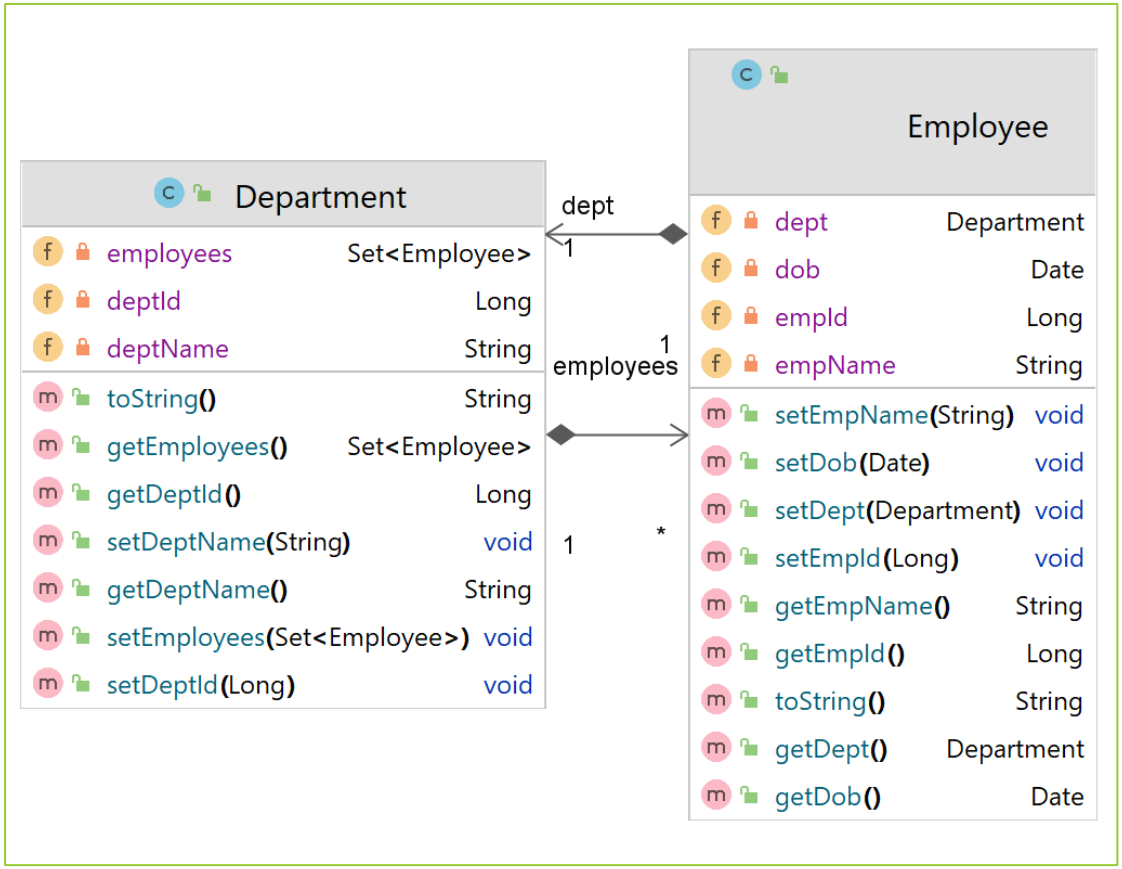
application.properties

```
spring.application.name=employee-mariadb-jpa
server.port=8084

spring.datasource.url=jdbc:mariadb://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.database-platform=org.hibernate.dialect.MariaDBDialect

```

Entities

```java
@Entity  14 usages  new *
@Table(name = "employees")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String role;
```

Entity sample



Repositories

Use repository for:

- Persist, update and remove one or multiple entities.
- Find one or more entities by their primary keys.
- Count, get, and remove all entities.
- Check if an entity exists with a given primary key.
- …

You do not need to write a boilerplate code for CRUD methods.

You should specify packages where container finding the repositories.

Repositories – Query creation

Auto-Generated Queries: Spring Data JPA can auto-generate database queries based on method names.
Auto-generated queries may not be well-suited for complex use cases. But for simple scenarios, these queries are valuable.
Parsing query method names is divided into subject and predicate.

- The first part (find…By…, exists…By…) defines the subject of the query,
- The second part forms the predicate
  - Distinct, And, Or, Between, LessThan, GreaterThan, Like, IgnoreCase, OrderBy (with Asc, Desc)

Repositories – Query creation

1. Method Name: *https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html#appendix.query.method.subject*
2. JPQL: Java Persistence Query Language
   - By default, the query definition uses JPQL.
3. Native

More: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation

```java
//  cách 1: get employee có salary lớn hơn {value}
List<Employee> findBySalaryGreaterThan(Double salary);  no usages   new *

// Cách 2: JPQL query
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")  no usages   new *
List<Employee> findEmployeesWithSalaryGreaterThan(@Param("salary") Double salary);

// Cách 3: Native SQL
@Query(value = "SELECT * FROM employees WHERE salary > :salary", nativeQuery = true)  1 usage   new *
List<Employee> findEmployeesWithSalaryGreaterThanNative(@Param("salary") Double salary);
```

Service

Mapping method name with Repository - inject repository

```
//1:
public List<Employee> getByDepartmentId(int departmentId) {  no usages   new *
    return repo.findByDepartment_Id(departmentId);
}

//2:
public List<Employee> findEmployeesWithSalaryGreaterThan(Double salary) {  no usa
    return repo.findEmployeesWithSalaryGreaterThan(salary);
}

//3:
public List<Employee> findEmployeesWithSalaryGreaterThanNative(Double salary) {
    return repo.findEmployeesWithSalaryGreaterThanNative(salary);

}
```

Controller

```
@GetMapping("/salary/{value}")  new *
public List<Employee> getBySalaryGreaterThan(@PathVariable Double value) {
    //1.
    //return service.getBySalaryGreaterThan(value);
    //2.
    //return service.findEmployeesWithSalaryGreaterThan(value);
    //3.Native SQL
    return  service.findEmployeesWithSalaryGreaterThanNative(value);
}
```

Queries – Pagination and Sorting

Pagination is often helpful when we have a large dataset, and we want to present it to the user in smaller chunks.
Also, we often need to sort that data by some criteria while paging.

repository

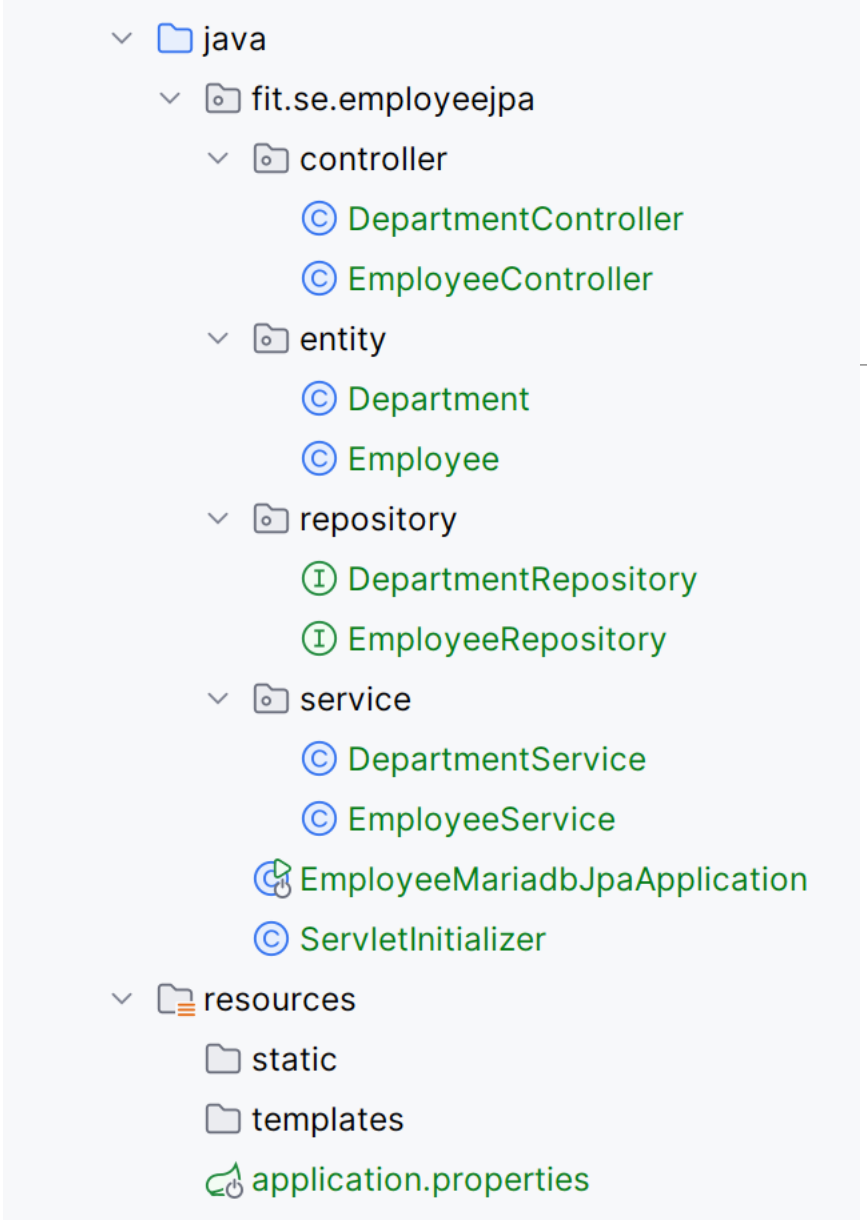
```
// phân trang employee
Page<Employee> findAll(Pageable pageable);  new *

// nếu muốn phân trang theo department
Page<Employee> findByDepartment_Id(Integer departmentId, Pageable pageable);
```

service

```
// Lấy danh sách Employee theo trang
public Page<Employee> getEmployees(int pageNumber, int pageSize) {  1 usage   new *
    Pageable pageable = PageRequest.of(pageNumber, pageSize, Sort.by( ...properties: "id").ascending());
    return repo.findAll(pageable);
}
public Page<Employee> getByDepartmentId(int departmentId, int pageNumber, int pageSize) {  1 usage   new *
    Sort sort = Sort.by( ...properties: "id").ascending();
    Pageable pageable = PageRequest.of(pageNumber, pageSize, sort);
    return repo.findByDepartment_Id(departmentId, pageable);

}
```

controller

```
// GET /employees?page=0&size=10
@GetMapping("/page")  new *
public Page<Employee> getEmployees(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size) {
    return service.getEmployees(page, size);
}
@GetMapping("/pagedepartId")  new *
public Page<Employee> getByDepartmentId(int deptId,
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "3") int size){
    return service.getByDepartmentId(deptId, page, size);
}
```



Example

1. Tạo Employee, Department
2. Repository:

- Tìm tất cả nhân viên
- Tìm nhân viên theo mã
- Tìm nhân viên theo Tên
- Tìm danh sách nhân viên theo phòng ban
- Tìm nhân viên theo mức lương

Spring Data JDBC

Spring Data JDBC makes it easy to implement JDBC based repositories.

- This module deals with enhanced support for JDBC based data access layers.
- It makes it easier to build Spring powered applications that use data access technologies.

Spring Data JDBC aims at being conceptually easy.

- It does NOT offer caching, lazy loading, write behind or many other features of JPA.
- This makes Spring Data JDBC a simple, limited, opinionated ORM.

https://spring.io/projects/spring-data-jdbc

Spring Data JPA vs. Spring Data JDBC

Spring Data JDBC is very similar to Spring Data JPA in terms of API.

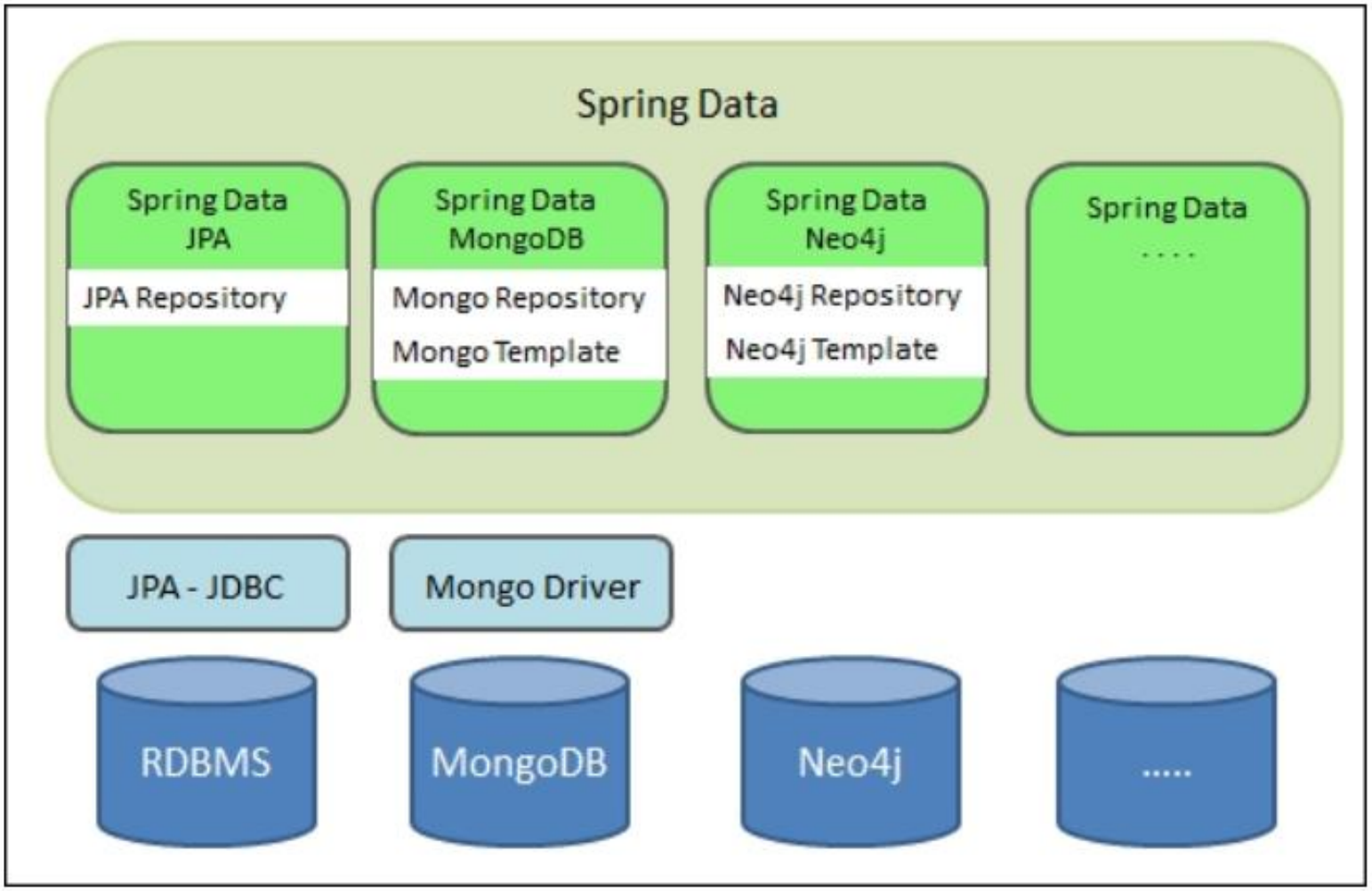
Both of them use Spring Data Commons as a base library.

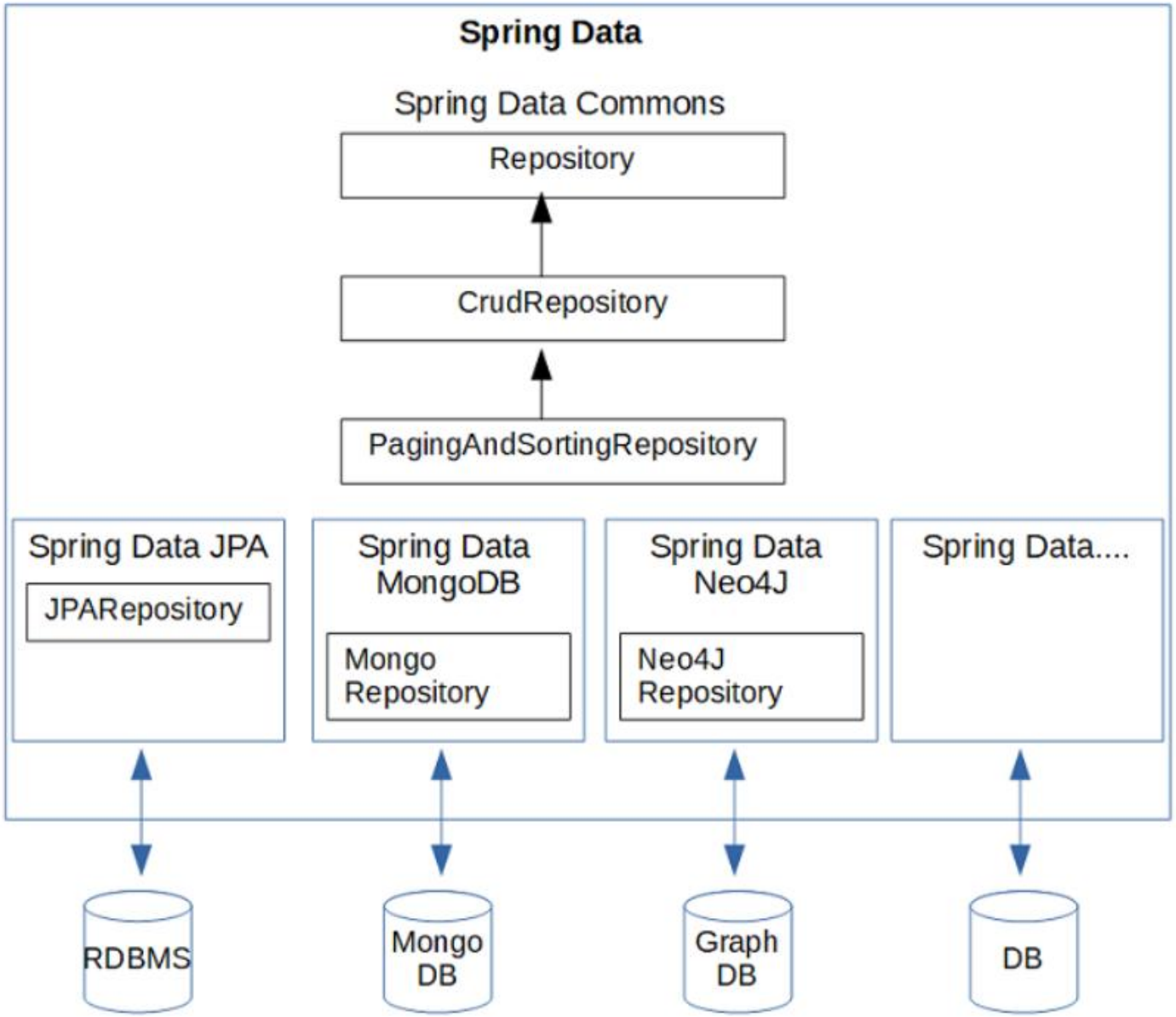
- In Spring Data JPA, repositories allow us to use derived methods instead of queries. Such methods transparently convert their invocations to queries using a JPA implementation.
- In Spring Data JDBC, method invocations are transformed into pure SQL and executed via JDBC.

MongoDB Model Implementation

MongoDB and Spring Boot can be interacted by using the MongoTemplate class and MongoRepository interface.

- `MongoRepository` — MongoRepository is used for basic queries that involve all or many fields of the document. Examples include data creation, viewing documents, and more.
- `MongoTemplate` — MongoTemplate implements a set of ready-to-use APIs. A good choice for operations like update, aggregations, and others, MongoTemplate offers finer control over custom queries.


Spring Data
Spring Data JPA
JPA Repository
Spring Data MongoDB
Mongo Repository
Mongo Template
Spring Data Neo4j
Neo4j Repository
Neo4j Template
Spring Data
. . . .
JPA - JDBC
Mongo Driver
RDBMS
MongoDB
Neo4j
.....


Spring Data
Spring Data Commons
Repository
CrudRepository
PagingAndSortingRepository
Spring Data JPA
JPARepository
Spring Data MongoDB
Mongo Repository
Spring Data Neo4J
Neo4J Repository
Spring Data....
RDBMS
Mongo DB
Graph DB
DB

MongoRepository

- extending CrudRepository & PagingAndSortingRepository
- Generates CRUD methods from method names

CRUD Repository

- `CrudRepository` : This interface provides the fundamental methods for interacting with a database, including:
  - `save(S entity)` : Saves a given entity.
  - `findById(ID id)` : Retrieves an entity by its ID.
  - `existsById(ID id)` : Checks if an entity with the given ID exists.
  - `findAll()` : Returns all instances of the type.
  - `count()` : Returns the number of entities available.
  - `deleteById(ID id)` : Deletes the entity with the given ID.
  - `delete(T entity)` : Deletes a given entity.

PagingAnhSortingRepository

`PagingAndSortingRepository`: This interface extends `CrudRepository` and adds methods specifically for handling pagination and sorting of data:

- `findAll(Sort sort)`: Returns all entities sorted by the given options.
- `findAll(Pageable pageable)`: Returns a `Page` of entities according to the pagination and sorting information in the `Pageable` object.

Demo

entity

```
@Document(collection = "employees")  25 usages  new *
public class Employee {

    @Id
    @JsonIgnore   // ẩn không trả về cho client do
    private String id;// Mongodb generate _id

    private String empId;  3 usages
    private String empName;  3 usages
    private String email;  3 usages
    private int age;  3 usages
    private int status;  3 usages
```

repository

```
@Repository  3 usages  new *
public interface EmployeeRepository extends MongoRepository<Employee, String> {

    Employee findByEmpId(String empId);  2 usages  new *

    List<Employee> findByEmpNameContaining(String name);  1 usage  new *

    List<Employee> findByStatus(int status);  1 usage  new *

    void deleteByEmpId(String empId);  1 usage  new *

    boolean existsByEmpId(String empId);  1 usage  new *
//    existsByEmpId
}
```

repository

```
Employee findByEmpId(String empId);  2 usages   new *
```

service

```
public Employee getEmployeesByempId(String empId) {  1 usage   new *
    return repo.findByEmpId(empId);
}
```

controller

```
@GetMapping("/{empId}")  new *
public Employee getEmployeeByEmpId(@PathVariable String empId) {
    return service.getEmployeesByempId(empId);
}
```

Paging and Sorting

repository : **MongoRepository**

service

```
public Page<Employee> getEmployeesWithPaginationAndSorting(int page, int size, String sortBy,
                                                           String direction) {
    Sort sort = direction.equalsIgnoreCase( anotherString: "desc") ?
            Sort.by(sortBy).descending() :
            Sort.by(sortBy).ascending();

    Pageable pageable = PageRequest.of(page, size, sort);
    return repo.findAll(pageable);
}
```

Paging and Sorting

controller

```
@GetMapping("/paging")  new *
public Page<Employee> getEmployeesPaging(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "3") int size,
        @RequestParam(defaultValue = "empId") String sortBy,
        @RequestParam(defaultValue = "asc") String direction) {

    return service.getEmployeesWithPaginationAndSorting(page, size, sortBy, direction);
}
```

Custom Queries with @Query

Use @Query annotation for complex queries:

```
// Tìm theo tuổi lớn hơn age
@Query("{ 'age': { $gt: ?0 } }")  no usages  new *
List<Employee> findByAgeGreaterThan(int age);

// Tìm theo email chính xác
@Query("{ 'email': ?0 }")  no usages  new *
Employee findByEmail(String email);

// Tìm theo empId & status
@Query("{ 'empId': ?0, 'status': ?1 }")  no usages  new *
Employee findByEmpIdAndStatus(String empId, int status);
```

MongoRepository

- Simple CRUD, basic queries
- Want concise, clean code
- Focus on business logic

Limitations:

- Less flexible than MongoTemplate
- Limited aggregation control

MongoTemplate

A central class within Spring Data MongoDB, providing a powerful and flexible way to interact with MongoDB database

Key Features and Usage:

Key Features

- CRUD operations: `MongoTemplate` provides methods for common CRUD (Create, Read, Update, Delete) operations, including `insert`, `save`, `find`, `findById`, `remove`, and `update`.
- Querying with Criteria & Query: construct complex queries using `Query` and `Criteria` objects, enabling you to specify conditions, projections, sorting, and pagination.

- Aggregation pipelines: supports MongoDB's aggregation framework, allowing you to perform advanced data processing and analysis using aggregation pipelines.
- Collection & Index management: provides methods for managing indexes and collections within your MongoDB database Callback methods & lifecycle hooks

MongoTemplate - CRUD Operations

CRUD operations: insert, save, find, findById, remove, update action

Example:

```
mongoTemplate.save(employee, "employees");
Employee emp = mongoTemplate.findById("E01",
Employee.class);
```

Querying with Criteria

Build flexible queries:

```
Query query = new Query();
query.addCriteria(Criteria.where("salary").gt(5000));
List<Employee> list = mongoTemplate.find(query,
Employee.class);
```

Aggregation Framework

Complex data analysis with pipelines

Example: group by deptId and compute avg salary

```
Aggregation agg = newAggregation(
        group("deptId")
                .count().as("count")
                .avg("salary").as("avgSalary"),
        project("count", "avgSalary")
                .and("_id").as("deptId")
);
```

Collection & Index Management

Create, drop, check collections & indexes

Example:

```
mongoTemplate.indexOps(Employee.class)
                .ensureIndex(new Index().on("empId", ASC));
```

Connect Server

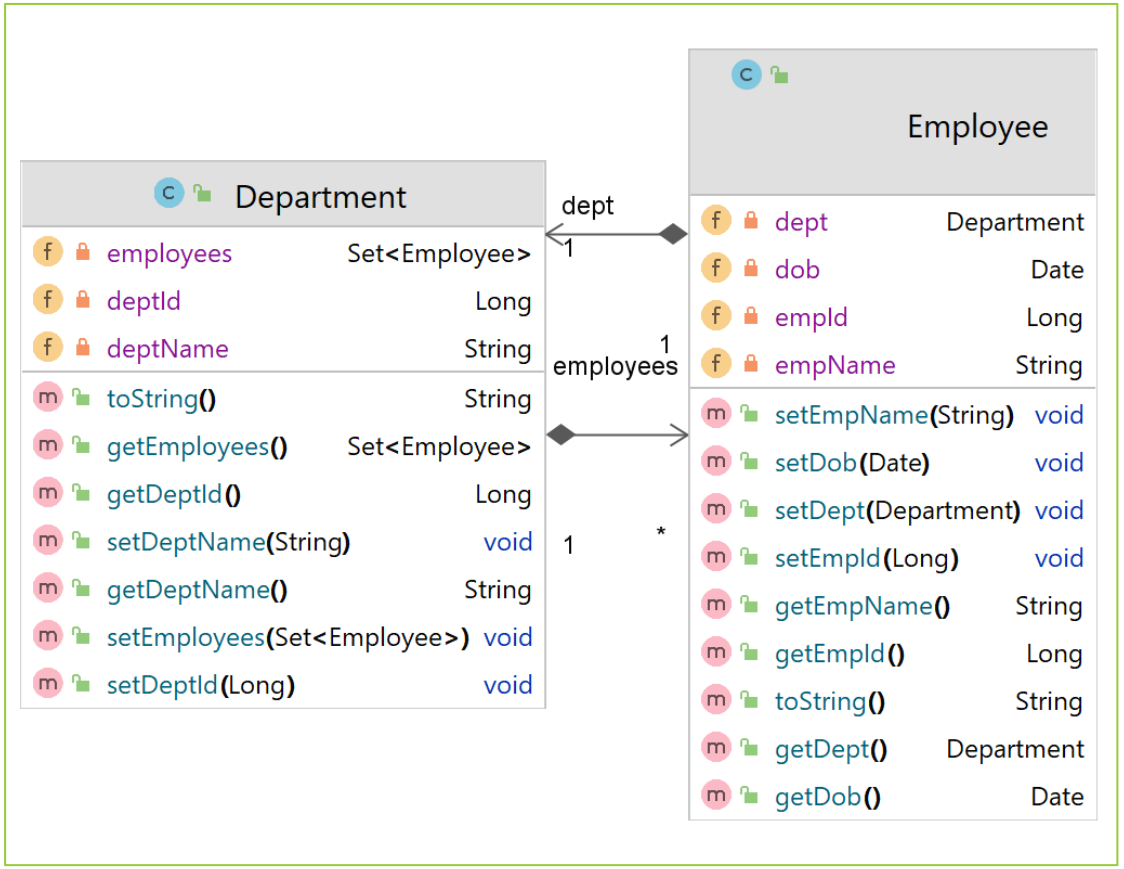
```
spring.application.name=employee-mongodb

server.port=8081

# MongoDB config
spring.data.mongodb.uri=mongodb://localhost:27017/employeewww
spring.data.mongodb.database=employeewww
```

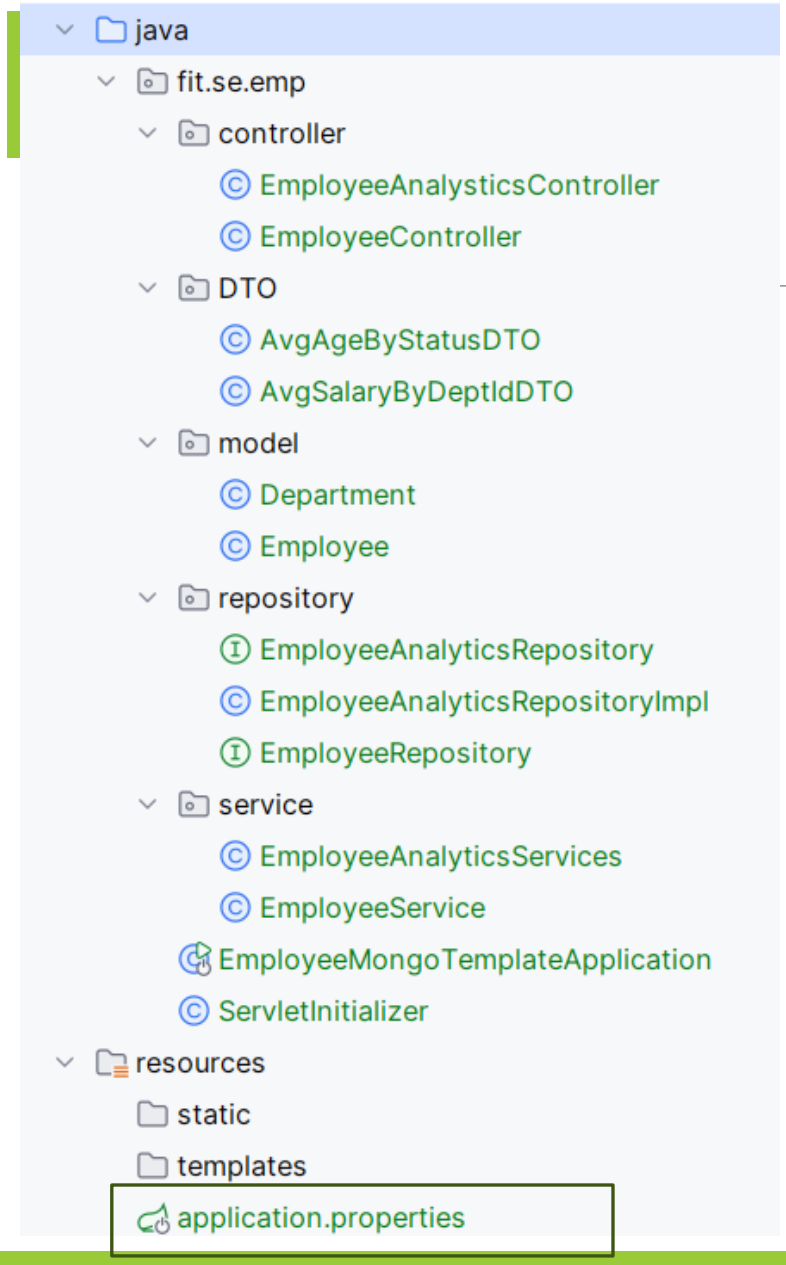
Entities Documentation

```
{
  _id: ObjectId("68c73b4e57c86db9eacd1628"),
  deptId: 'D001',
  deptName: 'IT',
  employees: [
    {
      empId: 'E001',
      empName: 'Alice',
      email: 'alice@example.com',
      age: 25,
      status: 1
    },
    {
      empId: 'E002',
      empName: 'Bob',
      email: 'bob@example.com',
      age: 30,
      status: 1
    }
  ]
},
```

```
{
  _id: ObjectId("68c73b3e57c86db9eacd1625"),
  empId: 'E001',
  empName: 'Alice Packer',
  email: 'Alicepaker@gmail.com',
  age: 20,
  status: 1,
  deptId: 'D002',
  salary: 1000
},
```

Entity sample





Example structure

New Project Spring Boot – Add dependencies
Step 0: Config Connection and Server
`application.properties`
Step 1: Entities Document
Step 2: Repository
- EmployeeRepository: MongoRepository
CRUD Method (Basic Query)
- EmployeeAnalyticsRepository: MongoTemplate
(Complex Query - Aggregate)
Step 3: Service
Step 4: Controller
Step 5: (Optional) DTO define class for result of MongoTemplate, that sechema is different from entites documents.
Depend on the

- CRUD Employees
- Hiển thị thông tin của các employees có salary cao nhất
- Hiển thị thông tin của các employees có Age cao nhất
- Xuất ra thông tin số employee và mức salary trung bình từng department
- Xuất ra thông tin số employee và mức salary trung bình từng loại status
- Xuất ra danh sách employee theo mức lương tăng dần theo từng department.

Thymeleaf

THYMELEAF WITH SPRING

Content

1. Introduction
2. Configuration
3. Standard Expression Syntax
4. Attribute Values
5. Iterated expression
6. Conditional Evaluation
7. Template Layout
8. Forms

1. Introduction

Thymeleaf as Templating Engine

Thymeleaf is a modern server-side Java template engine for both web and standalone environments, capable of processing HTML, XML, JavaScript, CSS.

Integration with Spring Controllers

- Mapped methods in Spring @Controller classes forward requests directly to Thymeleaf templates for rendering.

Introduction

Support for Spring Expression Language

- Thymeleaf supports SpEL, offering robust and intuitive data binding within templates

Compatibility with Spring Boot

- Full compatibility with Spring Boot simplifies configuration and speeds up development workflows.

Form Creation and Binding

- Thymeleaf enables seamless form creation and binding with Spring’s form-backing beans for efficient data handling.

Introduction

Validation and Conversion

- Built-in validation and conversion services ensure accurate user input processing and meaningful error feedback.

Resource Resolution

- Integration with Spring’s resource resolution allows efficient template location and rendering within standard paths.

Introduction -The Standard Dialect

Thymeleaf's core library provides a dialect called the **Standard Dialect**, which should be enough for most users.

Most of the processors of the Standard Dialect are *attribute processors*.

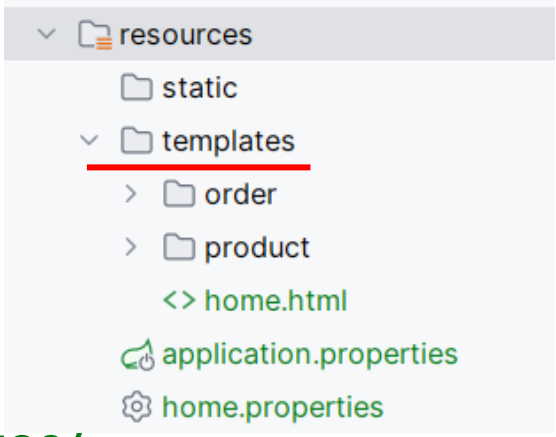
```
<input type ="text" name="userName"
value="${user.name}" />
```

```
<input type="text" name="userName" value="James
Carrot" th:value="${user.name}" />
```

2. Configuration

pom.xml

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



application.properties

```
#thymeleaf Config
spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

3. Standard Expression Syntax

1. Text: <th:text=" ">

Message Expressions: #{...}

Get from #####.properties

2. Variable: <th:text=" ">

Variable Expressions: ${...}

Get from Controller in model.addAtribute()

3. Link: <th:href=" ">

Link URL Expressions: @{...}

Get from RequestMapping/GetMapping of Controller

```
<img src="../../images/gtvglogo.png" alt="GTVG logo" title="GTVG logo"
     th:src="@{/images/gtvglogo.png}"/>

<h1 th:text="${message}">mess</h1>

<h2 th:text="#{home.welcome}">welcome</h2>

<p th:text="${date}"></p>
<p>Please select an option</p>
<ol>
    <li><a href="product/list.html" th:href="@{/product}">Product List</a></li>
    <li><a href="order/list.html" th:href="@{/order}">Order List</a></li>
</ol>
```

```java
@Controller
@RequestMapping("/home")
public class HomeController {

    public HomeController() {
        super();
    }
    @GetMapping
    public String HomePage(Model model) {
        LocalDate date = LocalDate.now();
        String mess ="Welcome Thymeleaf";
        model.addAttribute("message", mess);
        model.addAttribute("date", date.toString());
        return "home";
    }
}
```

4. *Attribute Values*

```
<img src="../../images/shopping.png"
th:src="@{/images/shopping.png}"/>
```

```
<form action="subscribe.html" th:action="@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!"
th:value="#{subscribe.submit}"/>
  </fieldset>
</form>
```

5.Iterated expression

Thymeleaf **iteration** allows you to loop through collections like lists, arrays, or maps and display each item in your HTML template.

```
model.addAttribute( attributeName: "products", productlist);
return "product/list";
```

```
<h1>List Product</h1>
<div th:each="product : ${products}" class="product">
   <a href="product/productdetail.html" th:href="@{'/product/' +
${product.id}}">
      <input type="hidden" th:value="${product.id}"
th:name="productId">
      <h2 th:text="${product.name}">Product Name</h2>
   </a>
</div>
```

6.Conditionals

If-then: (if) ? (then)

If-then-else: (if) ? (then) : (else)

Default (Elvis): (value) ?: (defaultvalue)

No-Operation: _

```
<span th:text="${user.isAdmin()} ? 'Admin Panel'">Default</span>
<span th:text="${user.isAdmin()} ? 'Admin' : 'User'">Role</span>
<span th:text="${user.name} ?: 'Guest'">Name</span>
<span th:if="${user == null} ? _ : ${user.name}">Name</span>
```

7.Template layout

Fragment:

Fragment Expressions: ~{...}

A **fragment** is a reusable piece of HTML code that you define in one template and use in other templates. This helps you avoid repeating code like headers, footers, or navigation bars.

Use a Fragment

- `th:insert` – inserts the **content inside** the fragment.
- `th:replace` – replaces the **entire element** with the fragment.
- `th:include` – includes the fragment **inside** the current element (less common).

templates/footer.html

```
<!DOCTYPE html>
 <html xmlns:th="http://www.thymeleaf.org">
<body>
    <div th:fragment="copy"> &copy; 2011 The Shopping
     </div>
</body>
</html>
```

Use a gragment

```
<body> ...
    <div th:insert="~{footer :: copy}">
    </div>
</body>
```

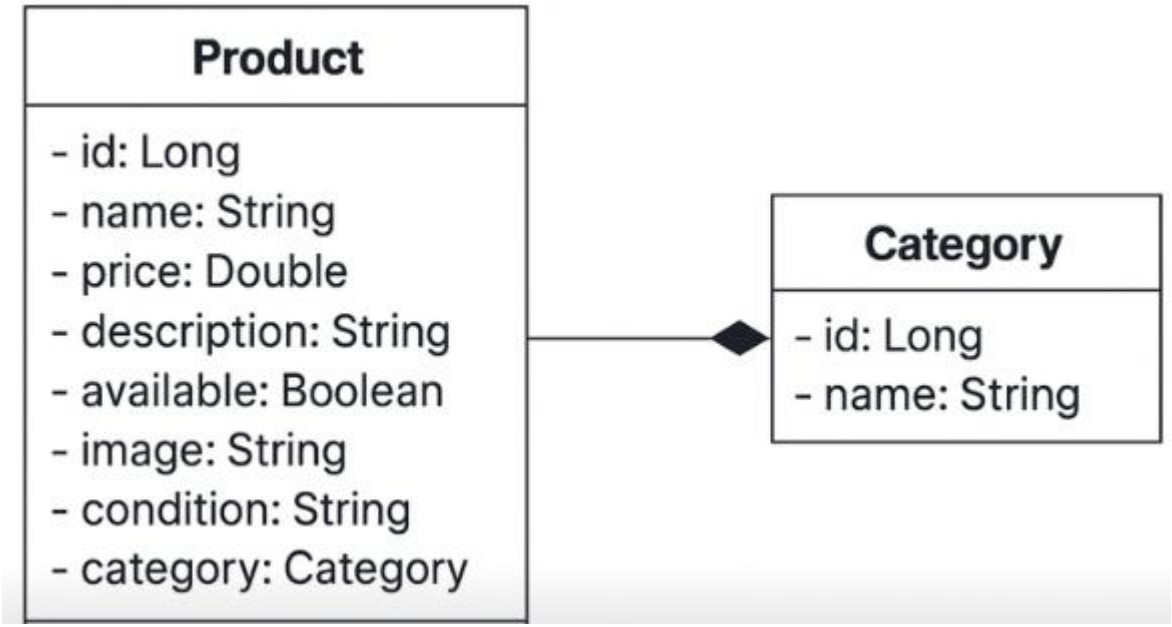
Thymeleaf with Forms

Form Declaration: In Thymeleaf, a form is bound to a **model object** (passed from the Controller)

```
<form th:action="@{/product/save}" th:object="${product}"
method="post" enctype="multipart/form-data">
    <!-- Inputs bound to Product fields -->
</form>
```

- `th:action` → specifies the URL the form submits to.
- `th:object="${product}"` → binds the form to the model object.
- `th:field="*{property}"` → binds an input field to a property of that object.

Element	Description	Syntax
Textbox	Input text value	`<input type="text" th:field="*{name}" />`
Password	Input hidden password value	`<input type="password" th:field="*{password}" />`
Checkbox	Boolean selection	`<input type="checkbox" th:field="*{available}" />`
Radio	Choose one among several options	`<input type="radio" th:field="*{condition}" value="New"/>`
Select (dropdown)	Choose from a list (e.g. categories)	`<select th:field="*{category}">...</select>`
File	Upload file (must use enctype="multipart/form-data")	`<input type="file" name="fileImage" />`


Product
- id: Long
- name: String
- price: Double
- description: String
- available: Boolean
- image: String
- condition: String
- category: Category
Category
- id: Long
- name: String

```
class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private Double price;
    private String description;
    private Boolean available;
    private String image; // stores file name
    @Column(name = "product_condition")
    private String condition;
}
```

```
@GetMapping("/add")
public String showAddForm(Model model) {
    model.addAttribute("product", new Product());
    model.addAttribute("categories",
categoryService.findAll());
    return "product/product_form";
}
```

```
@PostMapping("/save")
public String saveProduct(@ModelAttribute("product") Product
product,
                          @RequestParam("fileImage") MultipartFile file) throws
IOException {
    if (!file.isEmpty()) {
        String fileName = file.getOriginalFilename();
        Path uploadDir = Paths.get("uploads/");
        Files.createDirectories(uploadDir);
        file.transferTo(uploadDir.resolve(fileName));
        product.setImage(fileName);
    }
    productService.save(product);
    return "redirect:/product/list";
}
```

Inputs bound to object fields

```
<!-- Textbox -->
<label>Name:</label>
<input type="text" th:field="*{name}" placeholder="Product
name"/><br/>

<!-- Description -->
<label>Description:</label>
<input type="textarea" th:field="*{description}"
placeholder="Description"/><br/>

<!-- Number -->
<label>Price:</label>
<input type="number" th:field="*{price}" step="0.01"/><br/>
```

Inputs bound to object fields

```
<!-- Checkbox -->
<label>Available:</label>
<input type="checkbox" th:field="*{available}"/> In stock<br/>

<!-- Radio -->
<label>Condition:</label><br/>
<input type="radio" th:field="*{condition}" value="New"/>
New<br/>
<input type="radio" th:field="*{condition}" value="Used"/>
Used<br/>
```

Inputs bound to object fields

```
<!-- Select -->
<label>Category:</label>
<select th:field="*{category}" required>
   <option value="">--Select--</option>
   <option th:each="c : ${categories}"
         th:value="${c}"
         th:text="${c.name}"></option>
</select><br/>

<!-- File upload -->
<label>Image:</label>
<input type="file" name="fileImage"/><br/>
```

Demo:

1. Generate Spring Boot (Config database, thymeleaf)
2. Entity
3. Repository
4. Service
5. Controller
6. View: Thymleaf

Annotation Validation

What is Bean Validation?

- A standard for validating JavaBeans using annotations
- Based on Jakarta Validation (formerly JSR-380)
- Integrated with Spring Boot via Hibernate Validator
- Prevent invalid data from entering the system
- Improve user experience with clear error messages
- Reduce bugs and database inconsistencies

Common Validation Annotations

Dependenies:

```xml
<dependency>
    <groupId>jakarta.validation</groupId>
    <artifactId>jakarta.validation-api</artifactId>
    <version>3.0.2</version>
</dependency>

<!-- Nếu dùng Hibernate Validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>8.0.1.Final</version>
</dependency>
```

Annotation	Description
`@NotNull`	Field must not be `null`
`@NotEmpty`	Must not be `null` and must not be empty (for strings, collections, arrays)
`@NotBlank`	Must not be `null`, empty, or only whitespace (for strings)
`@Size(min=, max=)`	Validates the length of a string or size of a collection
`@Min(value)`	Minimum numeric value allowed

`@Max(value)`	Maximum numeric value allowed
`@Positive`	Value must be greater than 0
`@PositiveOrZero`	Value must be 0 or greater
`@Negative`	Value must be less than 0
`@NegativeOrZero`	Value must be 0 or less
`@Pattern(regexp="")`	Validates string against a regular expression
`@Email`	Validates email format

`@Past`	Date must be in the past
`@PastOrPresent`	Date must be in the past or today
`@Future`	Date must be in the future
`@FutureOrPresent`	Date must be today or in the future
`@AssertTrue`	Boolean must be `true`
`@AssertFalse`	Boolean must be `false`
`@Digits(integer=, fra`	Validates number format with integer and decimal precision

`@DecimalMin(value)`	Minimum decimal value
`@DecimalMax(value)`	Maximum decimal value
`@CreditCardNumber`	Validates credit card format (Hibernate Validator only)
`@Length(min=, max=`	Validates string length (Hibernate Validator only)

Setup validation in Entity

```
@Entity
Public class Product {
  @Id
  @GeneratedValue
  private Long id;
  @NotBlank(message = "Tên sản phẩm không được để trống")
  @Size(min = 2, message = "Tên phải có ít nhất 2 ký tự")
  private String name;
  @DecimalMin(value = "0.1", message = "Giá phải lớn hơn 0")
  private Double price;
  @Future(message = "Ngày nhập phải sau hiện tại")
  private LocalDate importDate;
  @Min(value = 1, message = "Số lượng phải lớn hơn 0")
  private int quantity;
}
```

```
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;
```

Controller Setup – Validation

```
@PostMapping("/save")
public String saveProduct(@Valid @ModelAttribute Product product,
                          BindingResult bindingResult,
                          @RequestParam("fileImage") MultipartFile file,
                          Model model
                         ) throws IOException {
if(bindingResult.hasErrors()) {
    model.addAttribute( attributeName: "categories", categoryService.findAll());
    return "product/productform";
}
...
}
```

- Captures validation errors **`BindingResult`**
- Must be placed immediately after **`@Valid`** parameter
- Used to check **`hasErrors()`** and return form if needed

Thymeleaf: View message errors

Tag	Purpose
th:field	Binds input to model field
th:errors	Shows validation message
#fields.hasErrors()	Checks if field has error

```
<!-- Textbox -->
<label>Name:</label>
<input class="form-control" type="text" th:field="*{name}" placeholder="Product name"/>
<div th:if="${#fields.hasErrors('name')}" th:errors="*{name}" class="alert alert-danger"> </div>
<br/>
```