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

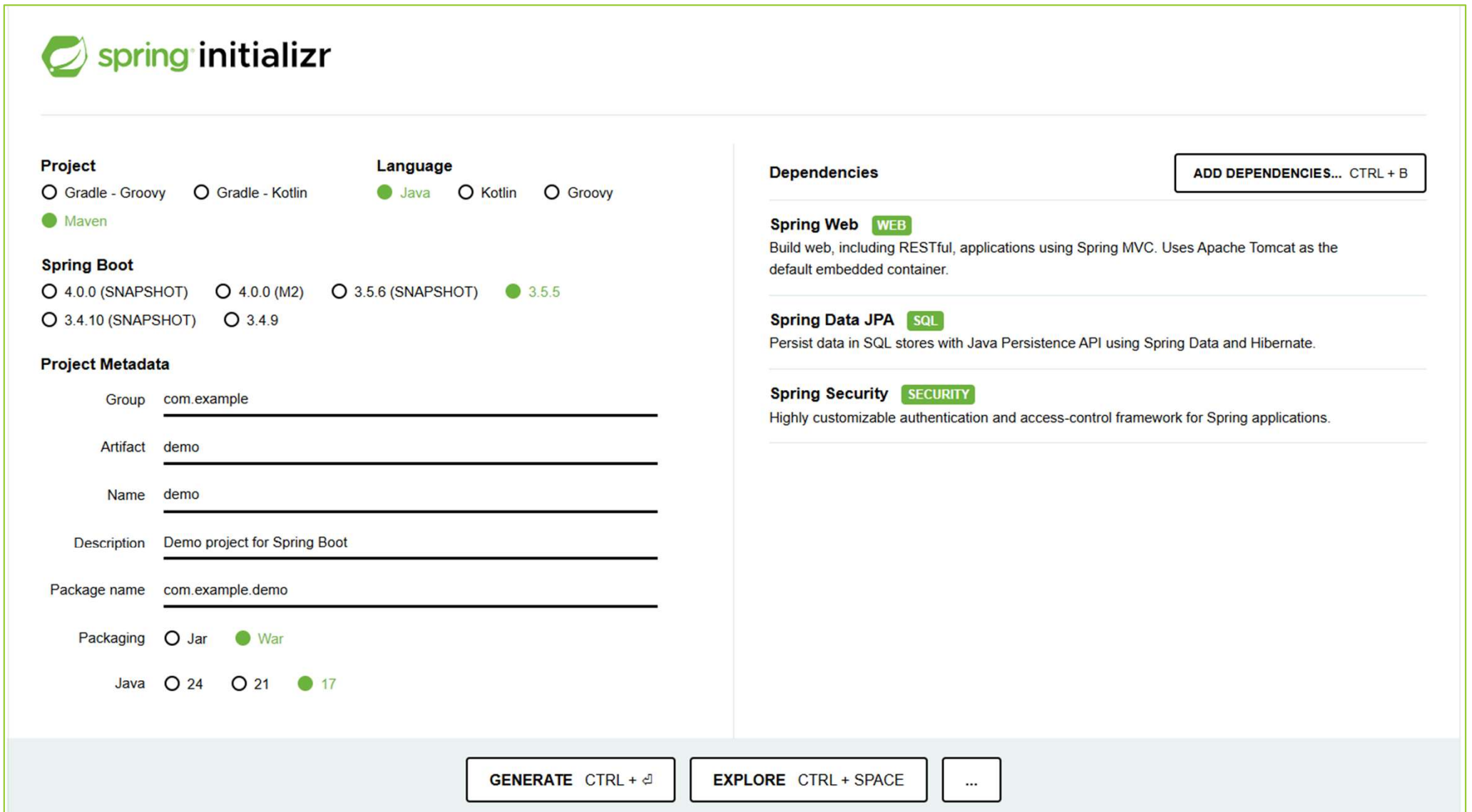
There are 3 main ways:

1. Using Spring Initializr Web (start.spring.io)

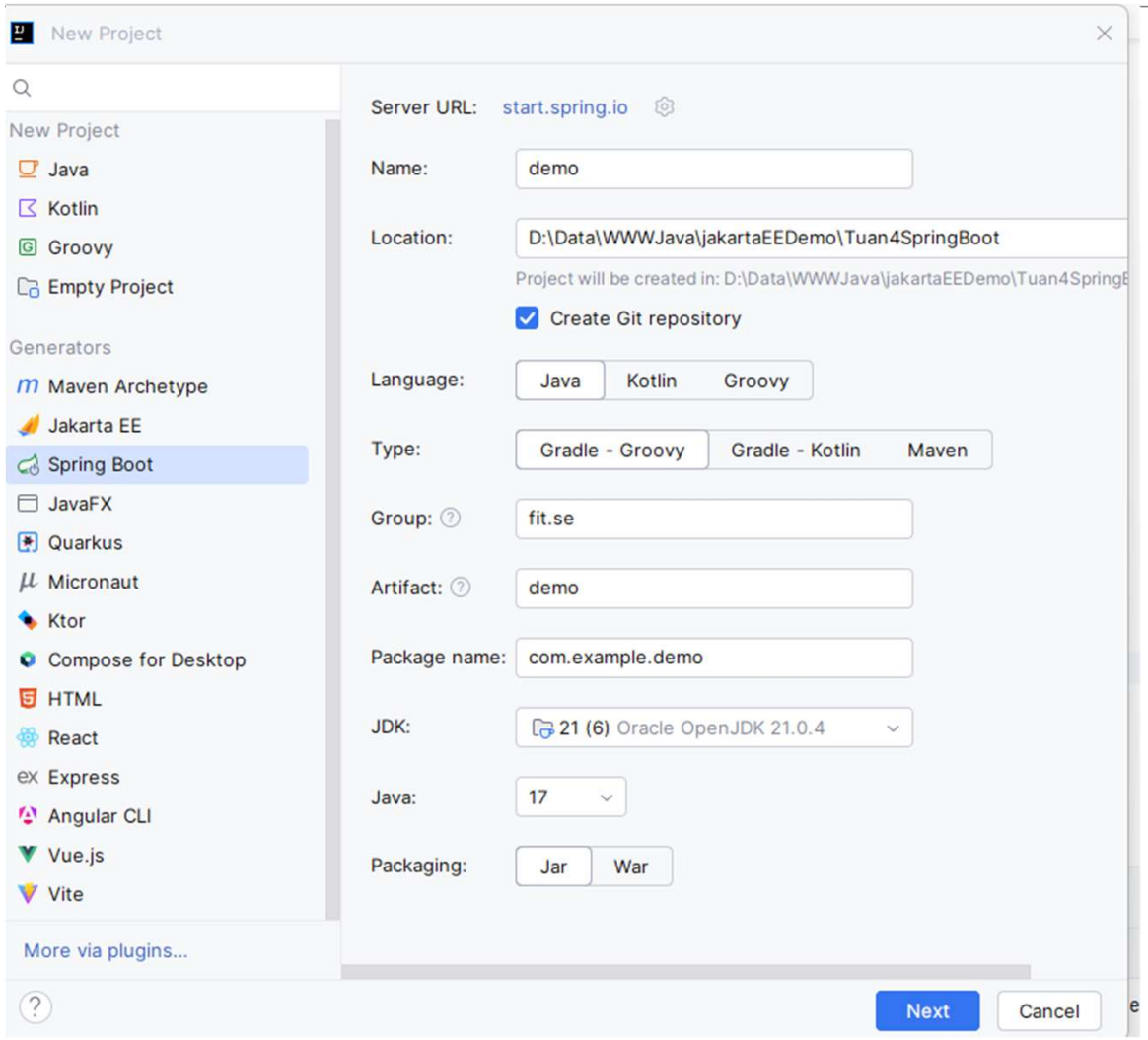
2. Using Spring Initializr directly in IntelliJ IDEA

3. Manually creating a Maven/Gradle project and adding Spring Boot

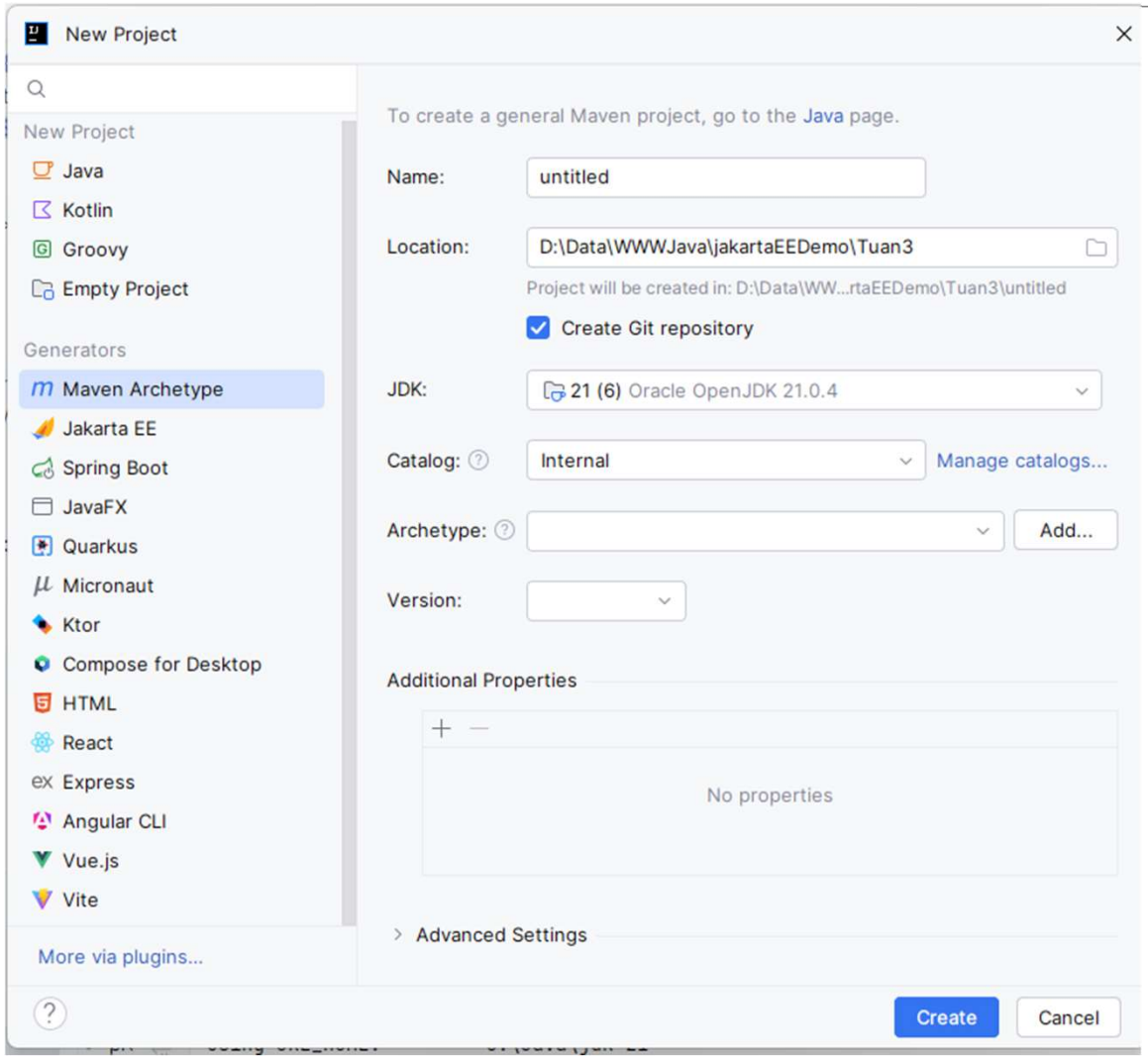
Spring initializer - start.spring.io



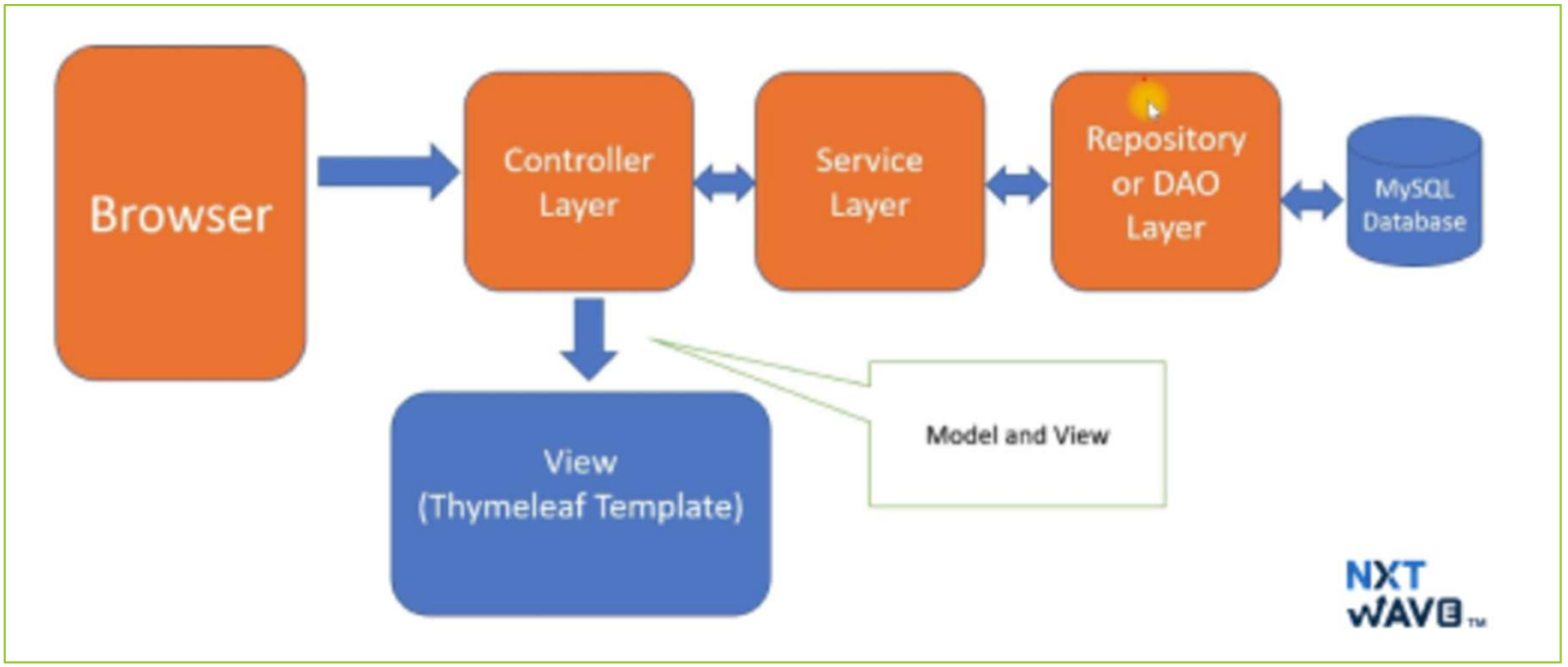
Spring initializer - IntelliJ



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

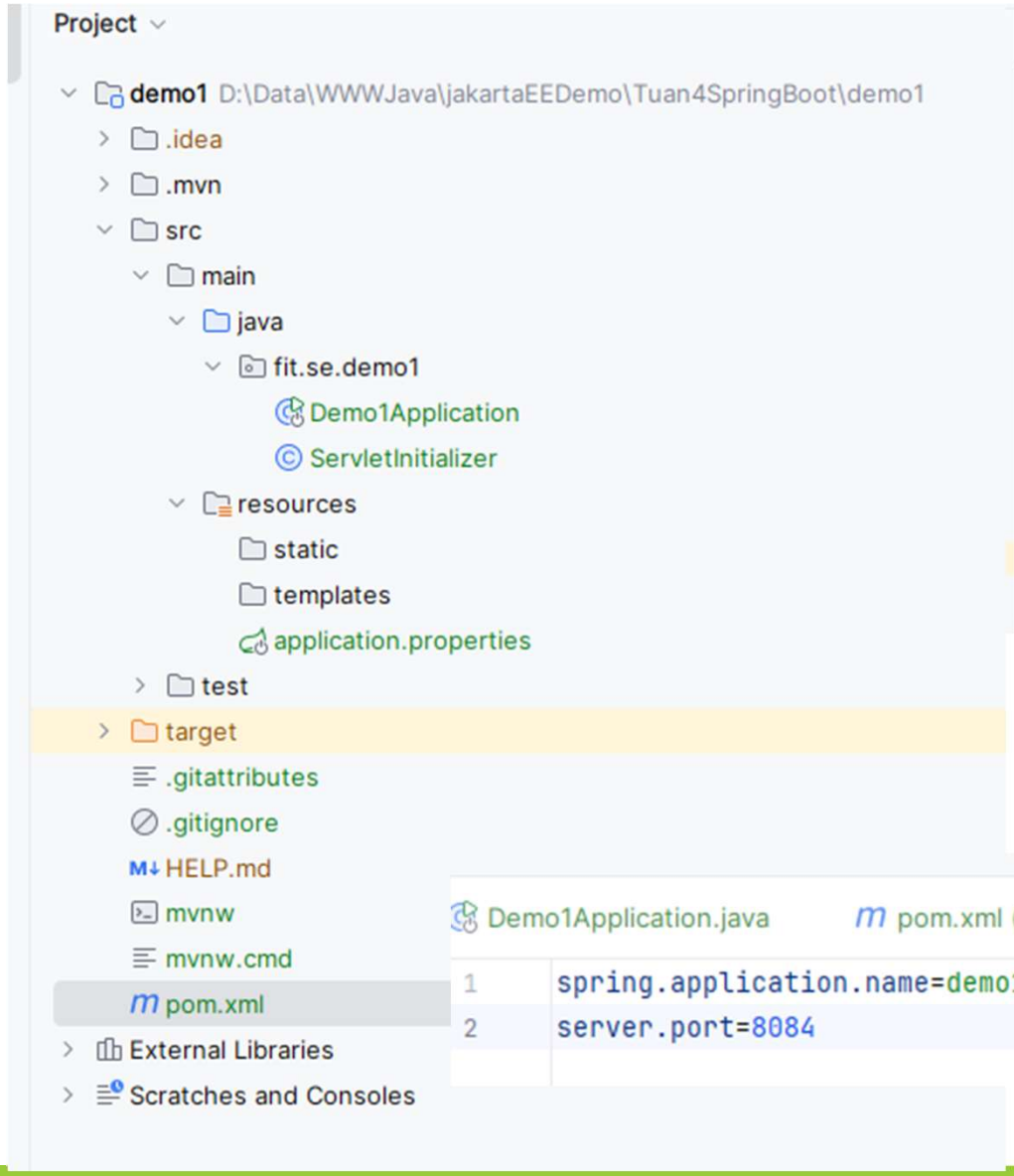
- Validation (JSR 380, Jakarta Validation API):

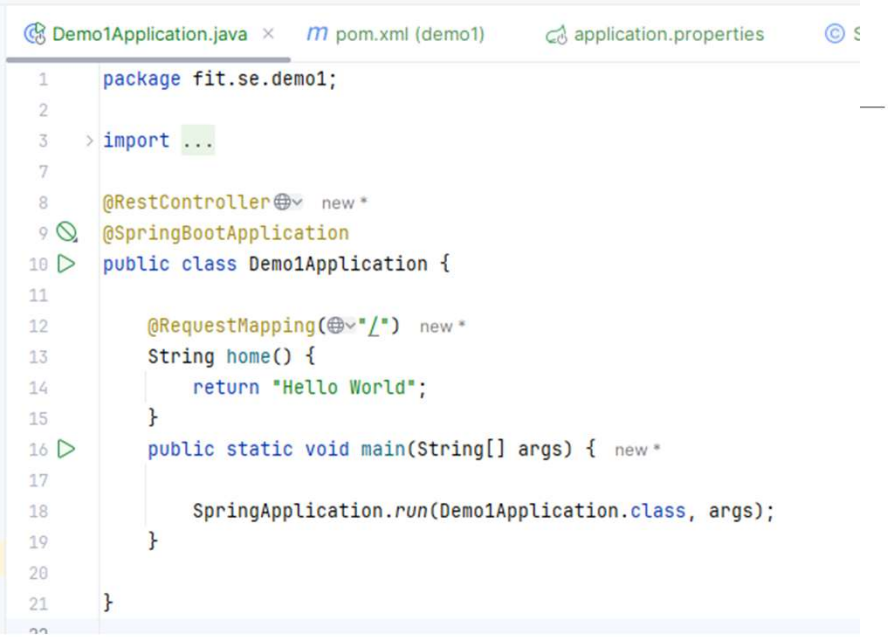
spring-boot-starter-validation

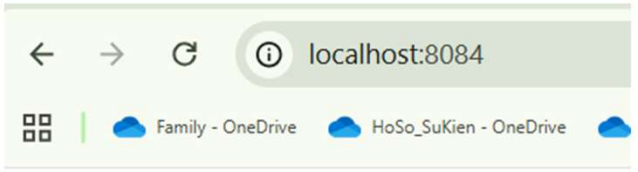
- Lombok (to reduce boilerplate): Lombok
- DevTools (auto restart during development): spring-boot-devtools

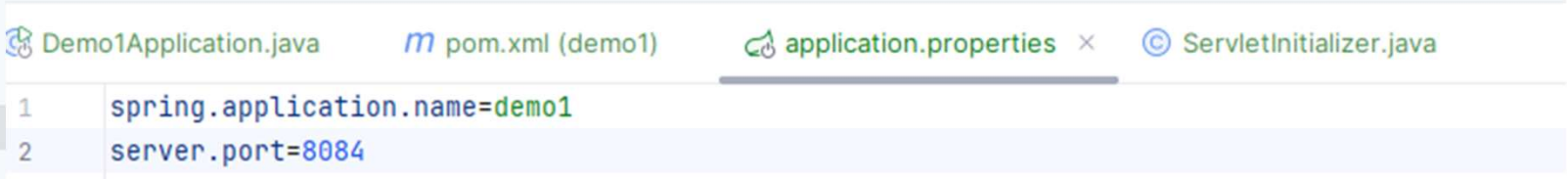
6. Testing: spring-boot-starter-test

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
```

src/main/resources/dbConfig.xml

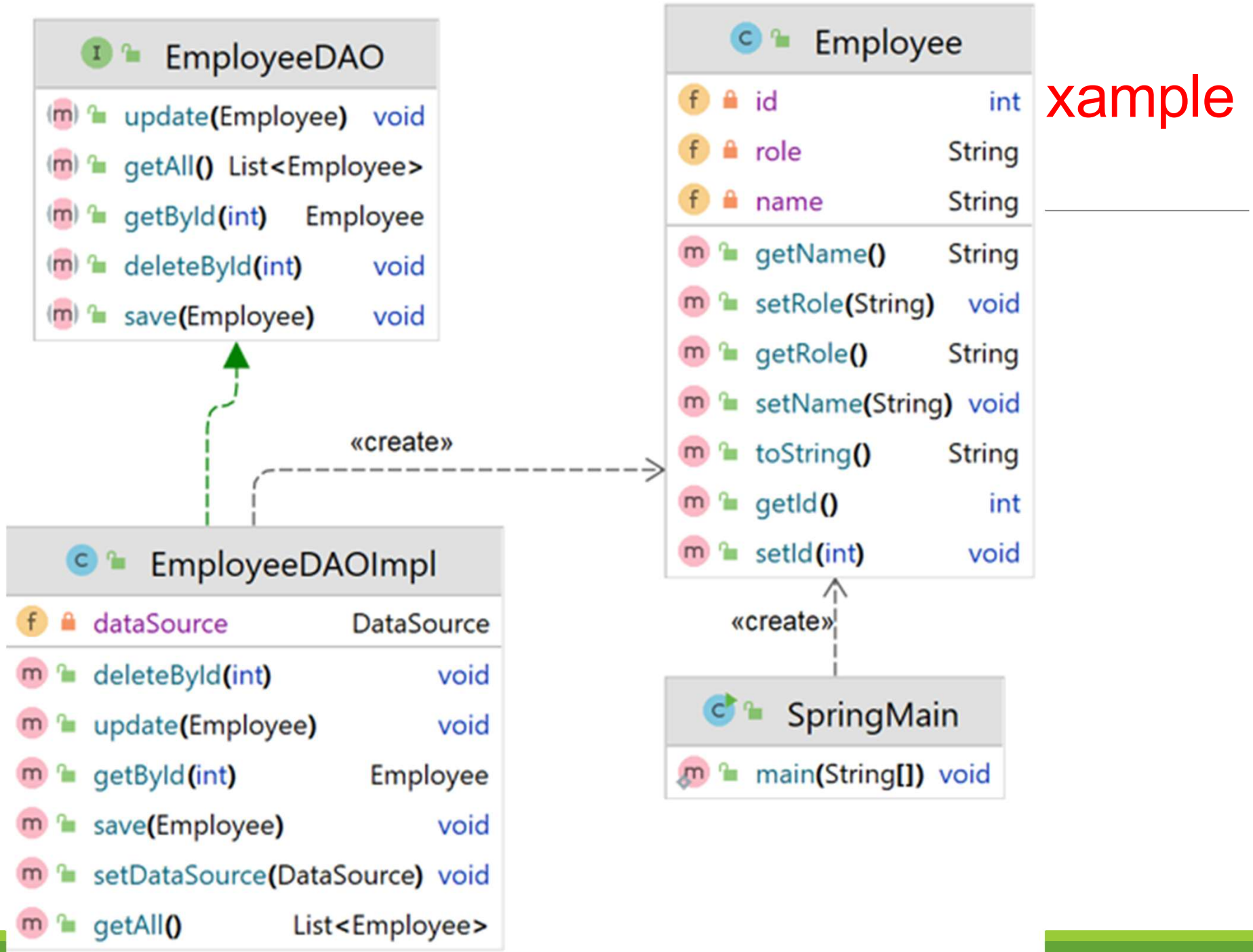
= ClassPathXmlApplicationContext("dbConfig.xml");

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
            scope="singleton">
    <property name="url" value="jdbc:mariadb://localhost:3306/sampledb" />
    <property name="driverClassName" value="org.mariadb.jdbc.Driver" />
    <property name="username" value="root" />
    <property name="password" value="password" />
</bean>
```

= new AnnotationConfigApplicationContext(DsConfig.class);

```java
@Configuration
public class DsConfig {
    no usages
    @Bean
    @Scope("singleton")
    public DataSource mariadbDataSource() throws Throwable{
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("org.mariadb.jdbc.Driver");
        ds.setUrl("jdbc:mariadb://localhost:3306/sampledb");
        ds.setUsername("root");
        ds.setPassword("password");
        return ds;
    }
}
```

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

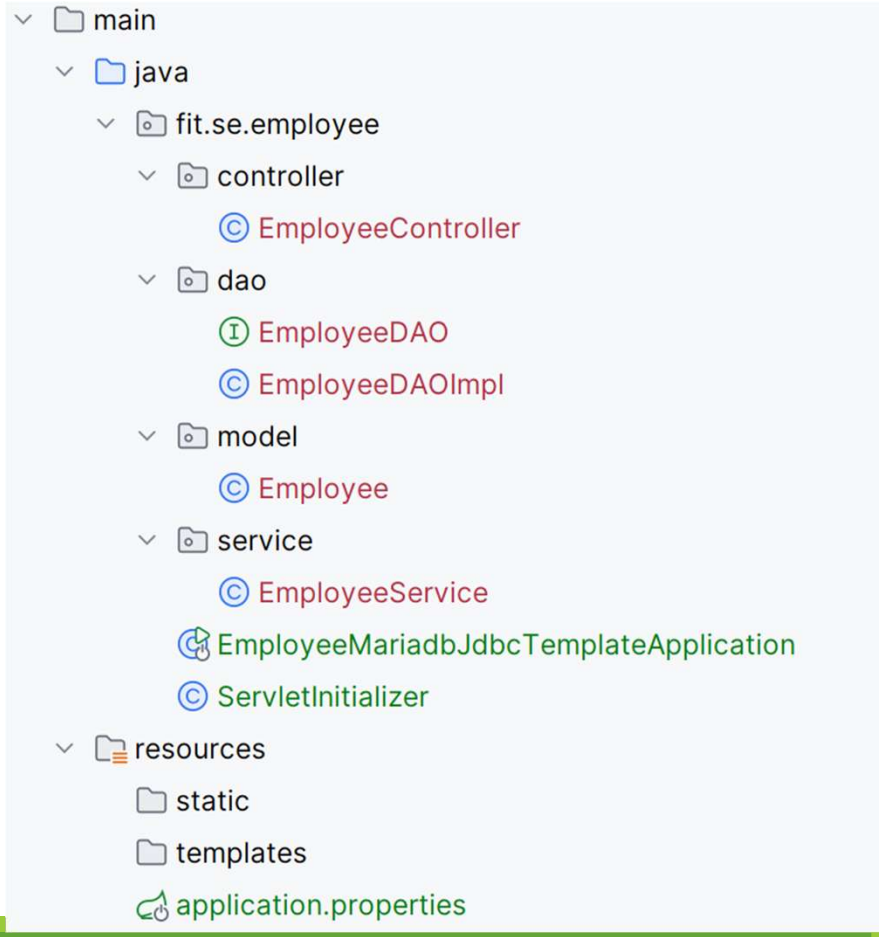
Step 0. Create Spring Boot Project

Dependencies to include:

Spring Web

Spring JDBC

MariaDB Driver



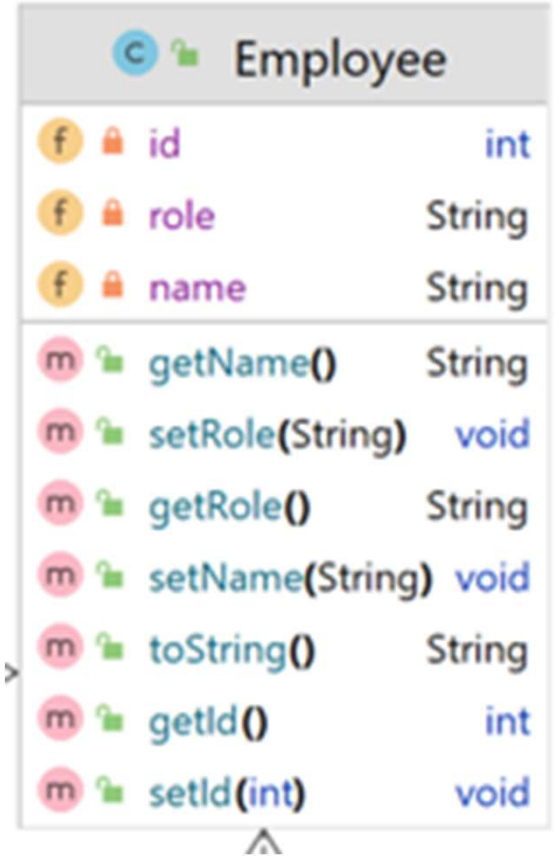
Step 1: Config Datasource

```
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
```

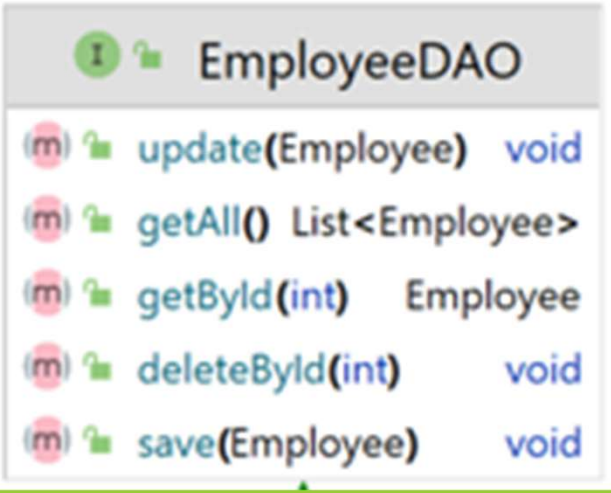
Controller → Service → EmployeeDAO (interface) → EmployeeDAOImpl (execute SQL) → Database

Step 2: Create Class Employee

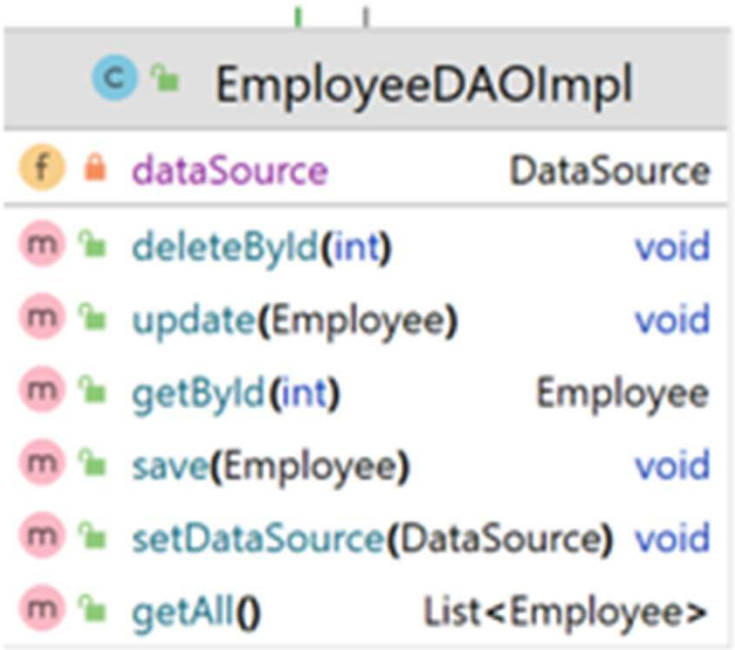


Step 3: Create ***interface*** EmployeeDAO

```
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 – Business logic component
- @Transactional – Manage transactions automatically
- @Async – Run method asynchronously
- @Cacheable, @CachePut, @CacheEvict – Enable caching

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

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

```java
@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.

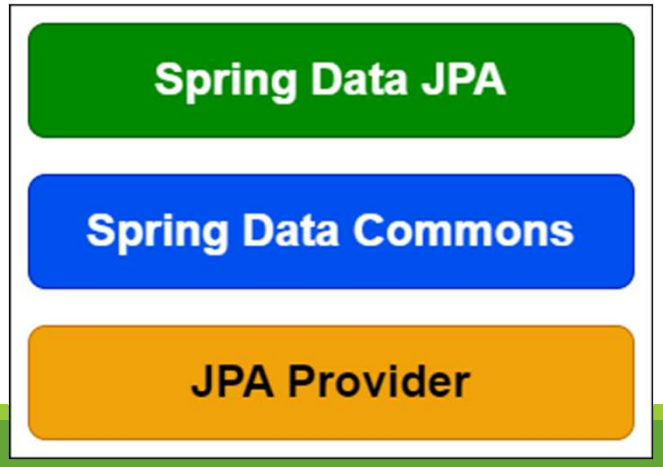
Spring Data JPA is not a JPA provider. It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider. If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following figure :

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.

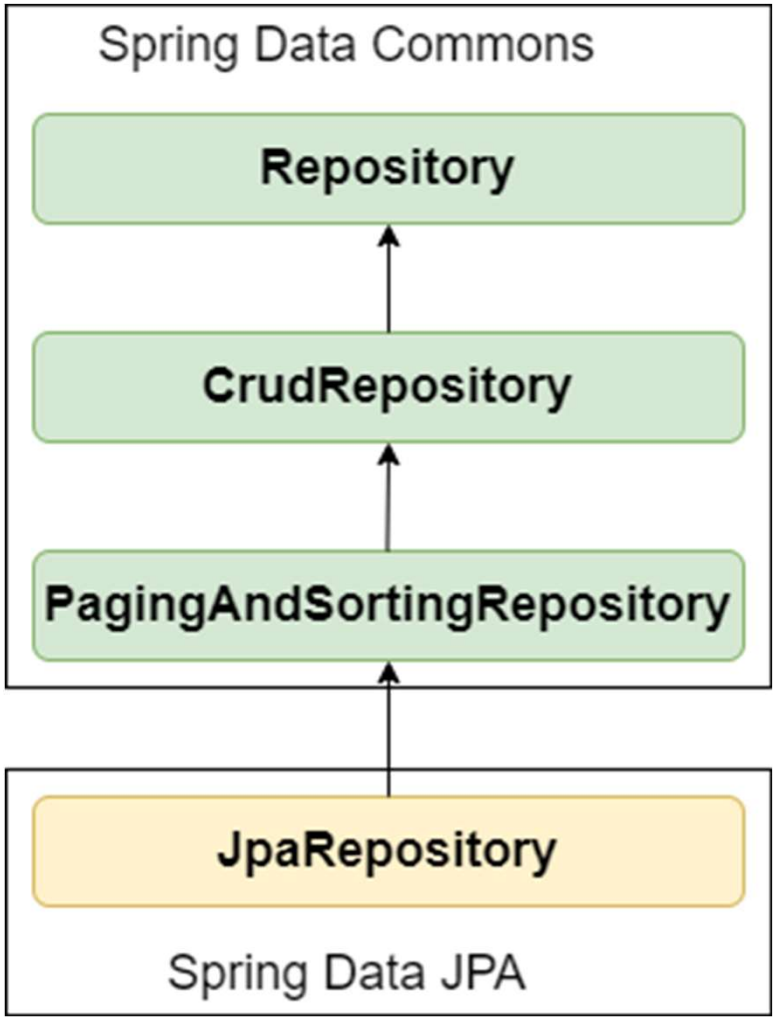
JPA Provider :- The JPA Provider implements the Java Persistence API.



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.

```
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
```

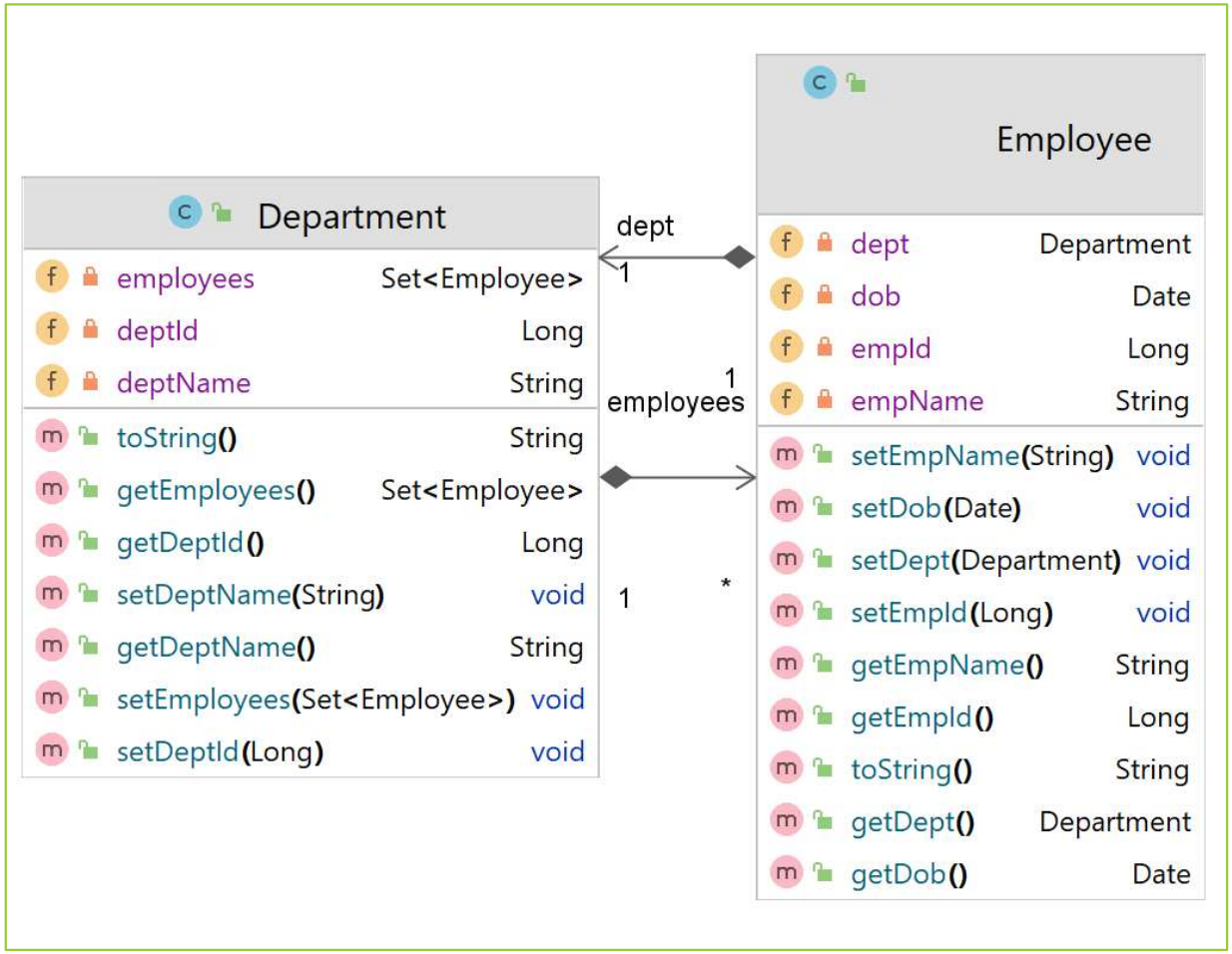
Entities

```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);
```

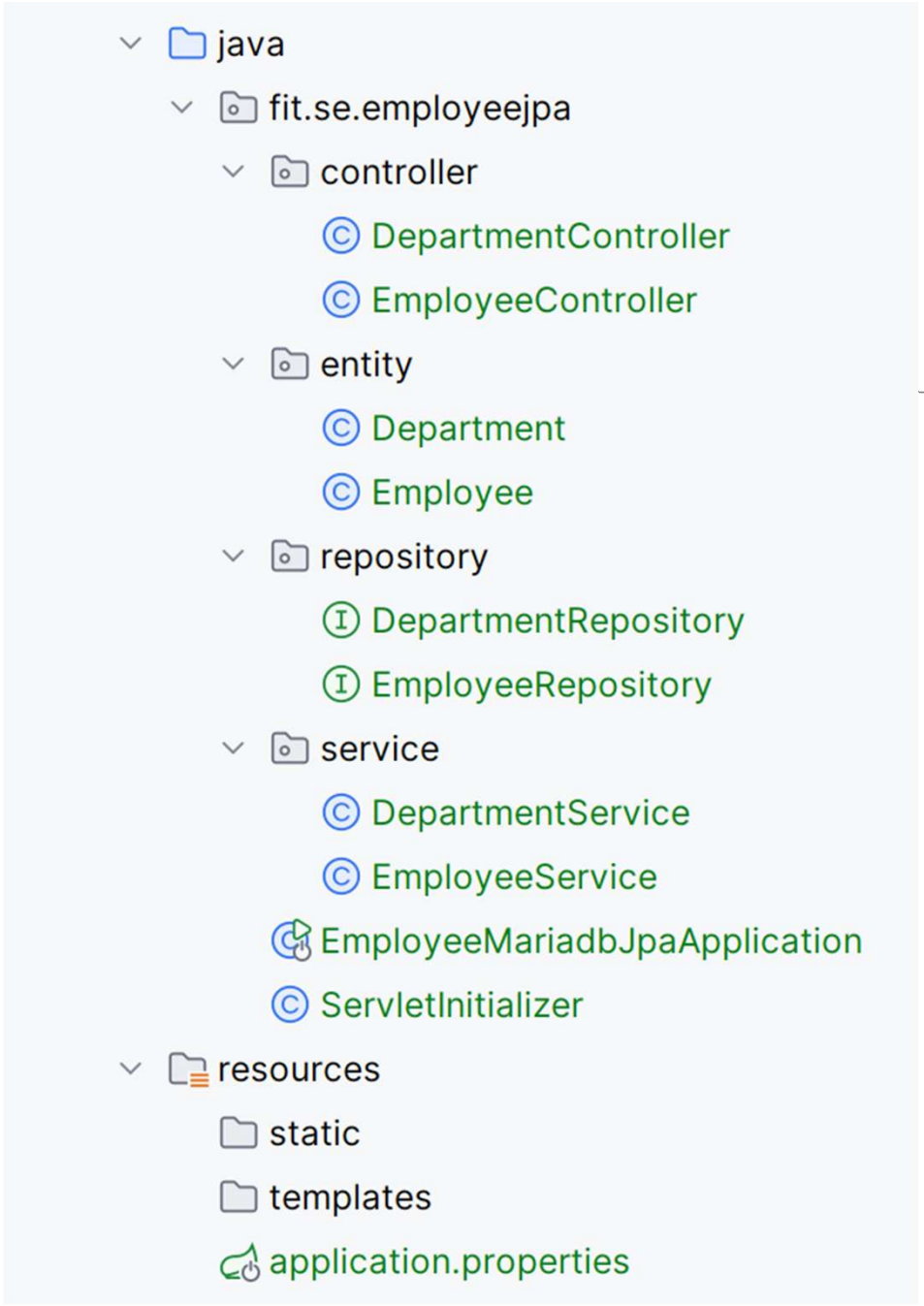
service

```
// 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.