

Spring Data

By Vo Van Hai

vovanhai@ueh.edu.vn

Spring Data - Main modules

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

- [Spring Data Commons](#) - Core Spring concepts underpinning every Spring Data module.
- [Spring Data JDBC](#) - Spring Data repository support for JDBC.
- [Spring Data JDBC Ext](#) - Support for database specific extensions to standard JDBC including support for Oracle RAC fast connection failover, AQ JMS support and support for using advanced data types.
- [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.
- [Spring Data for VMware Tanzu GemFire](#) - Easy configuration and access to Pivotal GemFire for your highly consistent, low latency/high through-put, data-oriented Spring applications.

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
 - 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

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 a Data Source

```
= ClassPathXmlApplicationContext("dbConfig.xml");
```

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

```
= new AnnotationConfigApplicationContext(DsConfig.class);
```

 application.properties

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

```
@Configuration
public class DsConfig {

    no usages

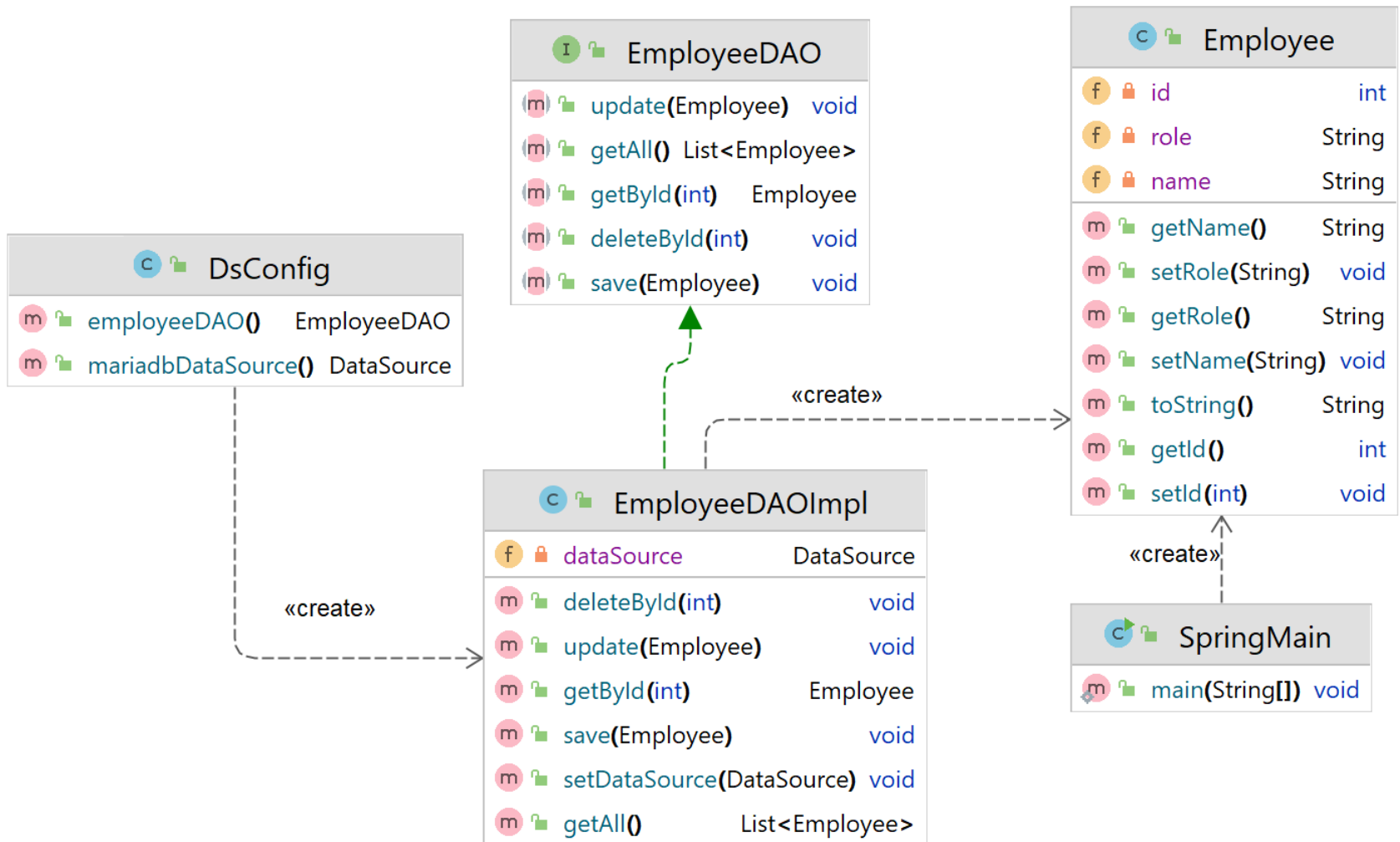
    @Bean
    @Scope("singleton")
    public DataSource mariadbDataSource() throws Throwable{
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("org.mariadb.jdbc.Driver");
        ds.setUrl("jdbc:mariadb://localhost:3306/sampled<u>b</u>");
        ds.setUsername("root");
        ds.setPassword("password");
        return ds;
    }
}
```

Execute SQL command

- The old way with JDBC objects:

```
public void insert(Employee employee) {  
    try {  
        String query = "insert into Employee (id, name, role) values (?, ?, ?)";  
        Connection con = dataSource.getConnection();  
        PreparedStatement ps = con.prepareStatement(query);  
        ps.setInt( parameterIndex: 1, employee.getId());  
        ps.setString( parameterIndex: 2, employee.getName());  
        ps.setString( parameterIndex: 3, employee.getRole());  
        int out = ps.executeUpdate();  
        if (out != 0) {  
            System.out.println("Employee saved with id=" + employee.getId());  
        } else System.out.println("Employee save failed with id=" + employee.getId());  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Example



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

```
public class StudentJdbcTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;
    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }
    public void create(String name, Integer age) {
        String SQL = "insert into Student (name, age) values (?, ?)";
        jdbcTemplateObject.update( SQL, name, age);
        System.out.println("Created Record Name = " + name + " Age = " + age);
        return;
    }
}
```

Example

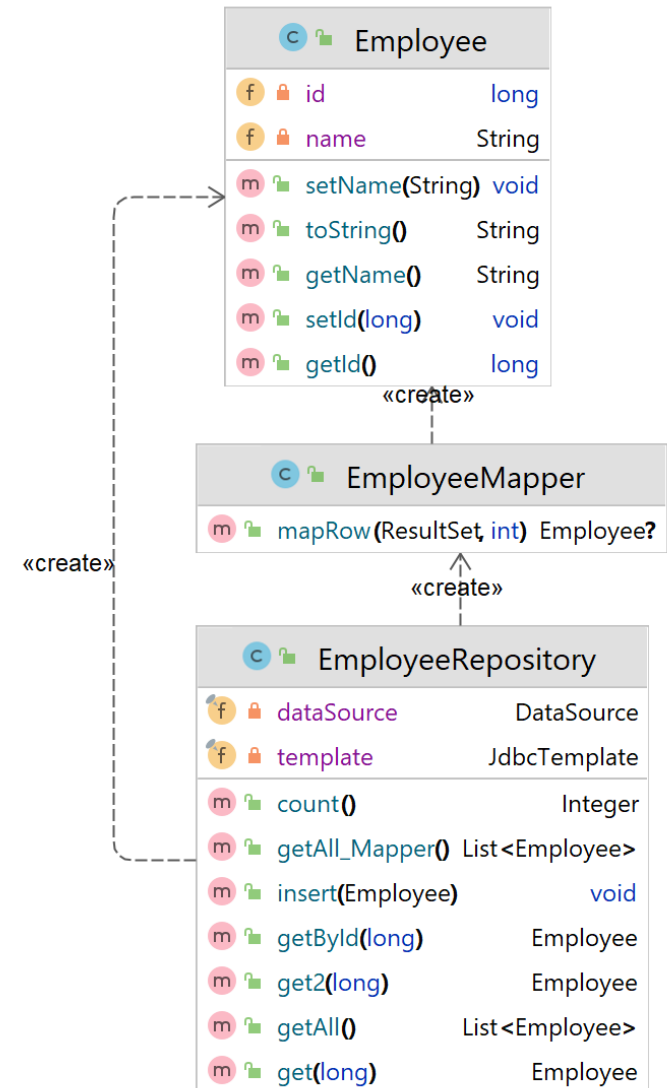
```
@SpringBootApplication
public class My1SpringbootApplication {
    @Autowired
    private EmployeeRepository repository;
    public static void main(String[] args) { SpringApplication.run(My1SpringbootApplication.class, args); }

    @Bean
    public CommandLineRunner example_01(){
        return new CommandLineRunner() {
            @Override
            public void run(String... args) throws Exception {
                Employee emp1=new Employee( id: 1002, name: "Teo");
                repository.insert(emp1);
                System.out.println("employee inserted");

                List<Employee> lst = repository.getAll();
                lst.forEach(System.out::println);

                Employee emp =repository.getById(1);
                System.out.println(emp);

                List<Employee> lst1 = repository.getAll_Mapper();
                lst1.forEach(System.out::println);
            }
        };
    }
}
```



Query row(s) with Rows Mapper

Use RowMapper object for mapping one row of the ResultSet to specific object.

```
public Employee getById(long id) {  
    return template.queryForObject(  
        sql: "select * from employee where id=?", //query  
        new EmployeeMapper()  
        ,id) //parameters  
    ;  
}
```

```
public class EmployeeMapper implements RowMapper<Employee> {  
    public Employee mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
        Employee emp = new Employee();  
        emp.setId(rs.getLong( columnLabel: "id"));  
        emp.setName(rs.getString( columnLabel: "name"));  
        return emp;  
    }  
}
```

Mapper class

```
public List<Employee> getAll_Mapper() {  
    String sql = "select * from employee ";  
    return template.query(sql,  
        new EmployeeMapper());  
}
```

Direct mapping

```
public Employee get2(long id) {  
    return template.queryForObject(  
        sql: "select * from employee where id=?", //query  
        //direct mapping  
        (rs, rowNum) -> new Employee(  
            rs.getLong( columnLabel: "id"),  
            rs.getString( columnLabel: "name")  
        )  
        ,id) //parameters  
    ;  
}
```

Query with BeanPropertyMapper

Using BeanPropertyMapper object will save you a lot of time.

```
//Query for single row
public Employee get(long id) {
    return template.queryForObject(
        sql: "select * from employee where id=?", //query
        new BeanPropertyRowMapper<>(Employee.class)//mapping
        ,id) //parameters
    ;
}
```

```
//Query for multiple rows
public List<Employee> getAll() {
    String sql = "select * from employee ";
    return template.query(sql,
        new BeanPropertyRowMapper<>(Employee.class));
}
```

```
//Query for single value
public Integer count() {
    String sql = "select count(*) from employee";
    return template.queryForObject(sql, //query
        Integer.class //value type return
    );
}
```

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

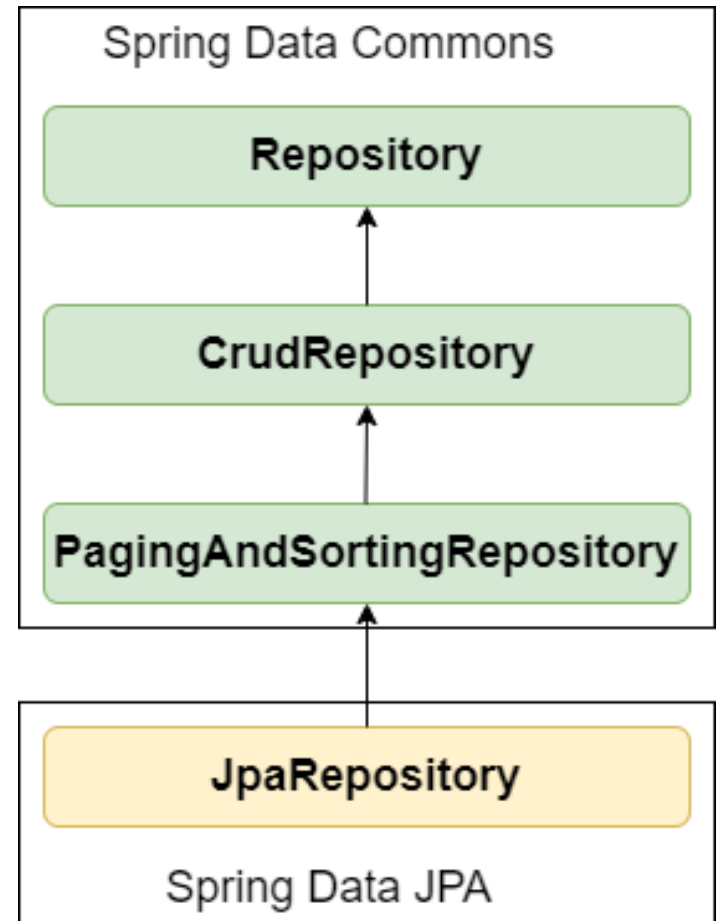
Spring Data Commons

JPA Provider

- ▶ **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 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.
 - *JpaSpecificationExecutor* — It is not a repository interface. It allows the execution of Specifications based on the JPA criteria API.

Working with Spring Data Repositories

Core concepts

- ▶ 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.
```

```
}
```

❶ Saves the given entity.

❷ Returns the entity identified by the given ID.

❸ Returns all entities.

❹ Returns the number of entities.

❺ Deletes the given entity.

❻ Indicates whether an entity with the given ID exists.

Setting up your project components

- ▶ The JDBC driver provides a database specific implementation of the JDBC API. (MS SQLServer, MySQL, MariaDB,...)
- ▶ 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.

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    runtimeOnly 'com.microsoft.sqlserver:mssql-jdbc'  
    runtimeOnly 'com.mysql:mysql-connector-j'  
    runtimeOnly 'org.mariadb.jdbc:mariadb-java-client'  
    //...  
}
```

Configure the DataSource

```
@Configuration
@EnableJpaRepositories(basePackages = {"iuh.edu.vn.repositories"})
@EntityScan(basePackages = {"iuh.edu.vn.entities"})
public class DbConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.mariadb.jdbc.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        dataSource.setUrl("jdbc:mariadb://localhost:3306/sampledb?createDatabaseIfNotExist=true");
        return dataSource;
    }
}
```



```
application.properties x
1 # MariaDB
2 spring.datasource.url=jdbc:mariadb://localhost:3306/sampledb
3 spring.datasource.username=root
4 spring.datasource.password=password
5 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
6 spring.jpa.hibernate.ddl-auto=update
7 spring.jpa.show-sql=true
```

Entities

- ▶ Use `@Entity` for entity
- ▶ Use `@Table` if you want to customize your table
- ▶ Should have `@Id` for the primary-key.
- ▶ Use `@Column` for customize the column
- ▶ And other annotation if needed
 - `@OneToMany`
 - `@ManyToOne`
 - `@OneToOne`
 - ...

Entity sample

@Entity

```
public class Department {
```

 @Id

 @GeneratedValue(strategy = GenerationType.AUTO)

```
  private Long deptId;
```

 @Column(unique = true, nullable = false, length = 150)

```
  private String deptName;
```

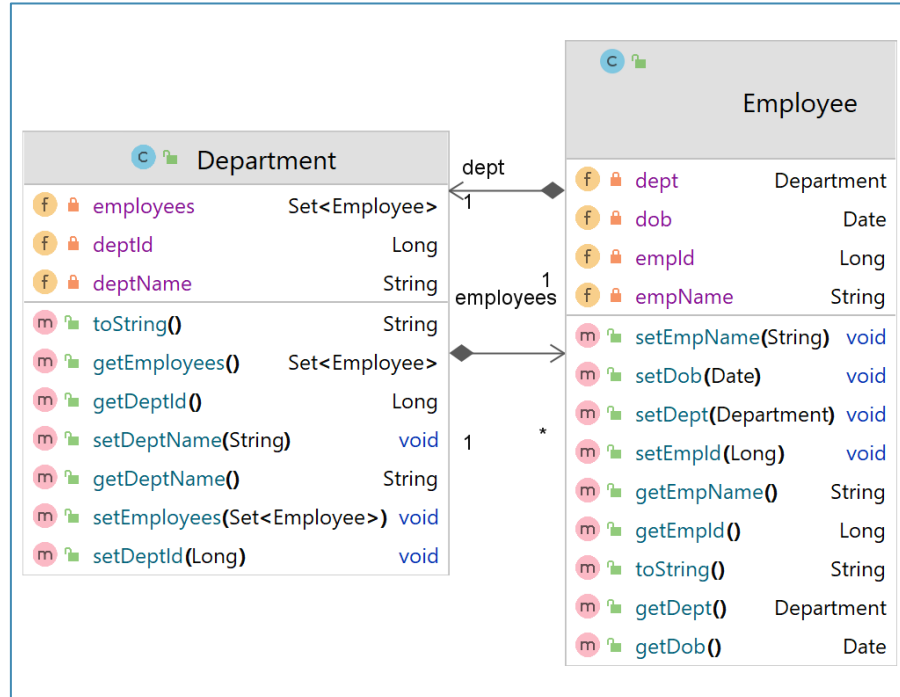
 @OneToMany(mappedBy = "dept",

 cascade = CascadeType.ALL, fetch = FetchType.EAGER)

```
  private Set<Employee> employees
```

```
  ...
```

```
}
```



@Entity

```
public class Employee {
```

 @Id @GeneratedValue(strategy = GenerationType.AUTO)

```
  private Long empId;
```

```
  private String empName;
```

```
  private Date dob;
```

 @ManyToOne(fetch = FetchType.LAZY)

```
  private Department dept;
```

```
  ...
```

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.

```
@Configuration
@EnableJpaRepositories(basePackages = {"iuh.edu.vn.repositories"})
@EntityScan(basePackages = {"iuh.edu.vn.entities"})
public class AppConfig {
```

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

► Example

```
interface PersonRepository extends Repository<Person, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```

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

Queries

► JPQL

- By default, the query definition uses JPQL.

```
@Query("SELECT COUNT(*) FROM Department d")  
long getDeptCount();
```

► Native

```
@Query(value = "SELECT * FROM Employee u WHERE u.status = 1", nativeQuery = true)
```

```
► List<Employee> findAllActiveEmployees();
```

@Modifying

```
@Query("update Employee u set u.status = :status where u.empld = :id")  
int updateEmployeeStatusForName(@Param("status") Integer status,  
                                @Param("id") Long empld);
```

Query Parameters (1)

► Indexed Query Parameters

- Spring Data will pass method parameters to the query in the same order they appear in the method declaration

```
@Query("SELECT e FROM Employee e WHERE e.status = ?1")
```

```
List<Employee> findEmployeeByStatus(Integer status);
```

```
@Query("SELECT e FROM Employee e WHERE e.status = ?1 and e.empName = ?2")
```

```
List<Employee> findEmployeeByStatusAndName(Integer status, String name);
```

► Named Parameters

- Each parameter annotated with @Param must have a value string matching the corresponding JPQL or SQL query parameter name.

```
@Query("SELECT e FROM Employee e WHERE e.status = :status and e.empName = :name")
```

```
Employee findEmployeeByStatusAndEmployeeName(  
    @Param("status") Integer empStatus,  
    @Param("name") String empName);
```

Query Parameters (2)

► Collection Parameter

- Use in the case when the where clause of our JPQL or SQL query contains the IN (or NOT IN) keyword:

Repository

```
@Query(value = "SELECT e FROM Employee e WHERE e.empName IN :names")  
List<Employee> findEmployeeByNameList(  
    @Param("names") Collection<String> names);
```

Service

```
public List<Employee> findEmployeeByNameList(Collection<String>names){  
    return employeeRepository.findEmployeeByNameList(names);  
}
```

Application

```
Collection<String> names=List.of("Ty","Teo","Men");  
List<Employee> lst = employeeServices.findEmployeeByNameList(names);  
lst.stream().iterator().forEachRemaining(employee -> {  
    System.out.println("-----" + employee.getEmpName());  
});
```

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.

```
import org.springframework.data.domain.Pageable;
//...
public interface ProductRepository extends
PagingAndSortingRepository<Product, Integer> {

    List<Product> findAllByPrice(double price, Pageable pageable);

}
```

```
Pageable sortedByName =
    PageRequest.of(0, 3, Sort.by("name"));
```

```
Pageable sortedByPriceDesc =
    PageRequest.of(0, 3, Sort.by("price").descending());
```

```
Pageable sortedByPriceDescNameAsc =
    PageRequest.of(0, 5, Sort.by("price").descending().and(Sort.by("name")));
```

Product		
f	price	double
f	id	int
f	name	String
m	getId()	int
m	setId(int)	void
m	from(int, String, double)	Product
m	setName(String)	void
m	getPrice()	double
m	setPrice(double)	void
m	toString()	String
m	getName()	String

Streaming Query Results

```
@Query("select e from Employee e")
Stream<Employee> findAllByCustomQueryAndStream();

@Query("select e from Employee e")
Stream<Employee> streamAllPaged(Pageable pageable);
```

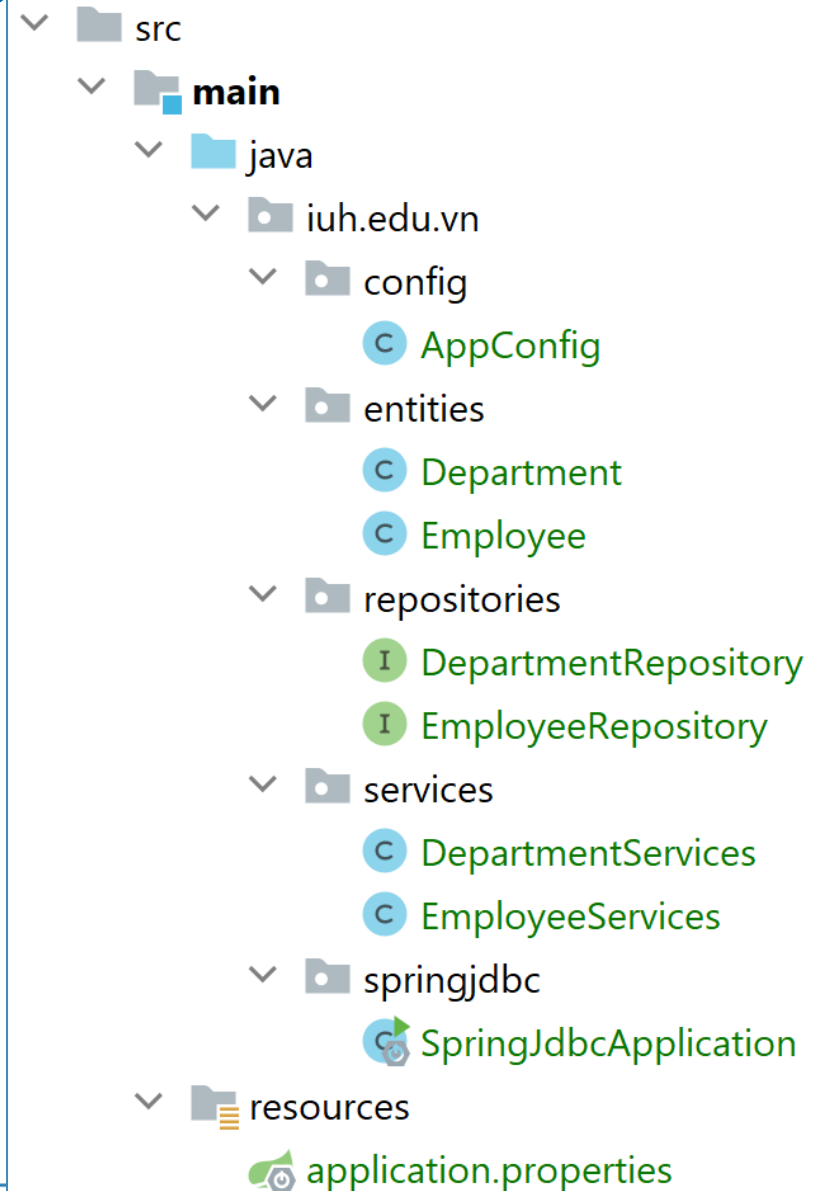
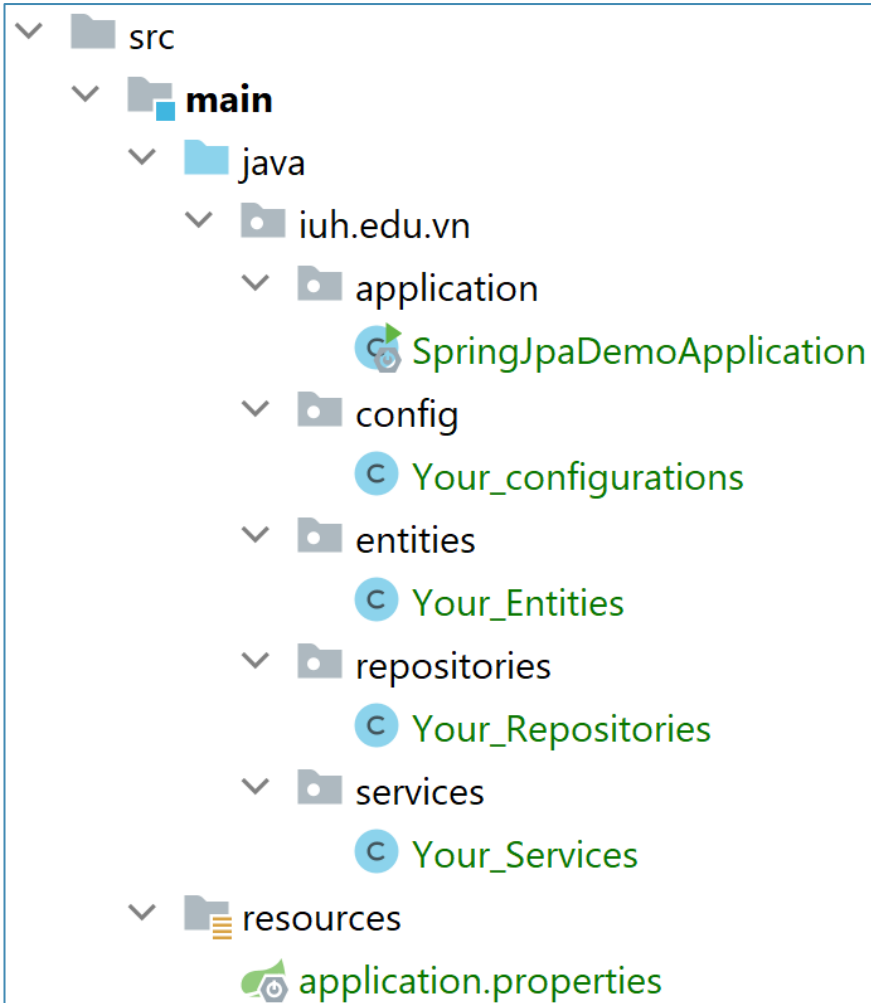
```
try (Stream<Employee> stream = employeeServices.findAllByCustomQueryAndStream()) {
    stream.forEach(employee -> {
        log.info(employee.getEmpName());
    });
}
```

Asynchronous Query Results

```
@Async
Future<Employee> findByEmpName(String name);

@Async
CompletableFuture<Employee> findOneByEmpName(String name);
```

Demo: Directories structures



entities/models/domains

Test application

```
@SpringBootApplication(scanBasePackages = "iuh.edu.vn")
public class SpringJdbcApplication {
    private static final Logger log = LoggerFactory.getLogger(SpringJdbcApplication.class);

    public static void main(String[] args) { SpringApplication.run(SpringJdbcApplication.class, args); }

    @Bean
    public CommandLineRunner sampleRecords(EmployeeServices employeeServices,
                                           DepartmentServices departmentServices) {
        return args -> {
            Department dept1 = new Department( deptName: "Software Engineering");
            Department dept2 = new Department( deptName: "Information Technology");
            departmentServices.save(dept1);
            departmentServices.save(dept2);

            employeeServices.save(new Employee( empName: "Nguyen Van Teo", Date.valueOf(s: "1997-10-21"),
                                                status: 1, email: "teo@gmail.com", dept1));
            employeeServices.save(new Employee( empName: "Than Thi Det", Date.valueOf(s: "1995-9-26"),
                                                status: 1, email: "det@hotmail.com", dept2));
        };
    }
}
```


MongoDB Model Implementation

- ▶ MongoDB and Spring Boot can be interacted by using the MongoTemplate class and MongoRepository interface.
 - 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.
 - MongoRepository — MongoRepository is used for basic queries that involve all or many fields of the document. Examples include data creation, viewing documents, and more.

implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'

Connect Server

```
application.properties x
1 spring.data.mongodb.host=localhost
2 spring.data.mongodb.database=test_db
3 spring.data.mongodb.port=27017
4 spring.data.mongodb.authentication-database=admin
5 spring.data.mongodb.username=root
6 spring.data.mongodb.password=root
7 spring.data.mongodb.uri=mongodb://localhost:27017/test_db
```

create user if you did not have one

```
db.createUser(
{
  user: 'admin',
  pwd: 'password',
  roles: [ { role: 'root', db: 'admin' } ]
}
);
```

```
use admin

db.createUser( { user: "root",
                  pwd: "root",
                  roles: [ { role: "readWrite", db: "db_test" } ],
                  } );
```

Connect Server -simple

```
@Configuration
public class SimpleMongoConfig {
    @Autowired
    private Environment env;
    @Bean
    public MongoClient mongo() {
        ConnectionString connectionString
            = new ConnectionString(env.getProperty("spring.data.mongodb.uri"));
        MongoClientSettings mongoClientSettings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
        return MongoClients.create(mongoClientSettings);
    }
    @Bean
    public MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongo(), databaseName: "test_db");
    }
}
```

Connect Server – abstract config

```
@Configuration
public class MongoConfig extends AbstractMongoClientConfiguration {

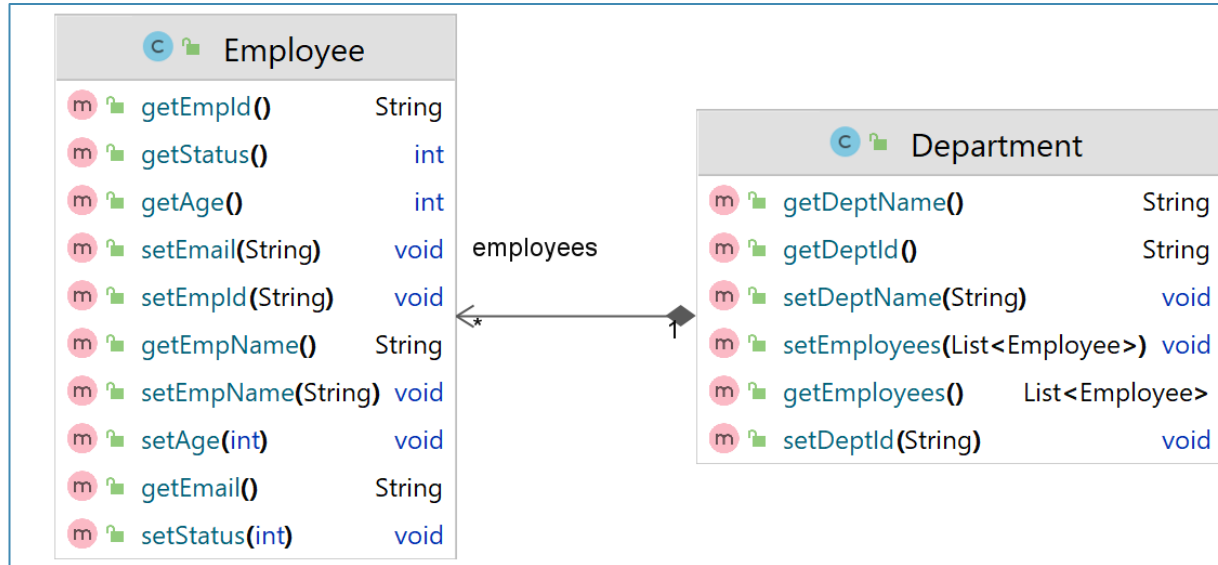
    @Autowired
    private Environment env;

    protected String getDatabaseName() {
        return "test";
    }

    public MongoClient mongoClient() {
        ConnectionString connectionString
            = new ConnectionString(env.getProperty("spring.data.mongodb.uri"));
        MongoClientSettings mongoClientSettings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
        return MongoClients.create(mongoClientSettings);
    }

    public Collection getMappingBasePackages() {
        return Collections.singleton("vn.edu.iuh.models");
    }
}
```

Entities Documentation



```
@Document("Department")
public class Department {
    @Id
    private String deptId;
    private String deptName;
    private List<Employee> employees;
```

```
@Document("Employee")
public class Employee {
    @Id
    private String empId;
    private String empName;
    private int age;
    private int status = 0;
    private String email;
```

Repositories

- ▶ Repositories in our application should extend the `MongoRepository<T, ID>` interface.
- ▶ `MongoRepository<T, ID>` provides most CRUD operators.
- ▶ We also define more queries using spring naming-method or using `@Query` annotation

```
public interface DepartmentRepository
    extends MongoRepository<Department, String> {

    Department findDepartmentByDeptNameContains(String something);

    @Query(value = "'employees.empName': ?0", fields = "'employees' : 0")
    Department findDepartmentByEmployeeName(String empName);
}
```

Operations

```
@Bean
public CommandLineRunner tests(DepartmentRepository departmentRepository) {
    return args -> {
        Department dept1 = new Department( deptId: "SE", deptName: "Software Engineering");
        List<Employee> lst = dept1.getEmployees();
        lst.add(new Employee( empId: "emp001", empName: "Nguyen Van Teo", age: 20, status: 1, email: "teo@gmail.com"));
        lst.add(new Employee( empId: "emp002", empName: "Than Thi Det", age: 25, status: 0, email: "det@gmail.com"));
        lst.add(new Employee( empId: "emp002", empName: "Tran Thi Men", age: 23, status: 1, email: "men@gmail.com"));
        departmentRepository.save(dept1);
        departmentRepository.save(new Department( deptId: "IS", deptName: "Information System"));

        departmentRepository.findAll().stream().forEach(department -> {
            System.out.println(department.getDeptName());
        });
    };
}
```

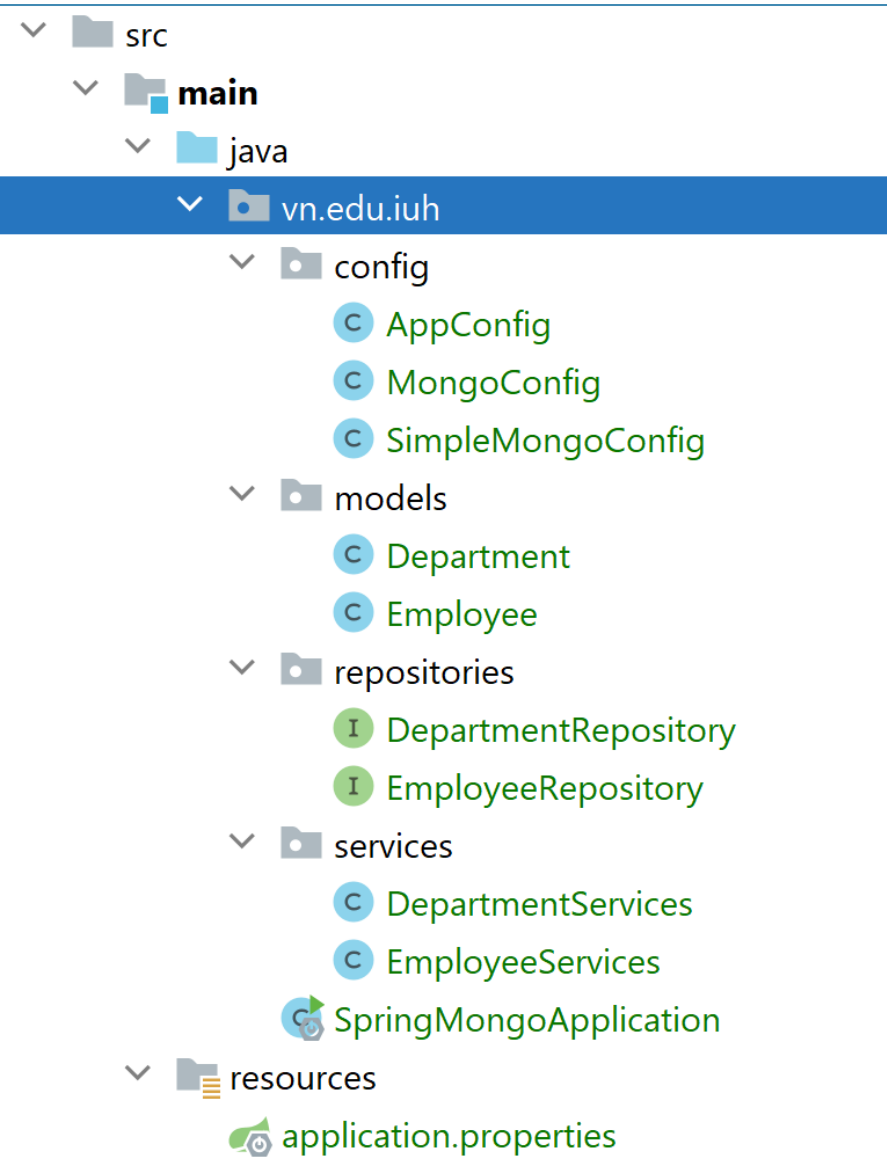
Using Repository

```
@Bean
public CommandLineRunner test2(MongoTemplate mongoTemplate) {
    return args -> {
        Department dept2 = new Department( deptId: "CS", deptName: "Computer Science");
        List<Employee> lst = dept2.getEmployees();
        lst.add(new Employee( empId: "emp003", empName: "Nguyen Van Ty", age: 24, status: 1, email: "teo@gmail.com"));
        lst.add(new Employee( empId: "emp004", empName: "Hoang van Tun", age: 25, status: 1, email: "tun@gmail.com"));

        mongoTemplate.save(dept2);
    };
}
```

Using MongoTemplate

Example structure



REST introduction

- ▶ **RE**presentational **S**tate **T**ransfer(REST) is lightweight approach for communicating between applications.
- ▶ REST APIs enable you to develop all kinds of web applications having all possible CRUD (**C**reate, **R**etrieve, **U**ppdate, **D**eleate) operations.
- ▶ Characteristics:
 - Language independent
 - Platform independent
 - Can use any data format (XML and JSON are common).

REST API

REST
Web Services

REST Services

RESTful API

RESTful
Web Services

RESTful Services

REST over HTTP

<https://www.w3.org/Protocols/rfc2616/rfc2616.txt>

► REST uses HTTP methods for communication.

- HTTP methods: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, TRACE, CONNECT.
- REST methods: POST, GET, PUT, PATCH, DELETE.

Methods	Operation
GET	Returns a representational view of a resource's contents and data.
POST	Primarily operates on resource collections.
PUT	Updates a resource by <u>replacing</u> its content entirely. It's possible to create a resource with a PUT method (careful : creating resource without intention.)
PATCH	Only modifies resource contents. Attention to applying PATCH methods to a whole resource collection.
DELETE	When a DELETE method targets a single resource, that resource is removed entirely. (Avoid using the DELETE method in a resource collection)

REST methods summary

HTTP Method	CRUD	Collection Resource (e.g. /users)	Single Resource (e.g. /users/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID	Avoid using POST on a single resource
GET	Read	200 (OK), list of users. Use pagination, sorting, and filtering to navigate big lists	200 (OK), single user. 404 (Not Found), if ID not found or invalid
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
PATCH	Partial Update/Modify	405 (Method not allowed), unless you want to modify the collection itself	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution	200 (OK). 404 (Not Found), if ID not found

Response status

- ▶ HTTP response status codes indicate whether a specific HTTP request has been successfully completed.
- ▶ Responses are grouped in five classes:

Code range	Meaning
100–199	Informational responses
200–299	Successful responses
300–399	Redirection message
400–499	Client error responses
500–599	Server error responses

Spring Data REST

Introduction

- ▶ Spring Data REST is itself a Spring MVC application and is designed in such a way that it should integrate with your existing Spring MVC applications with little effort.
- ▶ Dependencies


Gradle

```
dependencies {  
    implementation 'org.springframework.data:spring-data-rest-webmvc:4.0.4'
```

Maven

```
<dependency>  
  <groupId>org.springframework.data</groupId>  
  <artifactId>spring-data-rest-webmvc</artifactId>  
  <version>4.0.4</version>  
</dependency>
```

Adding Spring Data REST to a Spring Boot Project

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
     implementation 'org.springframework.boot:spring-boot-starter-data-rest'
```

<https://docs.spring.io/spring-data/rest/docs/current/reference/html/#reference>

Basic Settings for Spring Data REST

► Basic Settings:

- Setting the Repository Detection Strategy
- Changing the Base URI
- Changing Other Spring Data REST Properties

Basic Settings for Spring Data REST

#1. Setting the Repository Detection Strategy

- ▶ Spring Data REST uses a implementation of the `RepositoryDetectionStrategy` interface to determine whether a repository is exported as a REST resource.
- ▶ The `RepositoryDiscoveryStrategies` inside this interface enumeration includes the following values:

Name	Description
DEFAULT	Exposes all public repository interfaces but considers the exported flag of <code>@(Repository)RestResource</code> .
ALL	Exposes all repositories independently of type visibility and annotations.
ANNOTATED	Only repositories annotated with <code>@(Repository)RestResource</code> are exposed, unless their exported flag is set to false.
VISIBILITY	Only public repositories annotated are exposed.

Table: Repository detection strategies

Basic Settings for Spring Data REST

#2. Changing the Base URI

- ▶ By default, Spring Data REST serves up REST resources at the root URI, '/'. It's harmful.
- ▶ Change the base(root) path by:
 1. Application.properties file: add the instructions:
`spring.data.rest.basePath=/api`
 2. Configuration

```
@Configuration
class CustomRestMvcConfiguration {
    @Bean
    public RepositoryRestConfigurer repositoryRestConfigurer() {
        return new RepositoryRestConfigurer() {
            @Override
            public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config,
                                                            CorsRegistry cors) {
                config.setBasePath("/api");
            }
        };
    }
}
```


Basic Settings for Spring Data REST

#2. Changing the Base URI

- ▶ Change the base(root) path by:
 3. Implement the RepositoryRestConfigurer interface

```
@Component
public class CustomizedRestMvcConfiguration implements RepositoryRestConfigurer {
    @Override
    public void configureRepositoryRestConfiguration(
        RepositoryRestConfiguration config, CorsRegistry cors) {

        config.setBasePath("/api");
    }
}
```

Basic Settings for Spring Data REST

#3. Changing Other Spring Data REST Properties

Property	Description
basePath	the root URI for Spring Data REST
defaultPageSize	change the default for the number of items served in a single page
maxPageSize	change the maximum number of items in a single page
pageParamName	change the name of the query parameter for selecting pages
limitParamName	change the name of the query parameter for the number of items to show in a page
sortParamName	change the name of the query parameter for sorting
defaultMediaType	change the default media type to use when none is specified
returnBodyOnCreate	change whether a body should be returned when creating a new entity
returnBodyOnUpdate	change whether a body should be returned when updating an entity

Repository resources exposure

Fundamentals

- ▶ The core functionality of Spring Data REST is to export resources for Spring Data repositories. Thus, the core artifact to look at and potentially customize the way the exporting works is the repository interface.

```
//@Repository  
public interface DepartmentRepository extends JpaRepository<Department, Long> {///...}
```

- For this repository, Spring Data REST exposes a collection resource at **/departments**
- ▶ Change the default path by using

```
@RepositoryRestResource(collectionResourceRel = "dept", path = "dept")  
public interface DepartmentRepository extends JpaRepository<Department, Long> {///...}
```

- **/departments** now changes to **/dept**
- ▶ Repository methods exposure
 - The resource exposure will follow which methods you have exposed on the repository → all HTTP resources we can register by default.
 - Use **@RestResource**(exported = **false**) annotation to limit the exposure

Repository resources exposure

Resource Discoverability

- ▶ <http://localhost:8080/api>

Response Body

```
{
  "_links": {
    "departments": {
      "href": "http://localhost:8080/api/departments{?page,size,sort}",
      "templated": true
    },
    "employees": {
      "href": "http://localhost:8080/api/employees{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/api/profile"
    }
  }
}
```

Repository resources exposure

Default Status Codes

- ▶ For the resources exposed, we use a set of default status codes:
 - 200 OK: For plain GET requests.
 - 201 Created: For POST and PUT requests that create new resources.
 - 204 No Content: For PUT, PATCH, and DELETE requests when the configuration is set to not return response bodies for resource updates.

Resource Discoverability

Tools

- ▶ *Postman*
- ▶ *CURL*
- ▶ The HAL Explorer implementation `'org.springframework.data:spring-data-rest-hal-explorer'`

The screenshot shows the HAL Explorer web application running in a browser. The address bar shows the URL `http://localhost:8080/api/explorer/index.html#uri=/api`. The application has a dark theme and a navigation bar with links for Theme, Settings, and About.

The main interface is divided into several sections:

- Edit Headers**: A text input field containing `/api` and a **Go!** button.
- Links**: A table with columns for Relation, Name, Title, HTTP Request, and Doc. It lists three relations: departments, employees, and profile, each with a set of navigation buttons (back, forward, search, etc.).
- Response Status**: Shows the status code `200 (OK)`.
- Response Headers**: A table listing headers such as `connection: keep-alive`, `content-type: application/hal+json`, `date: Mon, 10 Apr 2023 03:01:21 GMT`, `keep-alive: timeout=60`, `transfer-encoding: chunked`, and `vary: Origin, Access-Control-Request-Method, Acc`.
- Response Body**: A JSON object representing the HAL resource, with links to departments, employees, and profile.

```
{
  "_links": {
    "departments": {
      "href": "http://localhost:8080/api/departments?page,size,sort",
      "templated": true
    },
    "employees": {
      "href": "http://localhost:8080/api/employees?page,size,sort",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/api/profile"
    }
  }
}
```

Accessing path

/api/collections_name/{id}

/api/collections_name/{id}/attribute

/api/collections_name/?name=value



```
{
  deptName: "Software Engineering",
  - employees: [
    - {
      empName: "Nguyen Van Teo",
      age: 20,
      status: 1,
      email: "teo@gmail.com"
    },
    - {
      empName: "Than Thi Det",
      age: 25,
      status: 0,
      email: "det@gmail.com"
    },
    - {
      empName: "Tran Thi Men",
      age: 23,
      status: 1,
      email: "men@gmail.com"
    }
  ],
  - _links: {
    - self: {
      href: "http://localhost:8080/api/departments/SE"
    },
    - department: {
      href: "http://localhost:8080/api/departments/SE"
    }
  }
}
```

The Collection Resources

► Parameters

- If the repository has pagination capabilities, the resource takes the following parameters:
 - **page**: The page number to access (0 indexed, defaults to 0).
 - **size**: The page size requested (defaults to 20).
 - **sort**: A collection of sort directives in the format *(\$propertyname,)+[asc|desc]?*.

► Supported Media Types

- *application/hal+json*
- *application/json*

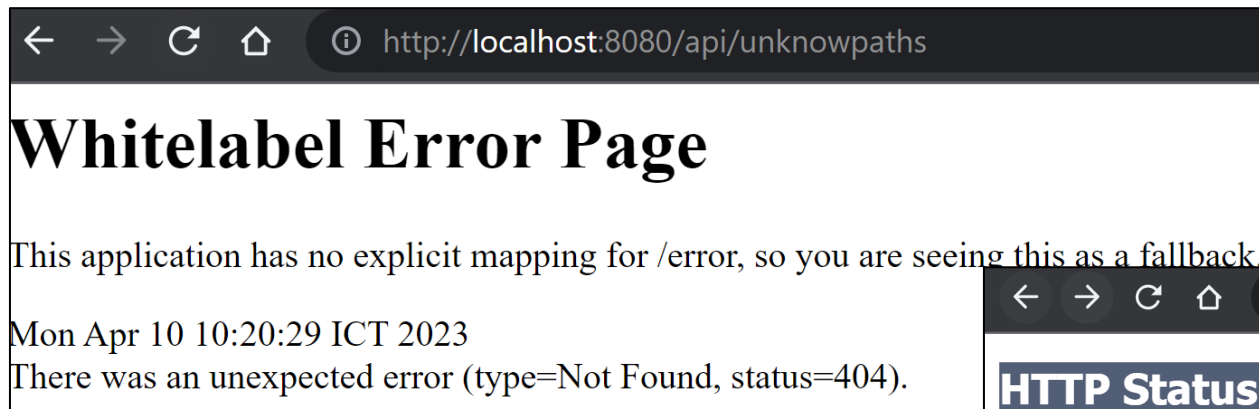
Whitelabel Error Page

► Disable using properties file

- `server.error.whitelabel.enabled=false`
- Or using annotation
- `@EnableAutoConfiguration(exclude {ErrorMvcAutoConfiguration.class})`

=

► Other fixings: (study later)



Transaction

► Global Transactions

- Global transactions let you work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA (Java Transaction API).
- `@EnableTransactionManagement`

► Local Transactions

- Local transactions are resource-specific, such as a transaction associated with a JDBC connection.
- Local transactions may be easier to use but have a significant disadvantage: They cannot work across multiple transactional resources.
- `@Autowired TransactionManager`

```

@Service
public class DepartmentServices {
    private DepartmentRepository departmentRepository;
    private TransactionManager transactionManager;
    @Autowired
    public DepartmentServices(DepartmentRepository departmentRepository,
                             TransactionManager transactionManager) {
        this.departmentRepository = departmentRepository;
        this.transactionManager = transactionManager;
    }
    public void insert(Department department){
        try {
            transactionManager.begin();
            departmentRepository.save(department);
            transactionManager.commit();
        } catch (Exception ex){
            try {
                transactionManager.rollback();
            } catch (SystemException e) {
                throw new RuntimeException(e);
            }
            ex.printStackTrace();
        }
    }
}

```

Spring HATEOAS 1.0

- ▶ <https://docs.spring.io/spring-hateoas/docs/current/reference/html/>

