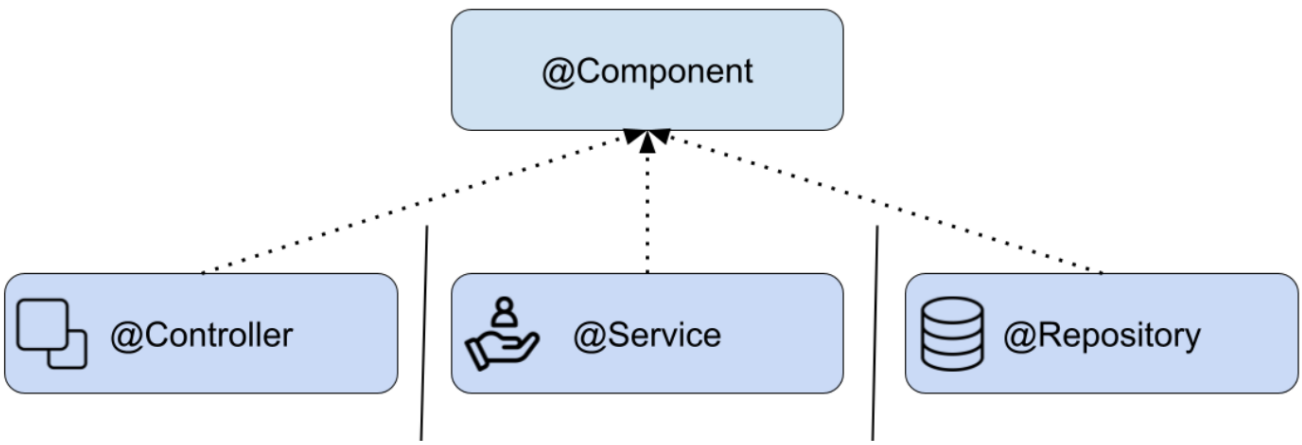
13-Repository & Entity

Author: Vincent Lau

Repository Bean

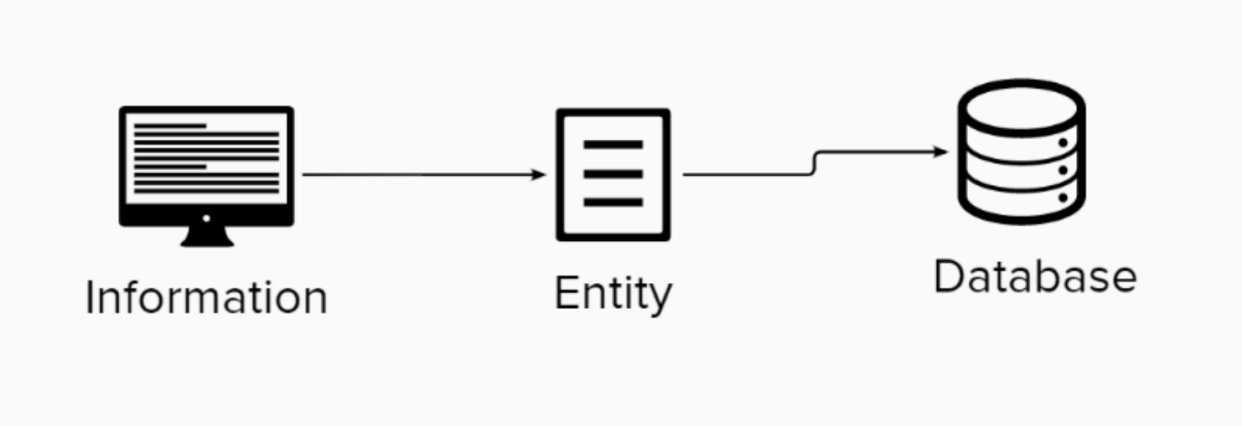
Introduction



A `@Repository` is a Spring component and stereotype that is part of the 'persistence layer'. The persistence layer is the way our microservice stores and retrieves the entities it

needs.

An `entity` is the class of our application that represents an object of your business, for example a User, Customer, Product, etc.



If our application needs a *User* (user with his first name, last name, date of birth) and needs to store it, this *User* is an entity of our domain model and is part of the persistence layer.

Entity

In Spring Data JPA, an **entity** represents a persistent object or data model that is typically mapped to a database table. Entities are a fundamental concept in the JPA (Java Persistence API) and are used to interact with the underlying relational database. Spring Data JPA simplifies the creation and management of entities.

Here's an example of a `User` entity in Spring Data JPA:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {

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

    private String username;
    private String email;

    // Constructors, getters, and setters
}
```

Let's break down the components of this `User` entity:

- `@Entity` : This annotation marks the class as a JPA entity, indicating that instances of this class can be stored in a relational database.
- `@Id` : This annotation specifies the primary key of the entity.
- `@GeneratedValue` : It configures how the primary key is generated. In this example, we use `GenerationType.IDENTITY` to specify that the primary key values are automatically generated by the database.
- The `id` field represents the primary key of the `User` entity.
- The `username` and `email` fields represent other attributes of the entity.
- Getters and setters: These methods are used to access and modify the attributes of the entity.

With this `User` entity defined, you can use it in a Spring Data JPA repository to perform common database operations such as saving, retrieving, updating, and deleting `User` records. Spring Data JPA will automatically generate the necessary SQL queries and handle the database interactions for you.

Repository

Here's an example of a Spring Data JPA repository for the `User` entity:

```
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // You can define custom query methods here if needed
}
```

The `UserRepository` interface extends `JpaRepository` , which provides out-of-the-box CRUD (Create, Read, Update, Delete) functionality for the `User` entity. You can also define custom query methods in this interface to retrieve data based on specific criteria.

Spring Data JPA simplifies data access and persistence in Spring applications, allowing you to focus on your business logic while it handles database interactions.

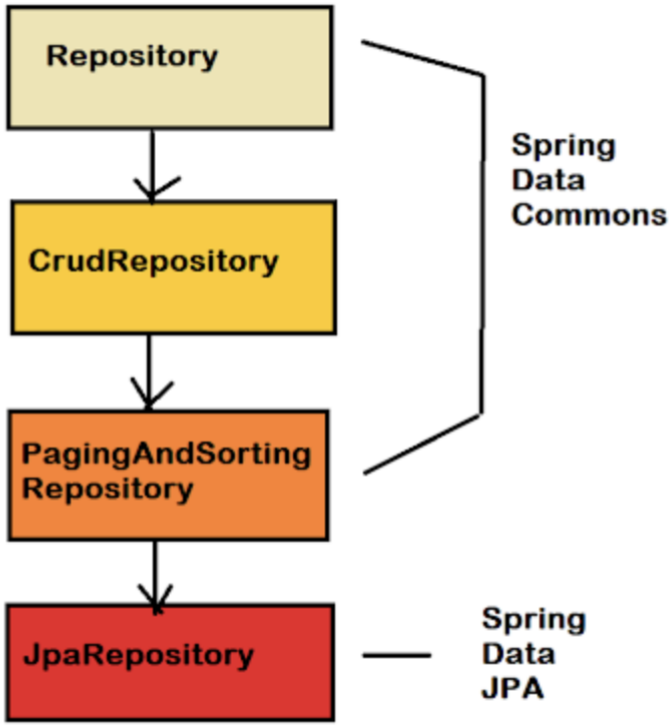
How it works

In Spring a ‘Repository’ is the component in charge of resolving the access to the data of our microservice. If we need to save, modify, delete records of a ‘User’ of the system; then the *UserRepository* component will be in charge of making direct changes on the User records.

Spring provides us with the basic functionality to store, delete and search entities. For this we have all the *interfaces* that extend "org.springframework.data.repository.Repository".

For our `User` entity, we can think of the basic operations to create, modify and search (CRUD). CRUD is an acronym that refers to the four minimum functions on an entity –> Create, Read, Update, Delete.

Layers of Repository



CrudRepository

Let's create an *interface UserRepository* that will extend *CrudRepository*.

- Annotate the class 'UserRepository' as a *@Repository* component.
- CrudRepository is a generic interface that receives two types. The first is the class that this interface will handle, and the second is the data type of the entity's *ID*.

The methods provided by *CrudRepository* are:

- `save` : saves an entity
- `saveAll` : saves the entities of an iterable list
- `findById` : searches for the identifier
- `existsById` : checks if an identifier exists
- `findAll` : returns all elements for the entity
- `findAllById` : searches for all elements having the identifier

- `count` : returns the total number of records for the entity
- `deleteById` : deletes a record for the identifier
- `delete` : deletes the entity
- `deleteAllById` : deletes all the elements corresponding to the id
- `deleteAll(Iterable)` : deletes all the elements received in the parameter
- `deleteAll()` : deletes all elements

```
import com.example.springbootcourse.model.User;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends CrudRepository<User, Long> {
}
```

- **Notice, that we do NOT implement the interface.**
- We create a new interface and extend *CrudRepository*. **Spring will be in charge of the implementation of the interface with the concrete class.**

JPARepository

JPARepository inherits all CrudRepository methods. Therefore, we can use all the methods of the CrudRepository interface. Also, JPARepository inherits from PagingAndSortingRepository, with methods for find with sort and paging.

JPARepository also adds other methods. JPARepository api.

- `saveAndFlush` : saves an entity and flushes changes instantly.
- `flush` : flushes all pending changes to the database.
- `saveAllAndFlush` : saves all entities and flushes changes instantly.
- `deleteAllInBatch` : deletes the given entities in a batch
- `deleteAllByIdInBatch` : deletes the entities identifier
- `deleteAllInBatch` : deletes all entities in a batch
- `getById` : returns a reference to the entity with the given identifier
- `findAll(Sort)` : returns all entities sorted
- `findAll(Pageable)` : returns a Page of entities

Custom JPA Methods (Spring Data JPA)

In all *@Repository*, we can create a "**custom query methods**". Spring automatically generates sql queries using the method name.

For example, this method will automatically generate this query:

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {

  List<User> findUserByNameAndBirthDate(String name, LocalDate birthDate);

  // result -> select u from User u where u.name = ? and u.birthdate = ?;

}
```

Observe, we created the methods using the names of the attributes of our 'User' entity. In this case, we want to use "and" in our query.

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {

    List<User> findUserByNameAndBirthDate(String name, LocalDate birthDate);

    // result -> select u from User u where u.name = ? and u.birthdate = ?;

}
```

```
@Entity
public class User {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String surname;
    private LocalDate birthDate;
```

It is possible to create queries with the most common SQL statements.

Spring Query Methods (from spring.io, always read through the official documentation)

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is`, `Equals`	`findByFirstname`, `findByFirstnameIs`, `findByFirstnameEquals`	`… where x.firstname = ?1`
`Between`	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual`	`… where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`

`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`, `Null`	`findByAge(Is)Null`	`… where x.age is null`
`IsNotNull`, `NotNull`	`findByAge(Is)NotNull`	`… where x.age not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike`	`… where x.firstname not like ?1`
`StartingWith`	`findByFirstnameStartingWith`	`… where x.firstname like ?1` (parameter bound with appended `%`)
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` (parameter bound with prepended `%`)
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` (parameter bound wrapped in `%`)
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)`	`… where x.age not in ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	`… where UPPER(x.firstname) = UPPER(?1)`

Setting up JPA with H2

Maven Dependencies

- Spring Boot provides **spring-boot-starter-data-jpa** dependency to connect Spring application with relational database efficiently.
- The **spring-boot-starter-data-jpa** internally uses the **spring-boot-jpa** dependency.
- In this chapter, we'll be using the **H2 in-memory database** for demonstration purpose.

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

<dependency>
```

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

application.yml

- To connect to a relational database, we need to configure the data source. This can be done by defining properties in the `application.yml` file.

```yaml
# Data Source Settings
spring.datasource:
  platform: h2
  url: jdbc:h2:mem:jpademo

# JPA Settings
spring.jpa:
  show-sql: true
  hibernate.ddl-auto: create-drop # for development purpose

# H2 Database Settings
spring.h2.console.enabled: true
```

- Setting `spring.jpa.hibernate.ddl-auto` to *create-drop* will create new tables upon application startup and drop tables upon closing every time. This is particularly useful for developement and testing purposes. We will discuss more on this property in the next chapter.

Entity

```java
@Entity
public class Book {

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

    private String author;
    private int page;
    private String title;

    // Constructors, getters, and setters
```

```
}
```

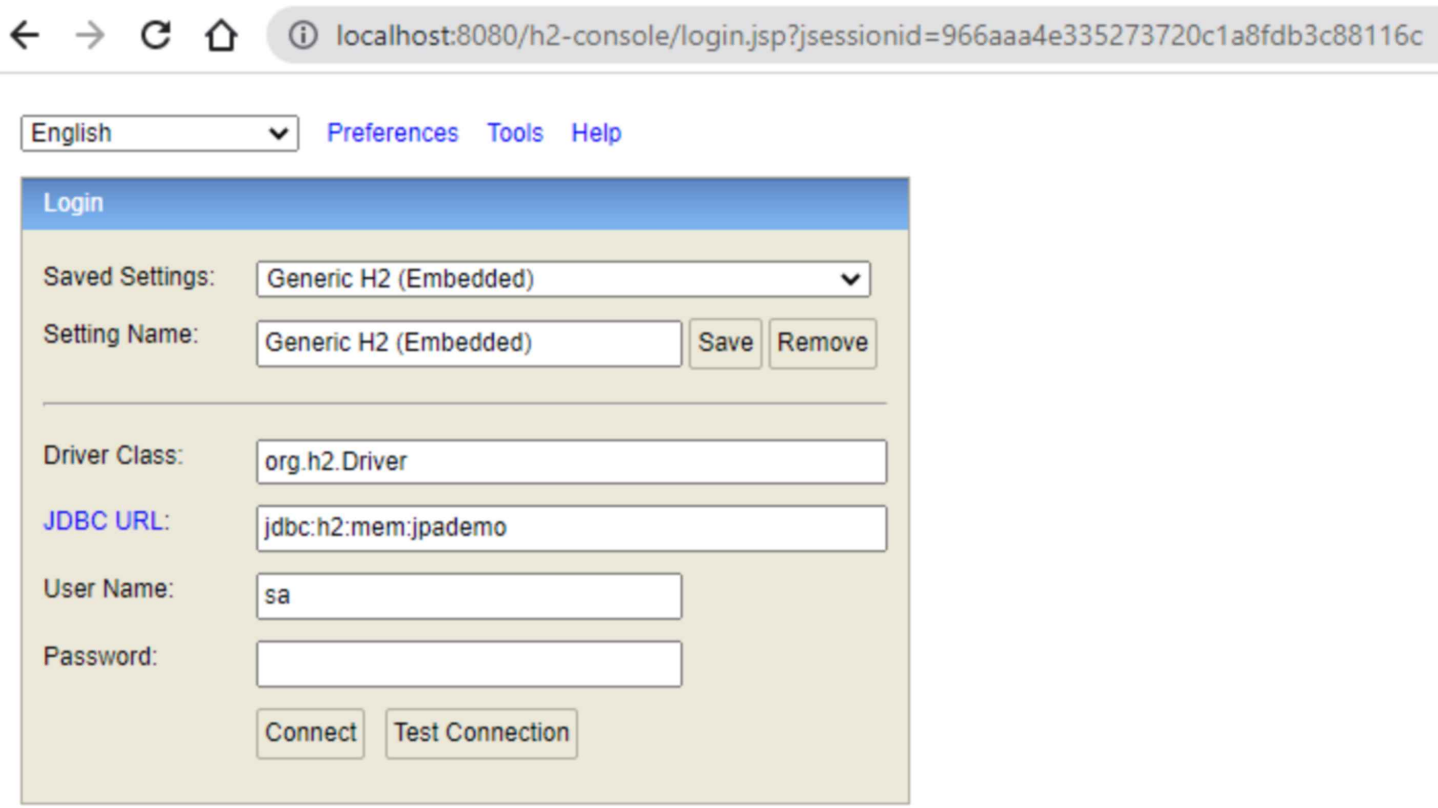
Repository

```
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
    // You can define custom query methods here if needed
}
```

H2 Console

- Since we have enabled h2-console in `application.yaml`, we can access the H2 console UI by typing `http://localhost:8080/h2-console` in the browser.



- We can perform regular SQL queries just like in other Database Product.

localhost:8080/h2-console/login.do?jsessionid=042e329c0ecfbc16135c293eb4c89492

Auto commit | Max rows: 1000 | Auto complete Off | Auto select On

- jdbc:h2:mem:jpademo
  - BOOK
    - ID
    - AUTHOR
    - PAGE
    - TITLE
    - Indexes
  - INFORMATION_SCHEMA
  - Sequences
  - Users
- H2 1.4.200 (2019-10-14)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM book
```

SELECT * FROM book;

ID	AUTHOR	PAGE	TITLE
1	Craig Walls	521	Spring in Action
2	Craig Walls	266	Spring Boot in Action
3	Bruce Eckel	1079	Thinking in Java
4	Bruce Eckel	1778	On Java 8
5	Joshua Bloch	413	Effective Java

(5 rows, 5 ms)

Edit