COURSEWARE

Repositories

Contents

Overview

Repositories provide methods for interacting with a database.
In Spring they take the form of interfaces that extend either `CRUDRepository` or `JPARepository` and are annotated with `@Repository`.

```
@Repository
public interface UserRepo extends JpaRepository<User, Long> {

}
```

At runtime Spring will create an instance of `UserRepo`.
By extending `JPARepository` this instance will inherit all of the basic CRUD functionality.

As you can see, the `JPARepository` being extended is of type `User, Long`, with `User` being the type of the `Entity` and `Long` the type of the `id`.

Repository Methods

Spring repositories are extremely useful, both for the selection of basic methods that are provided with them and the intuitive way new ones can be added.

save()

Persists an entity to the database.
If the entity does not have an id then Spring assumes that it is a new entity and adds it to the table - if the entity *does* have an id then Spring will try to update the existing record.

findAll()

Returns all of the entities as a `List`.

findByID()

Takes in an id and attempts to return a matching entity as an `Optional`.

removeByID()

Takes in an id and attempts to remove a matching entity.

Tutorial

We'll start off by adding a `PersonRepo` to the previous example.

Creating the Repository

First, create the interface:

```
public interface PersonRepo {

}
```

Then annotate it as a `@Repository`:

```
@Repository
public interface PersonRepo  {

}
```

Make it extend `JpaRepository`:

```
public interface PersonRepo extends JpaRepository<T, ID> {

}
```

Finally, set the type of the `Entity` and `id` - remember, these will be the name of the `Entity` class and they type of its `id` field, e.g.

```
@Entity
public class Person { // <- Type of Entity

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // <- Type of id

    // cont.
```

```
@Repository
public interface PersonRepo extends JpaRepository<Person, Long> {

}
```

And that's it for the repo!
With this we'll be able to start persisting our data; but before that we'll need to update our `PersonService` class to use our new repo rather than a `List`.

Updating the service

First things first, let's add the `PersonRepo` dependency into our `PersonService`

```
@Service
public class PersonService {

    private PersonRepo repo;

    public PersonService(PersonRepo repo) {
        super();
        this.repo = repo;
    }

    // CRUD methods
}
```

Now we can update our CRUD methods with our new repo.

Create

Start off with the method structure

```
    public Person addPerson(Person person) {

    }
```

In order to persist the `Person` to the database we will need to use the `save` method; this will write a new entry to the database and then read it back out so the `Person` we get back should be exactly the same as the one we put in *except* this one will have an auto-generated `id`.

```
    public Person addPerson(Person person) {
        return this.repo.save(person);
    }
```

Read

Starting off with the method structure again, the read method will require no parameters and will return a `List` of people.

```
    public List<Person> getAllPeople() {

    }
```

To finish implementing this we can simple use the `findAll` method in our `PersonRepo`

```
    public List<Person> getAllPeople() {
        return this.repo.findAll();
    }
```

Update

Our update method will take in two parameters: the id of the `Person` to be updated and a `Person` object containing the new data.

```
    public Person updatePerson(Long id, Person newPerson) {

    }
```

In order to update an existing record the first step is to fetch it from the database using `findById` - remember that `findById` returns an `Optional` so if we want to access the actual `Person` we will need to use a suitable `Optional` method to extract it.

```
    public Person updatePerson(Long id, Person newPerson) {
        Optional<Person> existingOptional = this.repo.findById(id);
        Person existing = existingOptional.get();
    }
```

Now we can update the existing `Person` with the new data - make sure NOT to change the id!

```
    public Person updatePerson(Long id, Person newPerson) {
        Optional<Person> existingOptional = this.repo.findById(id);
        Person existing = existingOptional.get();

        existing.setAge(newPerson.getAge());
        existing.setName(newPerson.getName());
    }
```

Finally, we can simply save the `Person` back to the database and the JPA will update the record with our new data.

```
    public Person updatePerson(Long id, Person newPerson) {
        Optional<Person> existingOptional = this.repo.findById(id);
        Person existing = existingOptional.get();

        existing.setAge(newPerson.getAge());
        existing.setName(newPerson.getName());

        return this.repo.save(existing);
    }
```

Delete

All we need to remove a `Person` is its id and all we need back is a boolean showing whether the delete succeeded so the method structure should look like this:

```
    public boolean removePerson(Long id) {

    }
```

Removing a record from the database is simple using `deleteById`

```
    public boolean removePerson(Long id) {
        this.repo.deleteById(id);
    }
```

The only issue with this method is that `deleteById` is a `void` method so we'll need to use something else to check the entity no longer exists, which is where `existsById` comes in!

```
    public boolean removePerson(Long id) {
        // removes the entity
        this.repo.deleteById(id);
        // checks to see if it still exists
        boolean exists = this.repo.existsById(id);
        // returns true if entity no longer exists
        return !exists;
    }
```

And that should be it! Now simply test it out in Postman.

Exercises

1. Add an `AccountRepo` to the project you first made in the Entities module.
2. Add the new repo to your `AccountService` as a dependency and update the CRUD methods to use it.