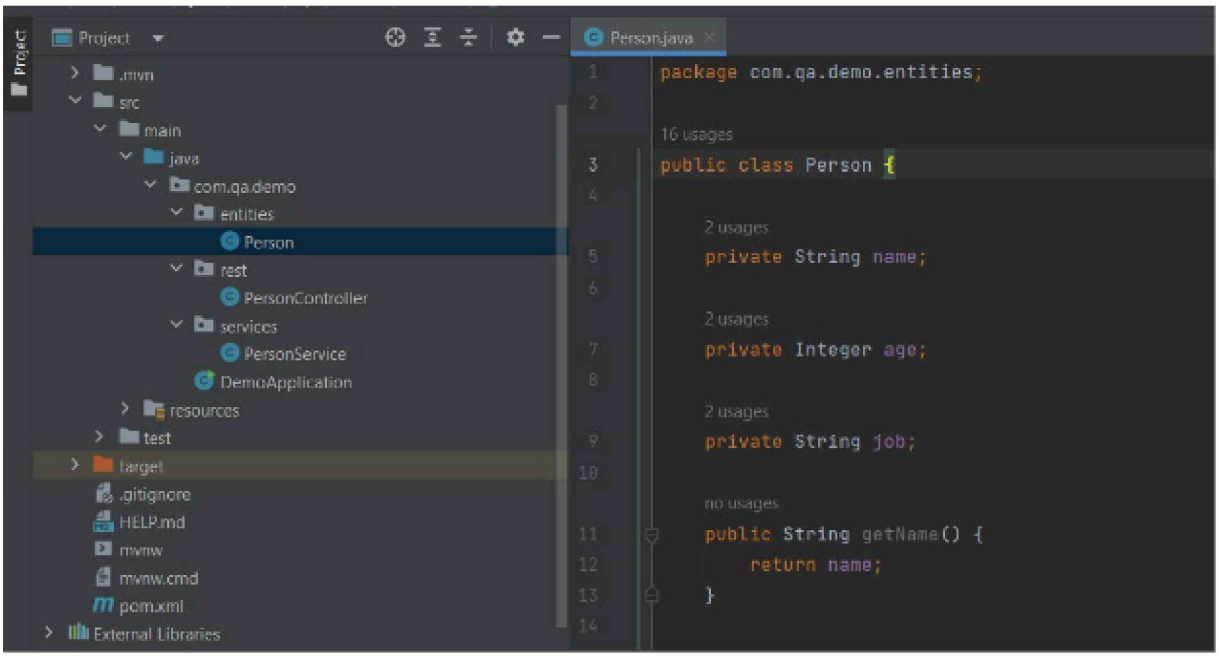
Lab 4 – Persistence

The purpose of this task is to set up a Person entity and repo so we can persist our person data to an H2 database.

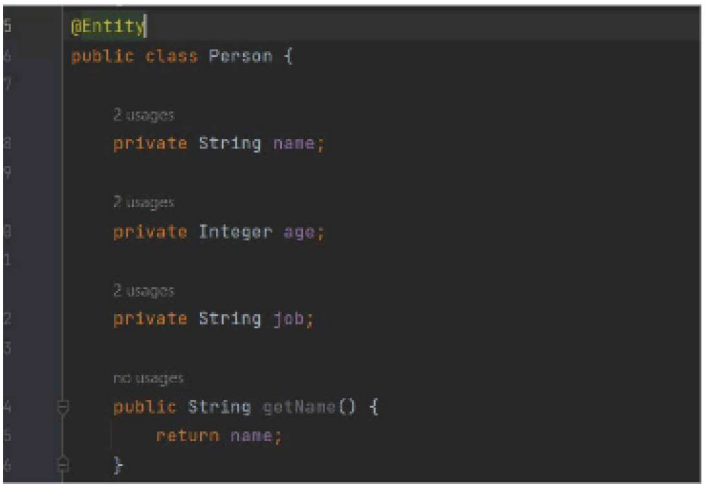
Task 1 – Creating the entity

1. Open the project you created in the previous lab.

2. Find the Person class.



3. Annotate this class as an **@Entity.** This will tell Spring onto the Person table in the database.



4. Add an Integer id field with a getter and setter.

```
@Entity
public class Person {

    2 usages
    private Integer id;
    2 usages
    private String name;

    2 usages
    private Integer age;

    2 usages
    private String job;

    no usages
    public Integer getId() {
        return id;
    }

    no usages
    public void setId(Integer id) {
        this.id = id;
    }
```

5. This field represents the primary key so annotate it as an **@Id**.

```
16 usages
@Entity
public class Person {

    2 usages
    @Id
    private Integer id;
```

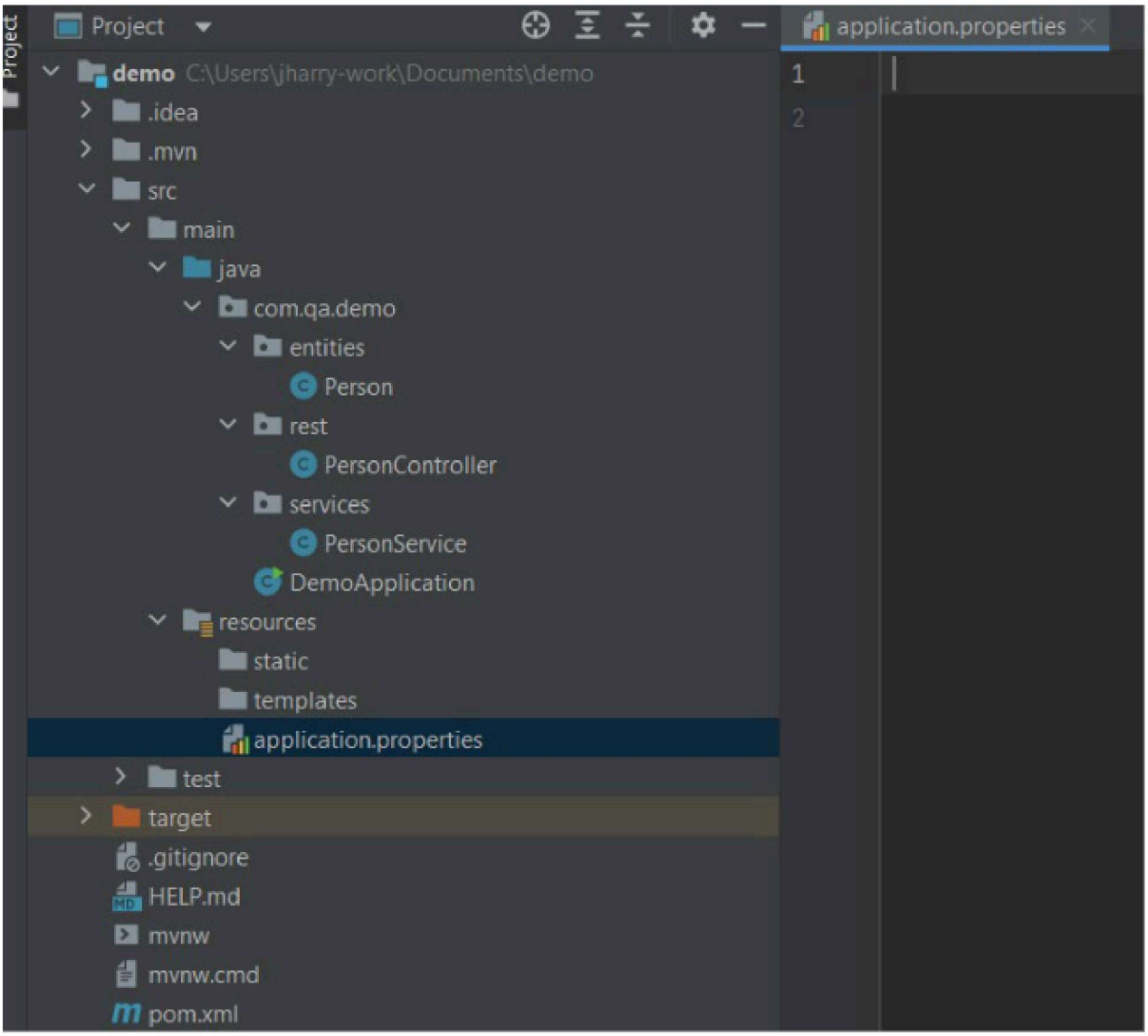
6. For convenience, make the key auto incremented using **@GeneratedValue.**

7. At this point, all our fields with getters and setters will be automatically mapped to columns in the person table by Spring. We can use **@Column** to configure these columns; for example, by changing the column name and applying a couple constraints

Task 2 – Connecting to the database

1. Now we need to connect our Spring app to a database. When we created the app we added H2 as a dependency. Spring will see the H2 dependency on the build path and use it to setup an in memory database which we can configure using **application.properties**.

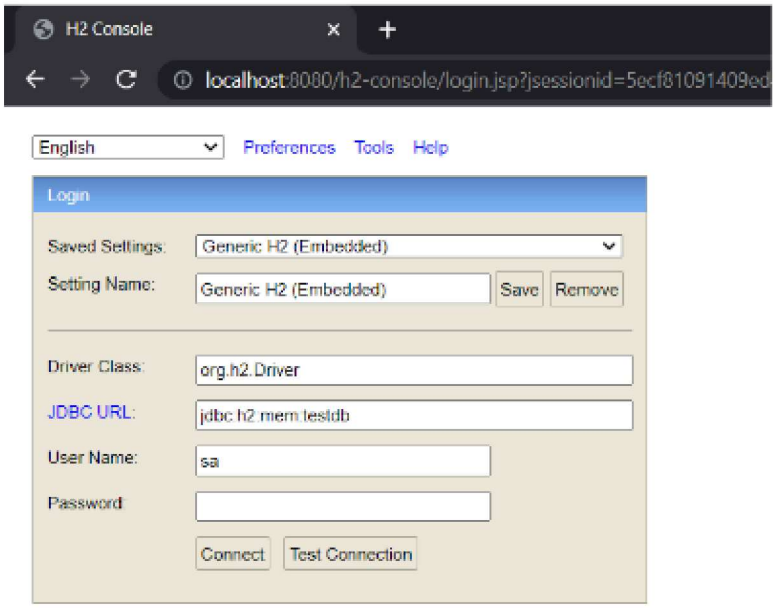
2. Open the properties file in **src/main/resources**



3. Add this snippet to your **application.properties**

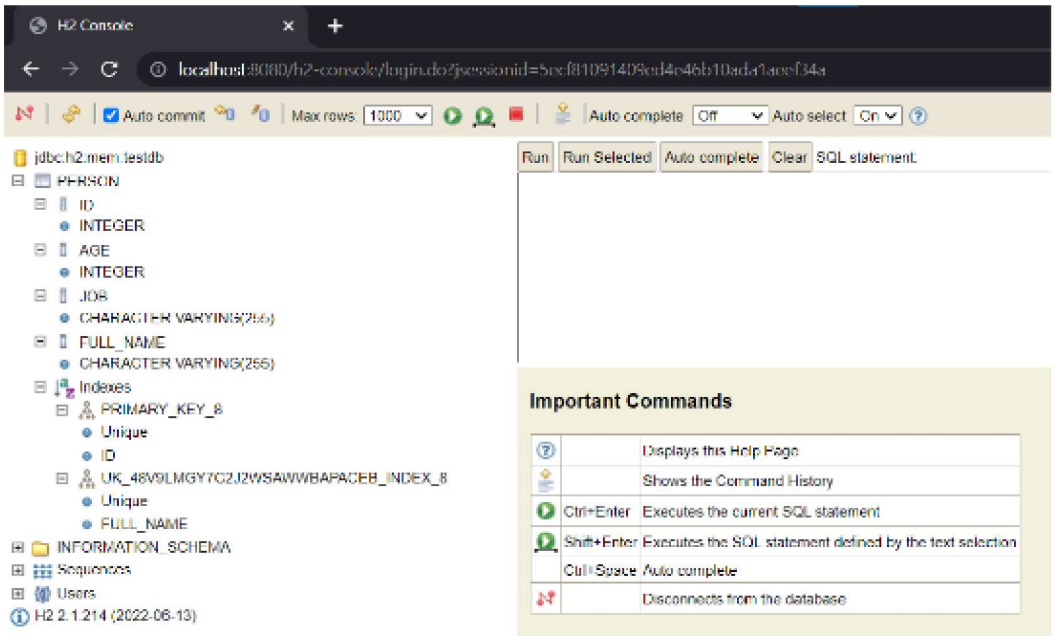
```
# Set the db URL
spring.datasource.url=jdbc:h2:mem:testdb
# Set the username to sa
spring.datasource.username=sa
# Set a blank password
spring.datasource.password=
```

4. You can view the database at http://localhost:8080/h2-console By default you'll see the below screen - make sure the JDBC URL matches what was entered in the previous step (jdbc:h2:mem:testdb)



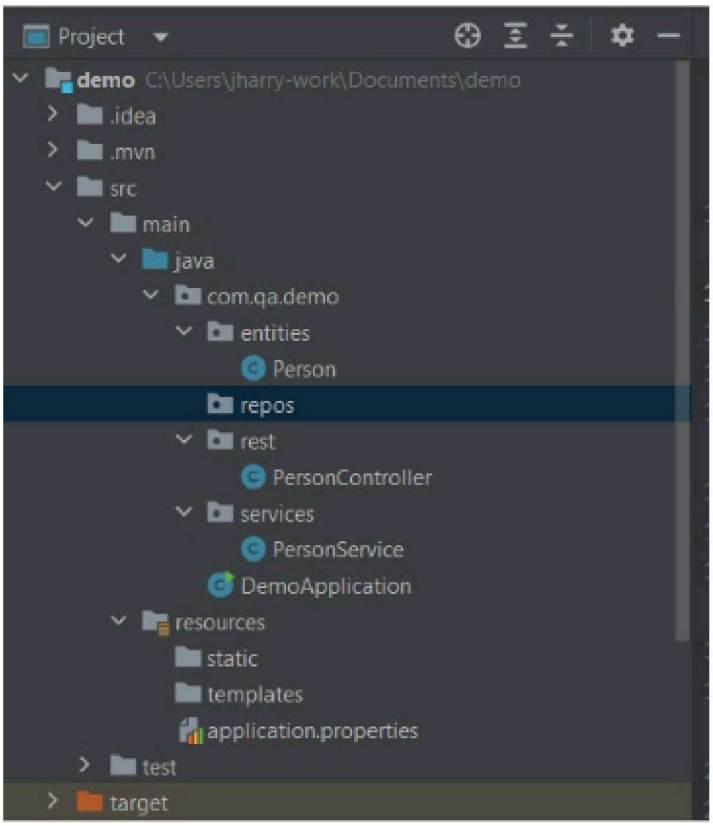
(If you can't see this menu, try adding **spring.h2.console.enabled=true** to your properties file).

5. Clicking **Connect** takes you into the H2 console itself, where you can see the table we configured in Task 1.

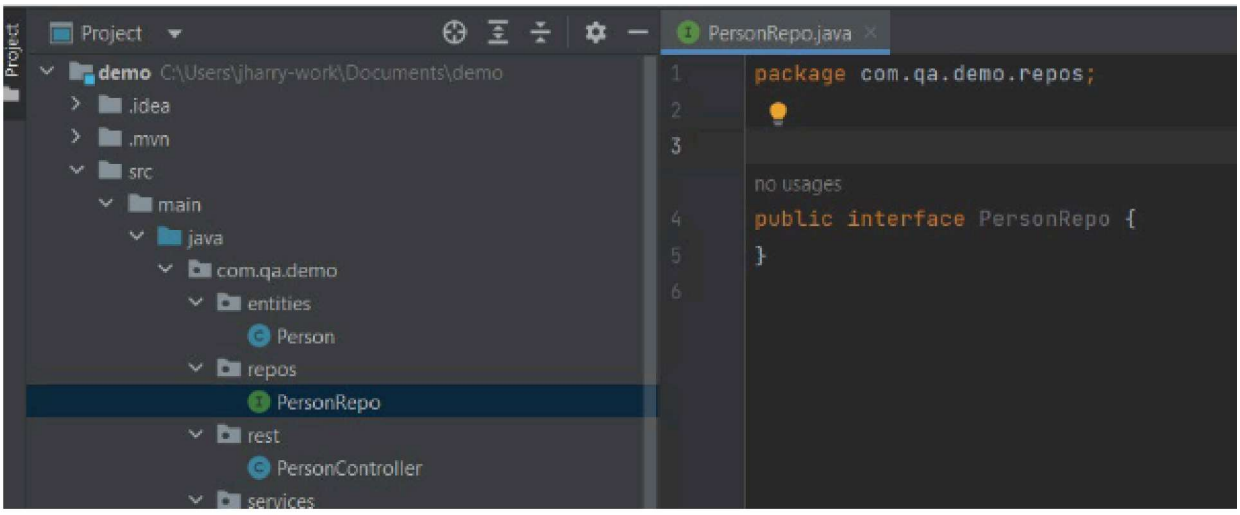


Task 3 – Creating the repo

1. Create a new package in **com.qa.demo** called **repos**.



2. Add a **PersonRepo** interface to the new package



3. Alter your inteface to extend **JpaRepository.**

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

no usages
public interface PersonRepo extends JpaRepository<T, ID> {
}
```

4. The **JpaRepository** has two generics; the first is for the type of the entity (Person) and the second is the type of the @Id field *in* Person (Integer). Update your interface with these types.

```
3 usages
public interface PersonRepo extends JpaRepository<Person, Integer> {
}
```

5. Finally, annotate the interface with **@Repository.**

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

Task 4 – Updating the service

1. Now that the repo exists, we can use it in the **PersonService** instead of the List. Start by deleting the List from the service and injecting the **PersonRepo** instead.

```
@Service
public class PersonService {

    1 usage
    private PersonRepo repo;

    no usages
    public PersonService(PersonRepo repo) {
        this.repo = repo;
    }

    1 usage
    public List<Person> getAll() { return this.people; }
```

2. Check each method, removing the bodies and making them return null. (This is so that the app compiles as we add the functionality.)

```
1 usage
public List<Person> getAll() { return null; }

1 usage
public Person get(int id) { return null; }

1 usage
public Person createPerson(Person person) { return null; }

1 usage
public Person updatePerson(int id, String name, Integer age, String job) { return null; }

1 usage
public Person removePerson(int id) { return null; }
```

3. Alter the **getAll** method to use the **findAll** method from your repo.

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

4. Do the same for **get** with **findById**

```
3 usages
public Person get(int id) { return this.repo.findById(id); }
```

5. As you can see this line causes an error. This is because **findById** returns an Optional<Person> instead of just a Person object. The simplest way to resolve this is to add **.get()** after the **findById**.

Notably this *will* cause an error if you request an id that is not in the table – you could potentially resolve this using **orElse** or **orElseThrow** instead but **get** is sufficient for our needs.

```
3 usages
public Person get(int id) { return this.repo.findById(id).get(); }
```

6. For **createPerson** we will use the **save** method from the repo to persist our new person to the database

```
1 usage
public Person createPerson(Person person) {
    return this.repo.save(person);
}
```

7. Updating an existing entity requires a couple extra steps; first you need to fetch it from the data base using the id, then update the fields with the new data (same way as with the list) and then persist the changes using the **save** method

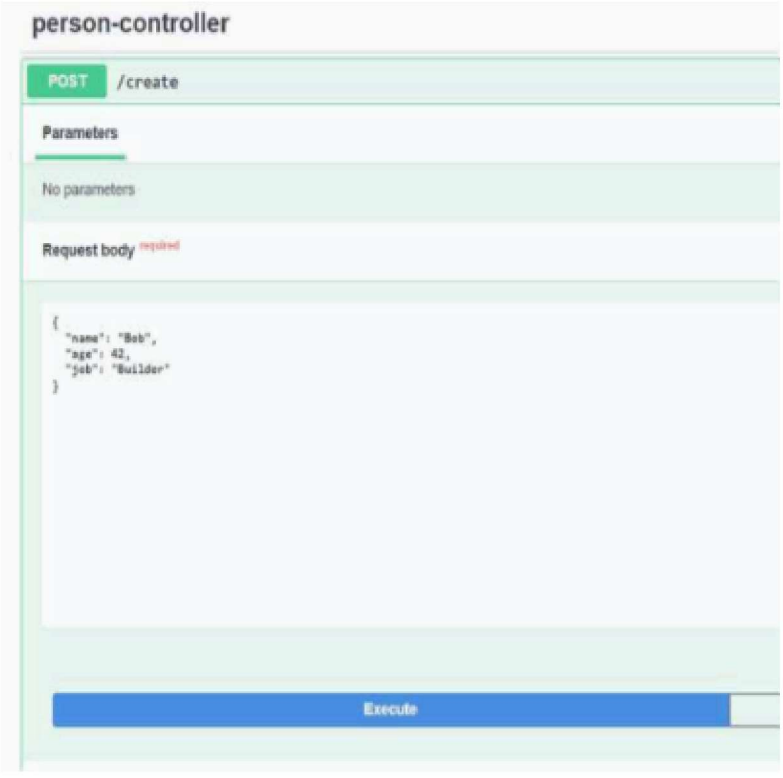
```
1 usage
public Person updatePerson(int id, String name, Integer age, String job) {
    Person toUpdate = this.get(id);

    if (name != null) toUpdate.setName(name);
    if (age != null) toUpdate.setAge(age);
    if (job != null) toUpdate.setJob(job);

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

8. For our final method, we can simply use **deleteById** to to remove the person from the database. However, because **deleteById** is a **void** method, we'll first fetch the person with that id so we can return it after the delete operation.

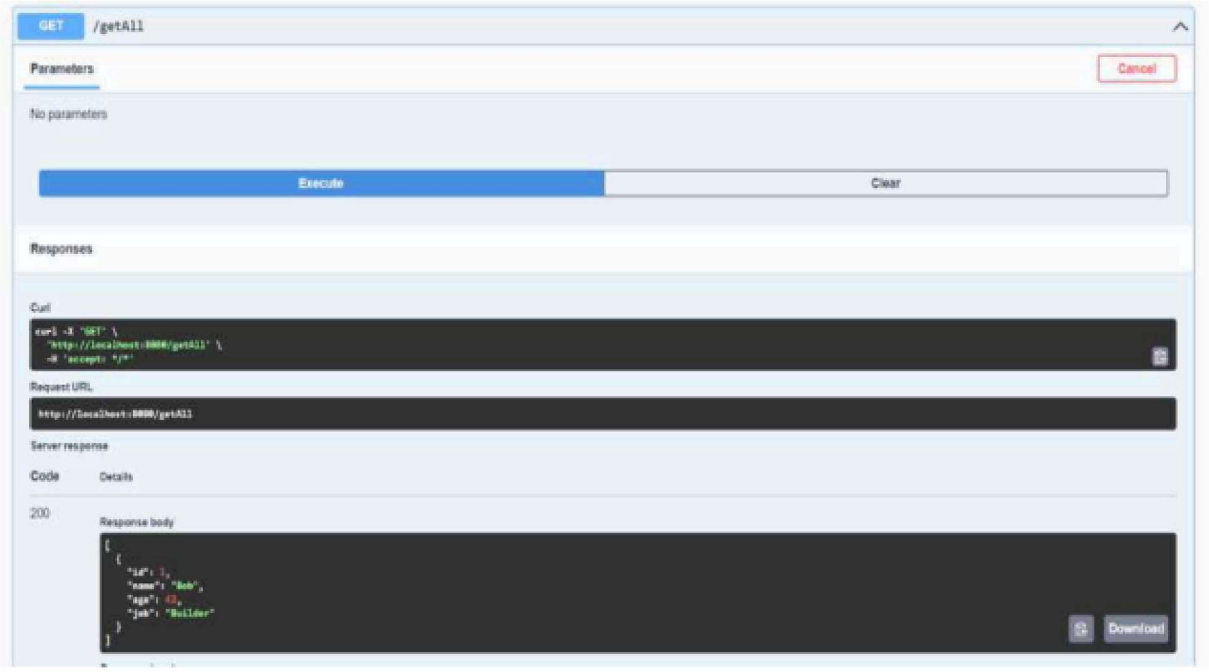
```
1 usage
public Person removePerson(int id) {
    Person removed = this.get(id);
    this.repo.deleteById(id);
    return removed;
}
```

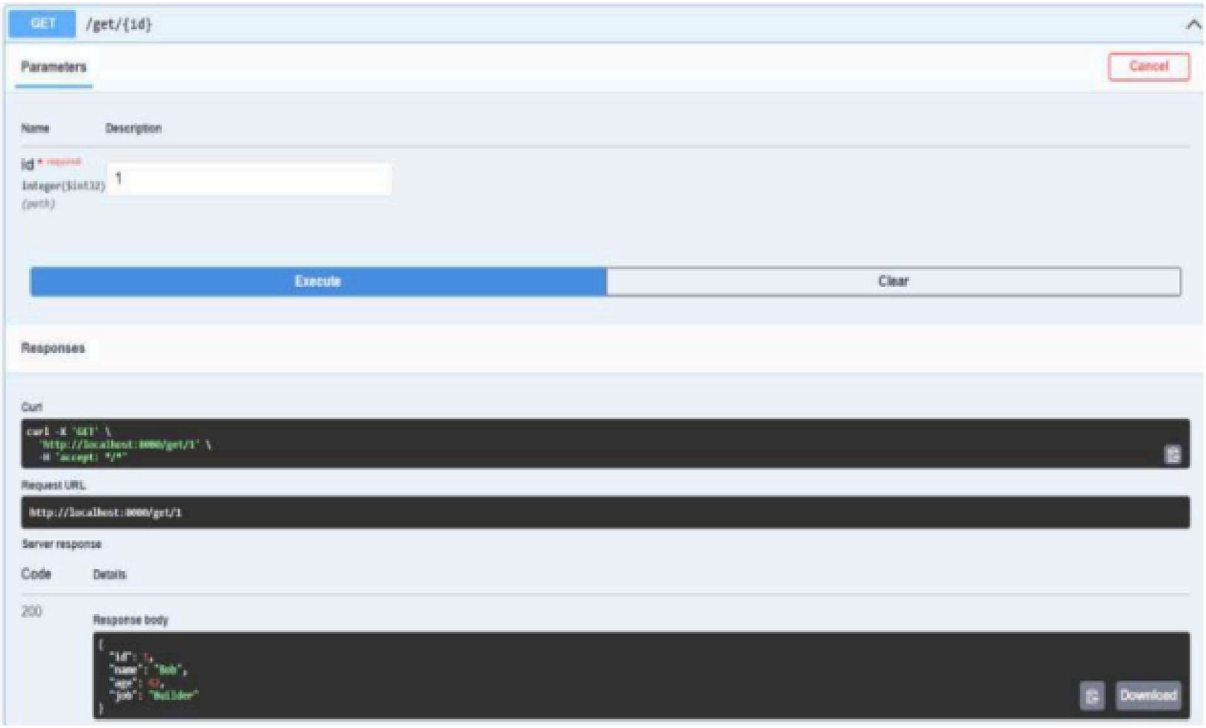
9. Now go back to the swagger docs and check your CRUD functionality still works – remember that we're using a table now, not a List, so ids will start at 1, not 0.

10. Testing POST: (Notice that the created person has an id of 1 even though there's no id in the request body – this is due to the **@GeneratedValue** annotation we added)

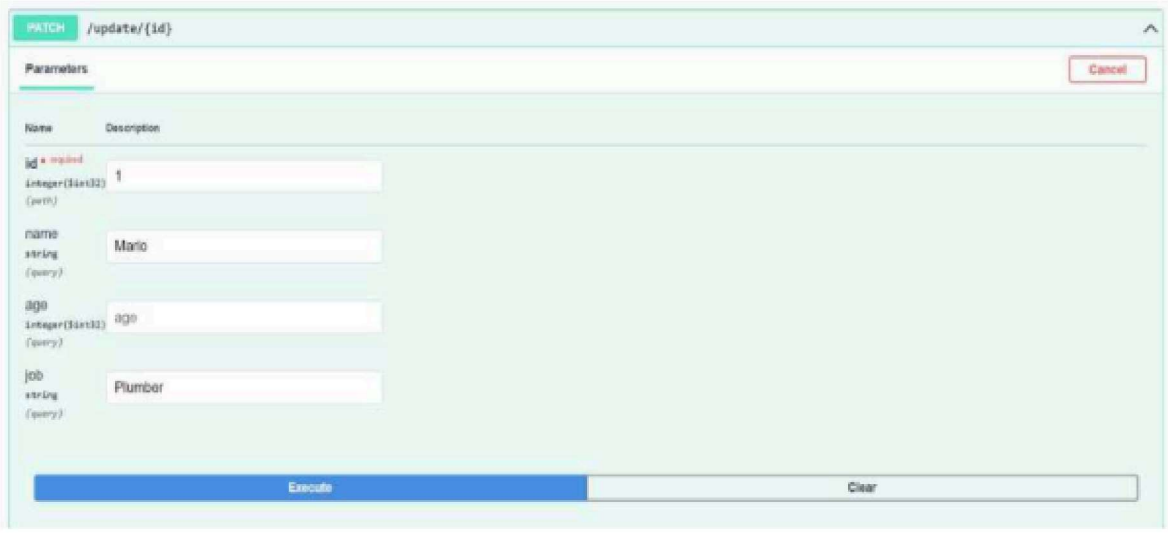


11. **Testing GET:**

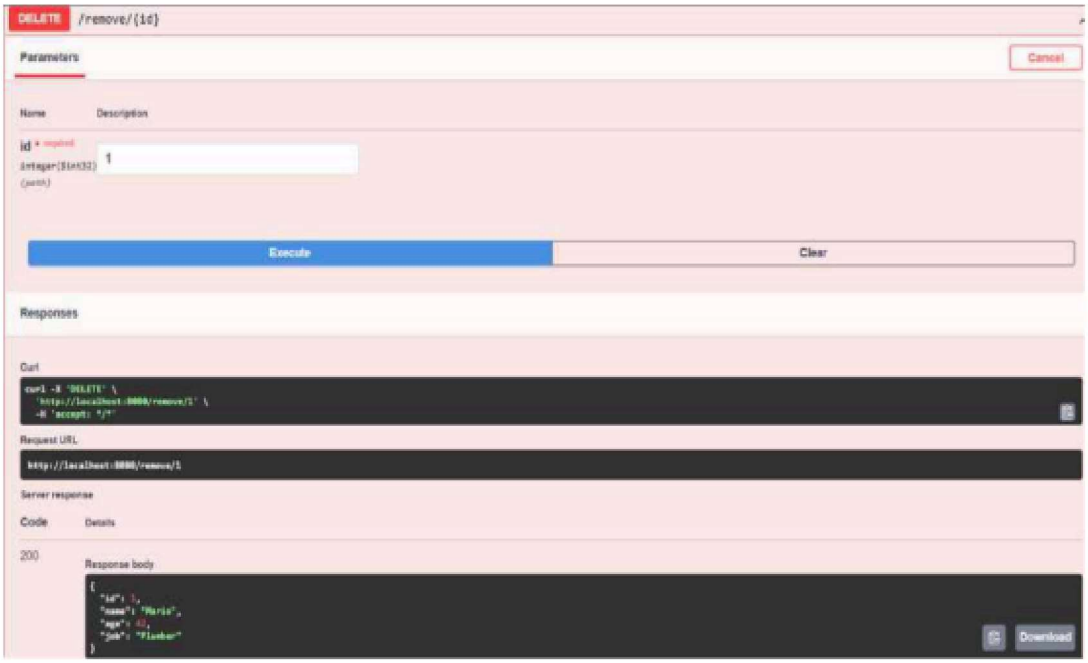


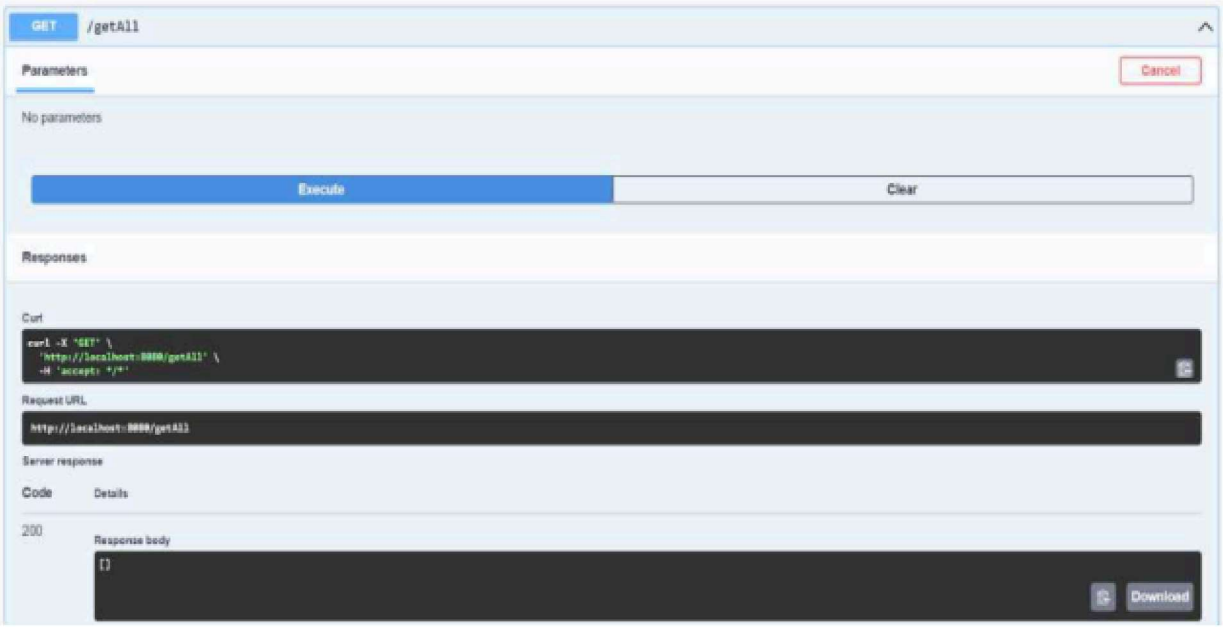


12. **Testing PATCH:**



13. **Testing DELETE:**




GET
/getAll
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:8080/getAll' \
-H 'accept: */*'
Request URL
http://localhost:8080/getAll
Server response
Code
Details
200
Response body
[]
Download