POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEI Departamento Engenharia Informática

Desenvolvimento de Aplicações Empresariais – 2021-22-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 3

Topics: JPA Entity annotations, *one to many*/*many to one* Entity relationships.

1. As you already have noticed, by default, database tables names are equal to the corresponding entity's name. However, usually, we use plural names for database tables. In order to define a different table name, we annotate entities with the `@Table` annotation. Annotate the `Student` entity with the annotation `@Table(name="STUDENTS")`. From now on, do this for all entities;
2. We may also want to define some validations to entities' attributes at the entities level (not only at business logic and/or client applications/presentation logic layer level). For example, we do not want any of the `Student` entity attributes to hold *null* values. Annotate the `password`, `name` and `email` attributes of the `Student` entity with the `@NotNull` annotation (why we do not need to do this for the *username* attribute?).
3. Annotate the `email` attribute of the `Student` entity with the @Email annotation so that the email format is validated.
4. Make sure your docker engine is running and run the application (`make deploy`). Confirm in your database that there is an additional table called `students` (`make sql`, followed by `\dt`), and drop the "old" one `student` (`drop table student;`);
5. Create the `Course` entity with the attributes: `code` (the Id, of *int* type), `name`, and `students` (a *List* of *Students* – ignore the error that this declaration may give you, by now). Add the appropriate constructors, getter and setter methods. Do not forget to initialize the students list in both constructors (the constructor with arguments does not receive the students list). Add the methods `addStudent(…)` and `removeStudent(…)` that, respectively, add and remove a given student from the students list;
6. Add the attribute `course` (of type `Course`) to the `Student` entity and the respective getter and setter methods (ignore the error that this declaration may give you, by now). Modify the constructor with arguments of this entity so that it also receives and sets the course attribute (this will lead to errors in the EJBs; you will soon fix this);
7. Annotate the `course` attribute in the `Student` entity with the following annotations:

```
@ManyToOne
@JoinColumn(name = "COURSE_CODE")
@NotNull
```

The `@ManyToOne` annotation states that there is a *many to one* mapping between the `Student` entity and the `Course` entity. The `@JoinColumn` annotation allows us to "set" the name of the column of the `STUDENTS` table corresponding to the foreign key of the `Course` entity. If we do not use this annotation, the name of the column corresponding to the foreign key of the *One* side of the relation will have the same name as the attribute's name (`COURSE`, in this case);

8. Annotate the `students` attribute of the `Course` entity with the following annotation:

```
@OneToMany(mappedBy = "course", cascade = CascadeType.REMOVE)
```

This annotation states that there is a *one to many* mapping between the `Course` entity and the `Student` entity. The `mappedBy` attribute of the annotation states what is the attribute's name of the *Many* side of the relation (the `Student` entity) that refers to the *One* side of the relation (the `Course` entity). The `cascade` attribute of the annotation allows us to define what operations are cascaded to the *Many* side instances of the relation when a CRUD operation is done over an entity of the relation's *One* side. In this example, we are stating that when we remove a course, all its students are removed too (more on this attribute on the Java EE Tutorial).

9. Annotate the `Course` entity with the following annotation:

```
@Table(
    name = "COURSES",
    uniqueConstraints = @UniqueConstraint(columnNames = {"NAME"})
)
```

In this case, the `@Table` annotation, besides "setting" the name of the database table corresponding to the `Course` entity, defines a constraint stating that there cannot be two courses with the same name (notice that the `name` attribute is not an Id one).

10. Create the named query `getAllCourses` that allows us to retrieve all the courses ordered by name (seek the former `getAllStudents` query in the `Student` entity for guidance);

11. Modify the `create(…)` method of the `StudentBean` EJB so that it now also receives the student's course code. It should first search for the corresponding course using the entity manager's `find(…)` method. If found, the student should be appropriately created (the found course should be included as a parameter to the `Student` entity constructor) and persisted. Make sure to also add the student to its `Course` list of students;

12. Create the `CourseBean` stateless EJB and add the `create(…)` and `getAllCourses(…)` methods to it;

13. Modify the `populateDB()` method of the `ConfigBean` EJB so that some courses are created before students, and then modify the calls to the `create(…)` method of the `StudentBean` EJB in order to define a course for each student;

14. Make sure your docker engine is running and run the application (`make deploy`). Confirm that a new table `courses` has come up in your database. Also, the `students` table should now have a `course_code` column;

15. Now we must update our Service Layer to be able to create new students and retrieve students with their course information, as well as a `Course` list. Update the `StudentDTO` to foresee 2 more attributes: `courseCode` and `courseName` (do not forget to update the arguments in the constructor and to include getters and setters for these new attributes);

16. Update the method `toDTO(…)` in the `StudentService` web service class to return also a student DTO with these 2 new attributes `courseCode` and `courseName`;

17. Create a method in the `StudentBean` class to find a student, given her/his username:

```
public Student findStudent(String username) {
    return em.find(Student.class, username);
}
```

18. Add a new web service method to the `StudentService` class to create new students:

```
@POST
@Path("/")
public Response createNewStudent (StudentDTO studentDTO){
    studentBean.create(studentDTO.getUsername(),
        studentDTO.getPassword(),
        studentDTO.getName(),
        studentDTO.getEmail(),
        studentDTO.getCourseCode());
    Student newStudent = studentBean.findStudent(studentDTO.getUsername());
    if(newStudent == null)
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
    return Response.status(Response.Status.CREATED)
                .entity(toDTO(newStudent))
                .build();
}
```

19. Make sure your docker engine is running, and run your application (`make deploy`);

20. Test this and the remaining web service methods by creating a new "HTTP Request" file named "HTTPRequests", and writing the following HTTP requests:

```
##### Students
GET http://localhost:8080/academics/api/students
Accept: application/json
###
POST http://localhost:8080/academics/api/students
Content-Type: application/json

{ "email": "johndoe@mail.com",
  "name": "John Doe",
  "username": "john",
  "password": "jd",
  "courseCode":"1",
  "courseName": "EI"
}
```

21. Run these requests (one by one) against your web services, and verify the results;

22. Based on the code examples you have made for students, develop all the necessary classes and methods to create a `CourseService` web service which returns a list of courses with their names and course codes. Write a HTTP request to test the implemented service;

23. Open the `academics-client` NUXT project.

24. Edit the index.vue page, to have a link to a new page that creates a student. To do that, change the content inside the <template>…</template> tag to this:

```
<template>
  <div>
      <b-container>
        <b-table striped over :items="students" :fields="fields"/>
      </b-container>
      <nuxt-link to="/create">Create a New Student</nuxt-link>
  </div>
</template>
```

25. Create a new file called `create.vue`, inside the pages directory.

26. Copy this content to the file:

```
<template>
  <form @submit.prevent="create">
    <div>
       username: <input v-model="username" type="text">
    </div>
    <div>
      password: <input v-model="password" type="password">
    </div>
    <div>
      name: <input v-model="name" type="text">
    </div>
    <div>
      email: <input v-model="email" type="email">
    </div>
    <div>
      course code:
      <select v-model="courseCode">
        <template v-for="course in courses">
          <option :key="course.code" :value="course.code">
            {{ course.name }}
          </option>
        </template>
      </select>
    </div>

    <nuxt-link to="/">Return</nuxt-link>
      <button type="reset">RESET</button>
      <button @click.prevent="create">CREATE</button>
  </form>
</template>

<script>
  export default {
    data() {
      return {
        username: null,
        password: null,
        name: null,
        email: null,
        courseCode: null,
        courses: []
      }
    },

  created() {
    this.$axios.$get('/api/courses')
      .then(courses => {
        this.courses = courses
    })
  },

  methods: {
    create() {
    this.$axios.$post('/api/students', {
      username: this.username,
```

```
        password: this.password,
        name: this.name,
        email: this.email,
        courseCode: this.courseCode
      })
      .then(() => {
        this.$router.push('/')
      })
      }
    }
  }
  </script>
```

27. Run the academics-client NUXT project and test to create a student.

Homework: you can now also write all the code necessary to list, create, consult, update and remove courses…

Note: you may/should consult the Order project that comes with the Java EE Tutorial.

Bibliography

Java EE Tutorial 8 (all of it).