POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEI Departamento Engenharia Informática

Desenvolvimento de Aplicações Empresariais – 2020-21-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 4

Topics: `one to many`/`many to one` and `many to many` entity mappings; entities and inheritance.

1. Create the `Subject` entity with the attributes `code` (the `@Id` of `int` type), `name`, `course`, `courseYear`, `scholarYear`, and `students` (a list/set of Students). Add the appropriate constructors, getter and setter methods. Do not forget to initialize the students' list/set (for instance, with a `LinkedHashSet<>` to avoid duplicates) in both constructors (the constructor with arguments does not receive the students' set). Add the methods `addStudent(…)` and `removeStudent(…)` that, respectively, add and remove a given student from the Subject's students list/set;
2. Use the @Column annotation to define the name of the table columns corresponding to attributes `courseYear` and `scholarYear` as "`COURSE_YEAR`" and "`SCHOLAR_YEAR`", respectively.
3. Add the attribute `subjects` to the `Course` entity. Modify appropriately the `Course` entity's constructors and add the appropriate getter and setter methods, as well as the `addSubject(…)` and `removeSubject(…)`;
4. Add the attribute `subjects` to the `Student` entity. Modify appropriately the `Student` entity's constructors and add the appropriate getter and setter methods, as well as the `addSubject(…)` and `removeSubject(…)`;
5. Add the annotations that allow to define properly the mapping between the `Course` and `Subject` entities (one to many/many to one);
6. There is a many to many mapping between the `Subject` and `Student` entities. This type of mapping implies the creation of a third (join) table in the database, besides the tables corresponding to the entities. Annotate the `students` attribute of the `Subject` entity as follows:

```
@ManyToMany
@JoinTable(name = "SUBJECTS_STUDENTS",
 joinColumns = @JoinColumn(name = "SUBJECT_CODE", referencedColumnName = "CODE"),
 inverseJoinColumns = @JoinColumn(name = "STUDENT_USERNAME", referencedColumnName =
"USERNAME"))
```

Now, annotate the `subjects` attribute in the `Student` entity with the following annotation:

```
@ManyToMany(mappedBy = "students")
```

7. Write the named query "getAllSubjects" that returns a list of subjects ordered first by `course name`, then by `scholarYear` (descending order), then `courseYear`, then `name`;

8. Annotate the `Subject` entity with the `@Table` annotation and define a unique constraint so that there are no two table lines with the same combination of name, course code and scholar year;

9. Create the `SubjectBean` stateless EJB and add the `create(…)` and getA`llSubjects()` methods to it;

10. Add the `enrollStudentInSubject(…)` method to the `StudentBean` EJB, which takes a username and a subject code, and enrolls the corresponding student in that subject;
Note: In order to do this properly, you should add the `equals(…)` and `hashcode()` methods to the `Course` entity. Why?

11. Modify the `populateDB()` method of the `ConfigBean` EJB so that some subjects are created and added to courses. Enroll some students in some subjects;

12. Make sure your database engine is running, run and test the application. Confirm that all data from the `populateDB()` method has been populated in the database;

13. Go to the `StudentService` web service and add the following method to retrieve a student's details:

```
@GET
@Path("{username}")
public Response getStudentDetails(@PathParam("username") String username) {
    Student student = studentBean.findStudent(username);
    if (student != null) {
        return Response.ok(toDTO(student)).build();
    }
    return Response.status(Response.Status.NOT_FOUND)
            .entity("ERROR_FINDING_STUDENT")
            .build();
}
```

14. Create the SubjectDTO with the attributes `code`, `name`, `courseCode`, `courseName`, `courseYear` and `schoolarYear`. Add constructors, getter and setter methods;

15. Add the `toDTO` and `toDTOs` methods to the `StudentService` class for converting `Subject` entities into `SubjectDTO` instances;

16. Add another web service method to the `StudentService` class to return the list of subjects of a certain student, as listed below:

```
@GET
@Path("{username}/subjects")
```

```
public Response getStudentSubjects(@PathParam("username") String username) {
    Student student = studentBean.findStudent(username);
    if (student != null) {
        var dtos = subjectsToDTOs(student.getSubjects());
        return Response.ok(dtos).build();
    }
    return Response.status(Response.Status.NOT_FOUND)
            .entity("ERROR_FINDING_STUDENT")
            .build();
}
```

17. Add new GET HTTP requests to your HTTP requests file to test the `getStudentDetails(…)` and the `getStudentSubjects(…)` web service methods above. Run these new requests;

18. In the last lectures, we had the students page (`index.vue`) under the `pages` folder. Now, as we can see, we will have students, courses and subjects. So, we need to separate these areas. Create a new folder, under the `pages` directory, called `students`. Move your index.vue in the `pages` folder to `pages/students`. Create a new `index.vue` within the `pages` folder with some content:

```
<template>
  <b-container>
      <h1>Welcome to Academics Management</h1>
    Please visit our
    <nuxt-link to="/students" class="btn btn-link">Students</nuxt-link>
  </b-container>
</template>
```

This will be our welcome page, with a link to visit `/students`.

19. In the `students/index.vue`, add a new field called `actions`:

```
data() {
    return {
      fields: ['username', 'name', 'email', 'courseName', 'actions'],
     // ...
    }
  },
```

20. Change the table and define a template that says how to render the action:

```
…
<b-table striped over :items="students" :fields="fields">
  <template v-slot:cell(actions)="row">
     <nuxt-link
       class="btn btn-link"
       :to="`/students/${row.item.username}`">Details</nuxt-link>
  </template>
</b-table>
<nuxt-link to="/">Back</nuxt-link>
…
```

21. Inside the `pages/students`, create a new page named "`_username.vue`". Pay attention that the underscore is very important! When you define a page called `_<foo>.vue`, this will translate into a URL parameter, called "foo". In our case, we want to get the details of some student. E.g.: `/students/aaa` should get the URL parameter called username with the value "aaa". But, if we change to `/students/bbb`, the value of the username parameter will change dinamically to "bbb".

22. Paste this content in pages/students/_username.vue file:

```
<template>
    <b-container>
        <h4>Student Details:</h4>
        <p>Username: {{ student.username }}</p>
        <p>Name: {{ student.name }}</p>
        <p>Email: {{ student.email }}</p>
        <p>Course: {{ student.courseName }}</p>

        <h4>Subjects enrolled:</h4>
        <b-table v-if="subjects.length" striped over :items="subjects"
:fields="subjectFields" />
        <p v-else>No subjects enrolled.</p>

        <nuxt-link to="/students">Back</nuxt-link>
    </b-container>
</template>
<script>
    export default {
        data() {
            return {
                student: {},
                subjects: [],
                subjectFields: ['code', 'name', 'courseCode', 'courseYear', 'scholarYear' ]
            }
        },
        computed: {
            username() {
                return this.$route.params.username
            }
        },
        created() {
            this.$axios.$get(`/api/students/${this.username}`)
                .then(student => this.student = student || {})
                .then(() => this.$axios.$get(`/api/students/${this.username}/subjects`))
                .then(subjects => this.subjects = subjects)
        },
    }
</script>
```

23. Make sure your database engine and Java EE application are running (make deploy), run and test the NUXT application.

We now want to add two more entities: The `Administrator` and the `Teacher` entities, which are intended to represent two other types of users of our enterprise application. Both these entities have, as the `Student` entity, the `username`, `password`, `name` and `email` attributes. So, it makes sense to create a `User` super class entity that the `Administrator`, the `Student` and the `Teacher` entities will extend from and where we will put what is common to all the users. Note: the Java EE Tutorial covers Entity Inheritance.

24. Create the `User` entity and move into it the `username`, `password`, `name` and `email` attributes from the `Student` entity, as well as the respective getter and setter methods; Modify the `Student` entity so that it now extends the User entity; Write the `User` entity constructors and adapt the `Student` entity ones;

```
@MappedSuperclass
@Table(name = "users")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
// Extra: try the other strategies… what happens to the database?
public class User {

    @Id
    private String username;

    @NotNull
    private String password;

    @NotNull
    private String name;

    @Email
    @NotNull
    private String email;


    // Default constructor ...


    // Constructor with all arguments ...


    // Getters and Setters ...
}
```

25. Create the `Administrator` entity which should extend the `User` entity. The `Administrator` entity does not have more attributes than the ones inherited from the `User` entity, so you just need to add the constructors;

26. Create the `Teacher` entity which should extend the `User` entity. Besides the attributes inherited from the `User` entity, this entity has the `office` and `subjects` (a list/set of subjects) attributes. Please note that there is a `many-to-many` mapping between teachers and subjects, so, write the appropriate code to reflect this;

27. Remove the `@Table` annotation from the Student entity. In the "new" database, all the users will be saved in a table USERS because we are using the "single table per class hierarchy" inheritance strategy (refer to the previous link to the Java EE Tutorial for more about entity inheritance mapping strategies);

28. Create the `AdministratorBean` stateless EJB and add it the `create(…)` method.

29. Create the `TeacherBean` stateless EJB and add it the `create(…)` method.

30. Modify the `populateDB()` method of the `ConfigBean` EJB so that some administrators and teachers are created;

31. If you chose the strategy SINGLE_TABLE at exercise 24, you can drop the STUDENTS table from your database, since all users will be populated in a USERS table;

32. Make sure your database engine is running, run and test the Java EE application (make deploy). Confirm that the data is properly populated in the database;

33. Now, write all the necessary code (Java EE Entities, EJBs, Service Layer, Vue.js/NUXT) for the following application features:

- Enroll/unroll students in/from subjects;
- Show all students enrolled in a subject;
- Associate/dissociate a teacher to/from a subject;
- Show all subjects a teacher is associated to;
- Show all teachers associated to a subject;
- CRUD (Create, Read/Find, Update and Delete) operations for all entities.

Bibliography

Java EE Tutorial 8 (all of it).