Desenvolvimento de Aplicações Empresariais – 2020-21-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 5

Topics: Strategies for loading data; controlling concurrent access to entity data with locking; Exceptions handling; DTOs.

Strategies for loading data

In Java EE, there are two strategies for loading data from the database: *eager loading* and *lazy loading*. Eager loading means that the system provider runtime should eagerly fetch data on the first-time data is accessed, while lazy loading means that data should be fetched lazily on the first access. When lazy loading is used, the loading of attributes and associations of an entity is deferred until they are needed. On the other side, when eager loading is used, all the attributes and associations of an entity are fetched once the entity is first accessed. By default, JPA uses lazy loading on `OneToMany` and `ManyToMany` mappings and eager loading on `ManyToOne` and `OneToOne` mappings. In order to set the loading strategy, you should use the `fetch` attribute of these mapping annotations (with values `FetchType.EAGER` or `FetchType.LAZY`). For other attributes (for example, with `String` attributes), you can use the same attribute (`fetch`) with the `@Basic` annotation.

Controlling concurrent access to entity data with locking

Entity data is concurrently accessed if the data in a datasource is accessed at the same time by multiple applications/execution contexts. When data is updated in, for instance, database tables in a transaction, the persistence provider assumes the database management system will hold locks to maintain data integrity. Most persistence providers will delay database writes until the end of the transaction, except when the application explicitly calls for a flush. By default, persistence providers use optimistic locking, where, before committing changes to the data, the persistence provider checks that no other transaction has modified or deleted the data since the data was read.

This is accomplished by a version column in the database table, with a corresponding version attribute in the entity class. When a row is modified, the version value is incremented. The original transaction checks the version attribute, and if the data has been modified by another transaction, a `javax.persistence.OptimisticLockException` will be thrown, and the original transaction will be rolled back. When the application specifies optimistic lock modes, the persistence provider verifies that a particular entity has not changed since it was read from

the database even if the entity data was not modified. Pessimistic locking goes further than optimistic locking. With pessimistic locking, the persistence provider creates a transaction that obtains a long-term lock on the data until the transaction is completed, which prevents other transactions from modifying or deleting the data until the lock has ended. Pessimistic locking is a better strategy than optimistic locking when the underlying data is frequently accessed and modified by many transactions. However, we will focus on optimistic locking because that is not only the most prevalent, but also the most useful way to scale an application.

We can ask how the provider can know whether somebody made changes in the intervening time since the committing transaction read the entity. The answer is that the provider maintains a versioning system for the entity. For it to do this, the entity must have a dedicated persistent field or property declared in it to store the version number of the entity that was obtained in the transaction. The version number must also be stored in the database. When going back to the database to update the entity, the provider can check the version of the entity in the database to see if it matches the version that it obtained previously. If the version in the database is the same, the change can be applied, and everything goes on without any problems. If the version was greater, somebody else changed the entity since it was obtained in the transaction, and an exception should be thrown. The version field will get updated both in the entity and in the database whenever an update to the entity is sent to the database. Version fields are not required, but their use is recommended for portability reasons.

1. Add the following field to the `User`, `Course` and `Subject` entities:

   ```
   @Version private int version;
   ```

   Version fields should not be modified by the application after the entity has been created. You do not need to receive it at the constructor nor have getter and setter methods for it.

2. By default, JPA assumes what is defined in the ANSI/ISO SQL specification and known in transaction isolation parlance as Read Committed isolation. This standard isolation level simply guarantees that any changes made inside a transaction will not be visible to other transactions until the changing transaction has been committed.

   The application may increase the level of locking for an entity by specifying the use of lock modes. Lock modes may be specified to increase the level of optimistic locking or to request the use of pessimistic locks. There are 8 lock modes. Here, we will use the `OPTIMISTIC` lock mode, which guarantees that both the transaction that obtains the entity read lock and any other that tries to change that entity instance will not both succeed. At least one will fail, but like the database isolation levels, which one fails depends upon the implementation.

   The lock modes can be used with different methods. Here, we will use the `EntitiyManager.lock()` method.

   Go to method `update()` of the `StudentBean` EJB and insert the following line right before the lines that modify the student attributes:

   ```
   …
       em.lock(student, LockModeType.OPTIMISTIC);
   …
   ```

Your code should be something like this:

```
public void updateStudent(String username, String password, String name, String
email, int courseCode) {
            Student student =  em.find(Student.class, username);
            if(student!=null){
                Course course = em.find(Course.class, courseCode);
                if(course!=null) {
                    em.lock(student, LockModeType.OPTIMISTIC);
                    student.setCourse(course);
                    student.setName(name);
                    student.setEmail(email);
                    student.setPassword(password);
                }else
                    System.err.println("ERROR_FINDING_COURSE");
            }else
                System.err.println("ERROR_FINDING_STUDENT");
    }
```

Run the application.

For more information on controlling concurrent access and their usage, consult Section 45 ("Controlling Concurrent Access to Entity Data with Locking") of the Java EE Tutorial;

Exceptions handling

In any application, we must identify what may go wrong and define how the application should behave if those situations occur. We will now address how to deal with three such situations regarding the *create student* feature of the Academic Management application. After these steps, you should apply this knowledge to all other situations that may go wrong in the application. Let us start by dealing with the situation where we want to persist an already existing entity and the one where we want to use a nonexistent entity.

3. Create the `MyEntityExistsException`, `MyEntityNotFoundException` and `MyIllegalArgumentException` exceptions in the `exceptions` package;
4. Change the `create()` method in the `StudentBean` EJB so that:
   - A `MyEntityExistsException` is thrown if a student with the received username already exists in the database;
   - A `MyEntityNotFoundException` is thrown if there is no course in the database with the course code received as parameter;
5. Define appropriate exception messages for each situation in all methods of all EJBs;
6. Add the following attribute to the `ConfigBean` EJB:

```
// Pay attention to the correct import: import java.util.logging.Logger;
private static final Logger logger = Logger.getLogger("ejbs.ConfigBean");
```

7. Add/Modify the catch block of the `populateDB()` method of the `ConfigBean` EJB so that the `Exception` exception is caught and a warning message is shown using the logger attribute. In this method, create a student with an already existent username, run the application and confirm that the exception message is shown in the server output console. Create a new student with a nonexistent course code and run the application again…

```
@PostConstruct
private void populateDB() {
    try {
    //Your previously written code goes here

    // The code below shall throw an instance of «MyEntityNotFoundException»,
because, in this example, the course with code "1000" does not exist.
    // You should be able to find the error message in the server logs console.
    studentBean.create("foo", "bar", "foo", "foo@mail.pt", 1000);
    } catch (Exception e) {
          logger.log(Level.SEVERE, e.getMessage());
    }
}
```

8. We will now do the necessary code to handle exceptions in the services. Create the `MyEntityExistsExceptionMapper` class (`exceptions` package) with the following code:

```
@Provider
public class MyEntityExistsExceptionMapper
        implements ExceptionMapper<MyEntityExistsException> {

    private static final Logger logger =
Logger.getLogger("exceptions.MyEntityExistsExceptionMapper");

    @Override
    public Response toResponse(MyEntityExistsException e) {
        String errorMsg = e.getMessage();
        logger.warning("ERROR: " + errorMsg);
        return Response.status(Response.Status.CONFLICT).entity(errorMsg).build();
    }
}
```

9. Create a similar `MyEntityNotFoundExceptionMapper`:

```
@Provider
public class MyEntityNotFoundExceptionMapper implements
ExceptionMapper<MyEntityNotFoundException> {
    private static final Logger logger =
Logger.getLogger("exceptions.MyEntityNotFoundExceptionMapper");

    @Override
    public Response toResponse(MyEntityNotFoundException e) {
        String errorMsg = e.getMessage();
        logger.warning("ERROR: " + errorMsg);
        return
Response.status(Response.Status.NOT_FOUND).entity(errorMsg).build();
    }
}
```

10. Finally, create a `MyIllegalArgumentExceptionMapper` and `CatchAllExceptionMapper` classes. The first one should return a `BAD_REQUEST` response code. The second one should return a *`INTERNAL_SERVER_ERROR`* response code.

11. Replace the `createNewStudent()` method in the `StudentService` class by the following one:

```
public Response createNewStudent (StudentDTO studentDTO)
      throws MyEntityExistsException, MyEntityNotFoundException{
      studentBean.create(
                        studentDTO.getUsername(),
                        studentDTO.getPassword(),
                        studentDTO.getName(),
                        studentDTO.getEmail(),
                        studentDTO.getCourseCode());
      return Response.status(Response.Status.CREATED).build();
}
```

12. You may be wondering why don't we write a try-catch block in this method to process the `MyEntityExistsException` and the `MyEntityNotFoundException` exceptions. The reason is that the `@Provider` annotation in the above `…Mapper` classes tells the JAX-RS runtime to discover these classes when one of the mapped exceptions is thrown (and not explicitly catched) and run the `toResponse()` method;

13. Modify all the service methods of the `StudentService` and `CourseService` classes according to this exceptions handling strategy;

14. Run the application (make monitor) and test the services through the HTTP request file (alter the requests so that exceptions are thrown);

Let's now deal with the `ConstraintViolationException` exception, which should be caught when we have entity attributes validation annotations that can be violated when we persist or update entities:

15. Create the `MyConstraintViolationException` exception:

```
package exceptions;

import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationException;

public class MyConstraintViolationException extends Exception {
    public MyConstraintViolationException(ConstraintViolationException e) {
        super(getConstraintViolationMessages(e));
    }

    private static String
getConstraintViolationMessages(ConstraintViolationException e) {
        Set<ConstraintViolation<?>> cvs = e.getConstraintViolations();
        StringBuilder errorMessages = new StringBuilder();
        for (ConstraintViolation<?> cv : cvs) {
            errorMessages.append(cv.getMessage());
            errorMessages.append("; ");
        }
        return errorMessages.toString();
    }
}
```

16. Create the corresponding mapper, which should return a `BAD_REQUEST` response code.

17. In the `create()` method in the `StudentBean` EJB, add a try-catch block around the code necessary to persist the student entity, catch the `ConstraintViolationException` exception and throw a `MyConstraintViolationException`. The result must be something like the following code:

```
public void create(String username, String password, String name, String email, int
courseCode)
        throws MyEntityNotFoundException, MyEntityExistsException,
MyConstraintViolationException{
    Student s = em.find(Student.class, username);
    if(s != null){
        throw new MyEntityExistsException("Student with username: " + username + "
already exists");
    }
    Course course = em.find(Course.class, courseCode);
    if(course == null){
```

```
            throw new MyEntityNotFoundException("Course with code: " + courseCode + "
not found.");
        }
        try {
            s = new Student(username, password, name, email, course);
            course.addStudent(s);
            em.persist(s);
        } catch (ConstraintViolationException e) {
            throw new MyConstraintViolationException(e);
        }
    }
```

18. Run and test the application (make monitor). You can artificially force, for example, an invalid email format in a POST or PUT call in the HTTP request file.

19. You may think that there is no point in having entity attributes validation annotations and in catching and dealing with the `ConstraintViolationException` exception since we may validate these in the client application. However, the development team that develops the entities and EJBs may want to prevent possible errors from teams that develop client applications.

20. Change the NUXT project to show the error message when the user failed to create a new student. For that, change the code in the pages/students/create.vue. The resulting code should be like this:

```
<template>
  <div>
    <h1>Create a new Student</h1>
    <form @submit.prevent="create">
      <div>
        username: <input v-model="formData.username" type="text">
      </div>
      <div>
        password: <input v-model="formData.password" type="password">
      </div>
      <div>
        name: <input v-model="formData.name" type="text">
      </div>
      <div>
        email: <input v-model="formData.email" type="email">
      </div>
      <div>
        course code:
        <select v-model="formData.courseCode">
          <option disabled selected>*** Choose a Course ***</option>
          <option v-for="course in courses" :id="course.code" :value="course.code">
            {{ course.name }}
          </option>
        </select>
      </div>
      <p v-show="errorMsg" class="text-danger">
        {{ errorMsg }}
      </p>
      <button type="reset" @click="errorMsg = false">
        RESET
      </button>
      <button @click.prevent="create">
        CREATE
      </button>
    </form>
    <nuxt-link to="/students">
      Back
    </nuxt-link>
  </div>
</template>

<script>
```

```
export default {
  data () {
    return {
      formData: {
        username: null,
        password: null,
        name: null,
        email: null,
        courseCode: null
      },
      courses: [],
      errorMsg: false
    }
  },

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

  methods: {
    create () {
      this.$axios.$post('/api/students', this.formData)
        .then(() => {
          this.$router.push('/students')
        })
        .catch((error) => {
          this.errorMsg = error.response.data
        })
    }
  }
}
</script>
```

DTOs

In worksheet 4 you have developed a page that shows the details of some student. This page must do two requests to the enterprise application API: one to retrieve the basic attributes of the student (username, name, etc.) and one to retrieve the student's list of subjects. Now, we are going to modify the code so that only one request is needed:

21. Add a list of `SubjectDTO` to the `StudentDTO` class as an attribute, create the list on both constructors and add the corresponding getters and setters methods.

Please, notice that, while we want to return a `StudentDTO` object with all student's subjects when we open the student details page, we still want to send a list of `StudentDTO` objects without the respective list of subjects when we open the students' index page with the table of students (why?). So, we need to have a way of building `StudentDTO` objects with and without the list of subjects.

22. Change the name of the `toDTO(Student student)` and `toDTOs(List<Student> students)` methods to `toDTONoSubjects(Student student)` and `toDTOsNoSubjects(List<Student> students)`, respectively.

23. Create the `toDTO(Student student)` and `toDTOs(List<Student> students)` methods, which return a `StudentDTO` object and a list of `StudentDTO` with the students' list of subjects.

24. Modify the `all()` method in the `StudentService` class so that it calls the method `toDTONoSubjects(Student student)` instead of `toDTOs(Student student)`. Notice that method `getStudentDetails(…)` still calls the `toDTO(…)` method (which also return the student with subjects), so we do not need to change it.

25. Run the application and test the changed services using the HTTP Request file.

26. In the NUXT project, edit the _username.vue page to now only fetch the student and render the subjects present in the student object. The resulting code should be like this:

```
<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</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: {},
      subjectFields: [ 'code', 'name', 'courseCode', 'courseYear',
'scholarYear' ]
    }
  },
  computed: {
    username() {
      return this.$route.params.username
    },
    subjects() {
      return this.student.subjects || []
    }
  },
  created() {
    this.$axios.$get(`/api/students/${this.username}`)
      .then((student) => {
        this.student = student || {}
      })
  },
}
</script>
```

27. Run the academics-management NUXT project and test it.

Bibliography

Java EE Tutorial 8 (all of it).