IPL escola superior de tecnologia e gestão instituto politécnico de leiria

DEI Departamento Engenharia Informática

Morro do Lena, Alto Vieiro · Apart. 4163 · 2401 – 951 Leiria
Tel.: +351 · 244 820 300 · Fax.: +351 · 244 820 310
E-mail: estg@estg.ipleiria.pt · http://www.estg.ipleiria.pt

Desenvolvimento de Aplicações Empresariais – 2018-19-1S

Engenharia Informática – 3.º ano – Ramo SI

Worksheet 5

Topics: Exceptions handling, validators, data transfer objects

Exceptions Handling

In any application, we must identify what may go wrong and define how the application should behave if those situations occur. You will now learn 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.

1. Create the `EntityExistsException` and `EntityDoesNotExistException` exceptions.
2. Change the `createStudent(…)` method in the `StudentBean` EJB so that:
3. An `EntityExistsException` is thrown if a student with the received *username* already exists in the database.
4. An `EntityDoesNotExistException` is thrown if there is no course in the database with the course code received as parameter.
   1. The exceptions thrown in step 2 are catched and rethrown.
   Define appropriate exception messages for each situation.
5. Modify the `populateDB()` method of the `ConfigBean` EJB so that the `EntityExistsException` and `EntityDoesNotExistException` exceptions are caught and their message shown using, for example, a `System.err.println(…)`; In this method, create a student with an already existent username; Run the application and confirm that the exception message is shown in the glassfish output window; Create a new student with a nonexistent course code; Run the application and confirm that…

What we have done so far is only useful for development time. What if a user creates a new student with an already existing username? S/he still sees nothing happening…

6. Now, catch the `EntityExistsException` and `EntityDoesNotExistException` exceptions in the *AdministratorManager* MB `createStudent()` method; In the catch block for those two exceptions add the following line:
   ```
   logger.warning(e.getMessage());
   ```
   In the catch block for other exceptions add the following line:
   ```
   logger.warning("Unexpected error. Try again latter!");
   ```
   Run the application and create a student with an already existing username through the web application interface. The `EntityExistsException` message should appear in the glassfish output window. However, this still is useless to the application user… We must find a way of warning the user about what happened.
   Download the `FacesExceptionHandler` class from Moodle and put it in the *web* package. This class has two `handleException(…)` methods. The first one should be used when we want a message to be shown using the `h:messages` JSF component; This component shows all messages that are not associated with specific page components. The second one should be used when we want a message to be shown in the `h:message` component of a given page component. Let's now see how to use each one of these methods.
7. Replace the `logger.warning(e.getMessage());` line by the following one:
   ```
   FacesExceptionHandler.handleException(e, e.getMessage(), logger);
   ```
   Replace the `logger.warning("Unexpected error. Try again latter!");` by the following one:
   ```
   FacesExceptionHandler.handleException(e, "Unexpected error! Try again latter!", logger);
   ```
   Replace the *return* `"admin_students_create?faces-redirect=true";` in both catch sections by `return null;` so that the `admin_students_create` page context remains the same.
8. Put the following component at the end of the admin_students_create.xhtml facelet (right before the `</h:body>` tag:
   ```
   <h:messages style="color:red"/>
   ```
9. Run the application and create a student with an already existing username. The `EntityExistsException` message should appear below the "Create" button.
10. Now, suppose that, instead of the previous behavior, we want the message to appear in the *h:message* component of the "Create" button (the message will appear to the right of the button instead of the bottom of the page). Add the following attribute to the AdministratorManager MB and the respective getter and setter methods:
    ```
    private UIComponent component;
    ```
    Add the following attribute to the "Create" button in the `admin_students_create.xhtml` facelet:
    ```
    binding="#{administratorManager.component}"
    ```

Replace the `FacesExceptionHandler.handleException(e, e.getMessage(), logger);` line by the following one:

```
FacesExceptionHandler.handleException(e, e.getMessage(), component, logger);
```

Replace the `FacesExceptionHandler.handleException(e, "Unexpected error! Try again latter!", logger);` line by the following one:

```
FacesExceptionHandler.handleException(e, "Unexpected error! Try again latter!", component, logger);
```

Remove the `<h:messages style="color:red"/>` component in the facelet.

11. Run the application and create a student with an already existing username. The `EntityExistsException` message should appear to the right of the "Create" button.

12. 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:

13. Create the `MyConstraintViolationException` exception.
14. Create the `Utils` class in the exceptions package and add it the following method:

```
public 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();
    }
```

15. Catch the `ConstraintViolationException` exception in the `create()` method in the `StudentBean` EJB. Add the following to the catch block:

```
throw new MyConstraintViolationException(Utils.getConstraintViolationMessages(e));
```

16. Catch the `MyConstraintViolationException` exception in the `createStudent()` method in the `AdministratorManager` MB and deal with it as for our two other exceptions. Do the same in the `populateDB()` method in the `ConfigBean` EJB.
17. Run and test the application. Since we have input validations in the facelets, you can artificially force, for example, a null student name and/or an invalid email format in the call to method `createStudent()` of the `StudentBean` EJB.

18. You may think that there is no point on having entity attributes validation annotations and on catching and dealing with the `ConstraintViolationException` exception since we already have validation code in facelets. However, the development team that develops the entities and EJBs may want to prevent possible errors from teams that develop (other) client applications.

Validators

JavaServer Faces (JSF) allows programmers to add more functionality to components, namely through converters, listeners and validators (read Chapter 11 of the Java EE Tutorial). We may use some already provided JSF converters and validators. For example, we have already used the `f:validateRegex` to validate the email format. You will now learn how to define your own validators. We have two ways of doing this: 1) using a method in an MB or 2) using a class that implements the `javax.faces.validator.Validator` interface. The second approach allows a greater portability of the validator. Let's try both approaches.

Approach 1)

19. Add the following method to the *AdministratorManager* MB:

```
public void validateUsername(FacesContext context, UIComponent toValidate, Object value) {
        try {
            //Your validation code goes here
            String username = (String) value;
            //If the validation fails
            if (username.startsWith("xpto")) {
                FacesMessage message = new FacesMessage("Error: invalid username.");
                message.setSeverity(FacesMessage.SEVERITY_ERROR);
                context.addMessage(toValidate.getClientId(context), message);
                ((UIInput) toValidate).setValid(false);
            }
        } catch (Exception e) {
            FacesExceptionHandler.handleException(e, "Unknown error.", logger);
        }
    }
```

20. Add the following attribute to the *username* inputText component in the `admin_students_create.xhtml` facelet:
`validator="#{administratorManager.validateUsername}"`
21. Run and test the application.

Approach 2)

22. Download the `UsernameValidator` class from Moodle and put it in the *web* package.
23. Change the *username* `inputText` component in the `admin_students_create.xh`tml facelet, adding it the following component:
`<f:validator validatorId="usernameValidator"/>`
24. Run and test the application.

Data Transfer Objects

Business methods in the EJBs we have written so far return (lists of) entities' instances (we are referring to methods supposed to return data about entities). However, returning entities instances is not a good practice (the reasons why were already explained in the theoretical classes). Instead of entities instances, we should return data transfer objects (DTOs) that contain the information needed by the method caller. The data in a DTO may correspond directly to the one

of a specific entity, but it can also be a mixture of data from different entities or consist in just part of an entity's data, depending on the method caller needs. We will now apply this approach to just some features of the `StudentBean` EJB. You should then generalize this practice to all features where EJBs' methods returning composite data about entities should be called.

25. Download the dtos.zip file from Moodle. This file has a directory (package) with the UserDTO, StudentDTO, CourseDTO, SubjectDTO and TeacherDTO classes. Copy it to the proper project directory. Has you may notice, these DTOs are "flat" classes. For example, the StudentDTO has no subjects list. We could opt for an approach where DTOs also include other DTOs lists or even use both approaches. Each approach has advantages and disadvantages that will be discussed in the class.
26. Modify the EJBs so that DTO instances are returned instead of entity instances.
27. Modify the `AdministratorManager` MB and all your facelets so that they now work with DTOs and not with entities. You may even replace the attributes `newStudentUsername`, `newStudentPassword`, etc. by a DTO so that you have less code in the MB (this will imply several modifications in the MB and in the `admin_students_create.xhml` facelet). Do not forget to instantiate the DTO in the MB constructor.
28. Run and test the application.

Bibliography

Java EE Tutorial 7 (all of it).