Table of Contents

Practical task: Eric Haneder (Front end)

Beginning

To start with designing user interfaces (= front end), a design must be agreed with the client which suits the client an is realizable for us. The front end is the most important part for the user, because it includes everything the user can see and interact with. I discussed this topic with the client and we came to a conclusion

Here is an abstract picture, how the user interfaces should look like:

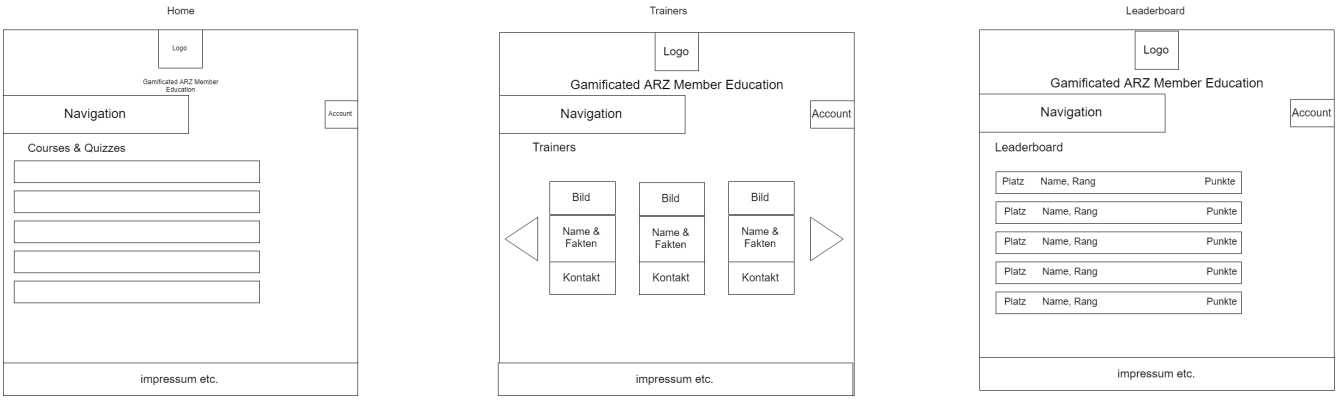


Figure 1. User interfaces

After the design was agreed upon, the programming of the interfaces could begin.

Fundamentals

To program a website, you need a lot of extensive knowledge. You have to know HTML, Javascript and CSS. In addition, you should also know how a finished page is generated from the code you programmed and how requests from the user can be evaluated and responded to.

JSF

Java Server Faces (JSF) is a programming framwork for the development of graphical user interfaces (GUIs). JSF is a part of the web-technologies of *Java Enterprise Edition* (Java EE). The following graphic shows the architecture of JSF:

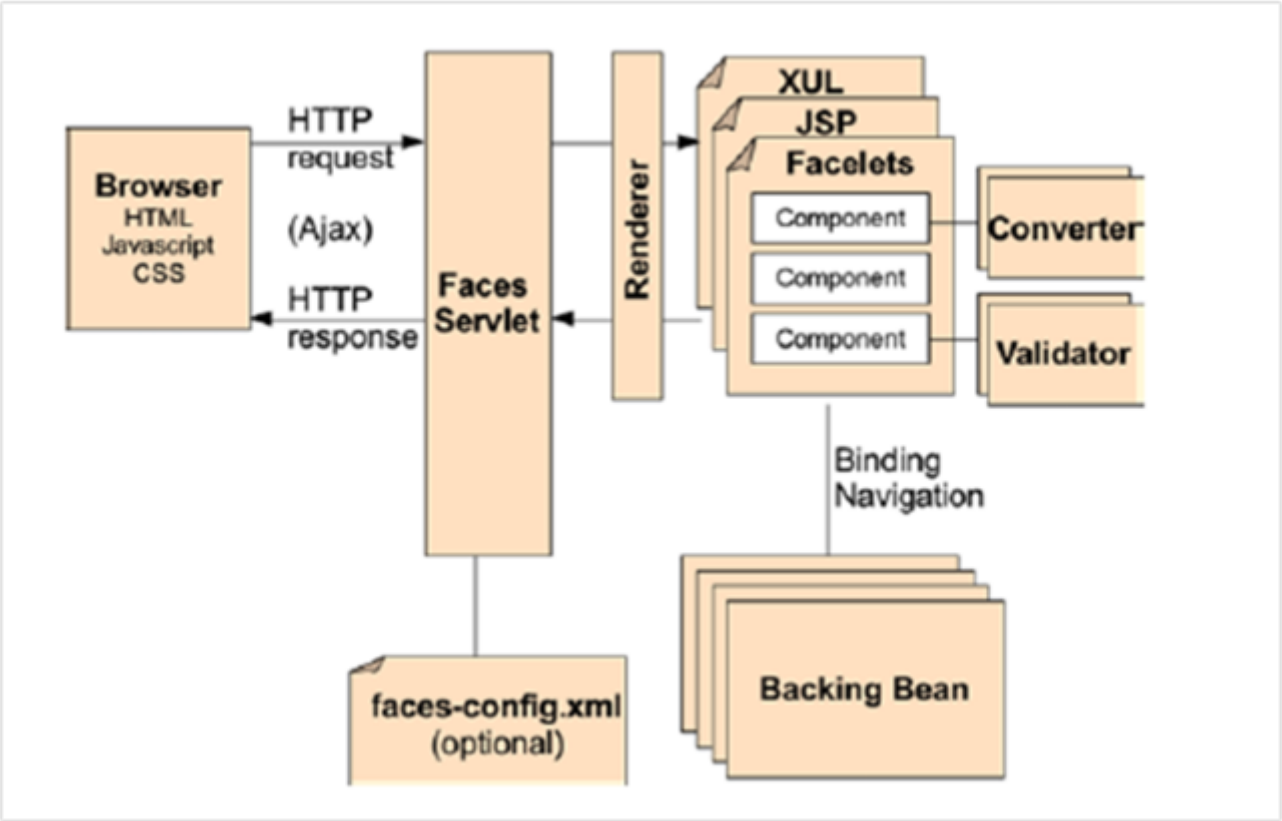


Figure 2. JSF architecture

Browser

The Browser is there, to display the websites to the user. Through the browser, users can navigate through the interfaces, make requests to the server and get a respone back.

Faces Servlet

The Faces Servlet handles user interactions that may lead to changes in the data structure (back en) or on the user interfaces (front end). It can be seen as a controller between the front end and the back end. Every request runs through the Faces Servlet which takes action accordingly. It can optionally be configured by a *faces-config.xml* file.

Renderer

The Renderers are responsible for displaying a component an translating a user's input into the component property values.

Converter

Converters convert a component's value (Integer, Boolean, etc.) to and from markup values (String).

Validators

Validators are responsible for ensuring that the value entered by a user is valid.

Backing Beans

The business logic is made in backing beans, which also control the navigation between pages.

Facelets

Facelets describes the language in which JSF files are written (XHTML). Every page has certain

components which will be displayed by the browser with the help of renderers.

(vgl. Goncalves A. (2013). Beginning Java EE 7. Apress.)

HTML

Hypertext Markup Language (HTML) is the common used language for building web pages. A HTML page is a text document (with a .html or .htm extension) used by browsers to present text and graphics. A web page is made of content, tags to change some aspects of the content, and external objects such as images, videos, JavaScript, or CSS files.

Sample HTML code

```
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <h1>This is my webpage</h1>
        <p>
            I hope you like it.
        </p>
        <a href="www.google.at">Link to Google</a>
    </body>
</html>
```

U can notice several tags in this code (such as <body> or <p>). Every tag has its on purpose, attributes and must be closed. Href is an attribute of the a-tag.

XHTML is just a validated version of html. This means, that there are certain rules, a html-page has to follow, to be valid. XHMTL pages have a .xhtml extension.

Some of the rules are the following:

- All tags must be closed. (so no
, <hr>, ...)
- All tags are lowercase.
- Attributes appear appear between single or double quotes (<table border="0"> instead of <table border = 0>)
- There must be a strict structure with <html>, <head> and <body> tags.

CSS

Cascading Style Sheets (CSS) is a styling language used to describe the presentation of a document written in html or xhtml. CSS is used to define colors, fonts layouts, and other aspects of document presentation.It allows separation of a document's content (written in XHTML) from its presentation (written in CSS). To embedd a .css file in your html or xhtml page, use the <link> tag. (e.g. <link rel="stylesheet" type="text/css" ?

Sample CSS code

```
p {
    font-size: 10px;
}
h1 {
    color: red;
    font-style: italic;
}
```

Getting Started

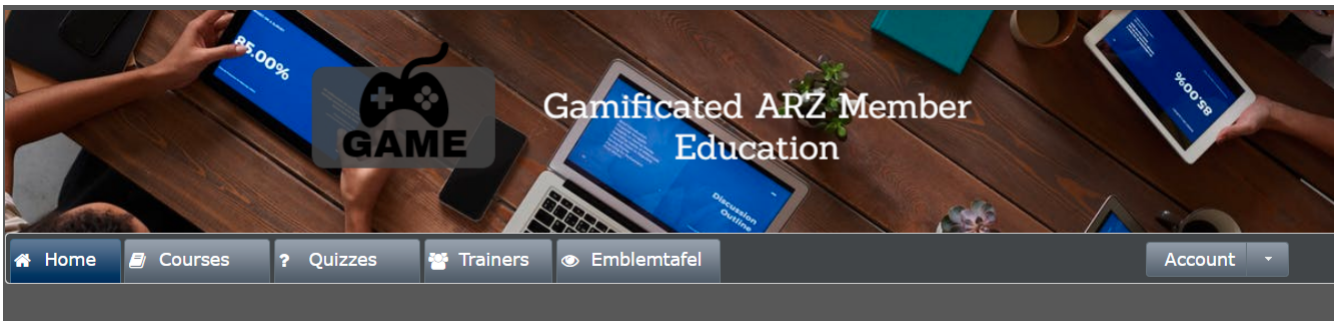
Creating a project structure

To start with a project like this, you need a clean project structure. The project structure has been automatically created with Maven. Our project is "Open Source", that means it is available online for free. That includes our whole project structure and all of the files used to create the website. The complete GAME project can be viewed under: https://github.com/game-admin/game

Creating a layout

To simplify the creation of the UIs, I prepared a layout which serves every page as a template. This layout is realised in "mainlyaout.xhtml". It constructs the picture at the the top of every file with the headline, the menubar and a little picture in the footer.

Design/Layout of the game platform



This page is not displayed directly to the user, but rather represents a template for all user interfaces. Other pages can define this page as a template and then adopt its content. By using <ui:insert> in the template file, you can give the template clients the possibility to define the content of this tag themselves with <ui:define>. This is shown in any of the actual user interfaces.

The menubar is constructed on another file called "menubar.xhtml". It is used to navigate through the platform. Furthermore, the menubar file is very unique, because it is not used directly in every file, but is displayed on every page. This is due to all of the pages using the mainlayout file as a template. It is include in the mainlayout file with the <ui:include> tag:

```
<ui:include src="./faces/menubar.xhtml"></ui:include>
```

The menubar is created by usind the <p:tabMenu> tag of the Primefaces library. It is very convenient to create a menubar with this tag.

menubar.xhtml

```
<p:tabMenu style="width:100%">
    <p:menuitem value="Home" outcome="index.xhtml" style="width:5em" icon="fa fa-home
">
    </p:menuitem>
    <p:menuitem value="Courses" outcome="courses.xhtml" style="width:7em" icon="fa fa-
book">
    </p:menuitem>
    <p:menuitem value="Quizzes" outcome="quizzes.xhtml" style="width:7em" icon="fa fa-
question">
    </p:menuitem>
    <p:menuitem value="Trainers" outcome="trainers.xhtml" style="width:6em" icon="fa
fa-users">
    </p:menuitem>
    <p:menuitem value="Emblemtafel" outcome="leaderboard.xhtml" style="width:8em"
icon="fa fa-eye">
    </p:menuitem>
</p:tabMenu>
```

The "Logout" button on the far left is done with the <p:splitButton> tag. It is placed in the menubar with CSS.

menubar.xhtml

```
<p:splitButton id="basic" value="Account" action="index.html">
    <p:menuitem value="Quizzes" action="quizzes.xhtml"/>
    <p:menuitem value="Courses" action="courses.xhtml"/>
    <p:separator/>
    <p:menuitem value="Logout" url="http://www.google.com"/>
</p:splitButton>
```

I had to separate the menubar from the mainlyout, because of interferences with the formulars.

User-Interfaces

User Interfaces describe everything the user can use, to commincate with the application.

The index-page is the standard page the browser will run, if you enter a website. On this page, the user should get an overwiev about his statistics, and he should be able to navigate to other pages. The structure of the index-page is pretty simple. All data of the current user is fetched from the database and displayed. The trainee can look at his nickname, score, progress and emblems. Furthermore, a little description of the GAME-site is displayed. From here on, the trainee can check out some courses, take quizzes, look at trainers or visit the emblemboard.

The data is displayed via expression langugage. Here is an example:

```
#{traineeController.getTraineesByID("1").get(0).nickname}
```

Courses-page

The Courses-page should display a list of courses the trainee can go through. These courses can be mandatory to complete Quizzes.

courses.xhtml

```
<p:dataTable var="kurs" value="#{kursController.kurse}">
            <p:column headerText="Titel">
                <h:outputText value="#{kurs.titel}"></h:outputText>
            </p:column>
            <p:column headerText="Beschreibung">
                <h:outputText value="#{kurs.beschreibung}"></h:outputText>
            </p:column>
            <p:column>
                <h:form>
                    <!--<p:commandButton
action="{kursController.takeKurs(kurs.kursID)}" value="Take Course!"  >
                    </p:commandButton>     -->
                    <!--<h:commandLink
action="{kursController.kursbean.find(kurs.kursID).getLink()}" value="Take
Course!"></h:commandLink>-->
                    <h:commandLink action=
"https://www.tutorialspoint.com/java/index.htm" value="Take Course!"></h:commandLink>
                </h:form>
            </p:column>
        </p:dataTable>
```

The Primefaces tag <p:dataTable> takes a list and knows how to display its content throught the columns. It is the equivalent to the <table> tag of html, but with some extra functions and style modifications. Here, I put in a list of courses. Every course in the list has a titel, description and a link which is displayed in separated columns.

The <p:commanButton> invokes the takeKurs-method when pressed. In this method, a link of the selected course is returned. The <h:commandLink> is linked to the URL of the course. By clicking on it, the user is redirected to the site of the course.

This is how the courses interface looks like:

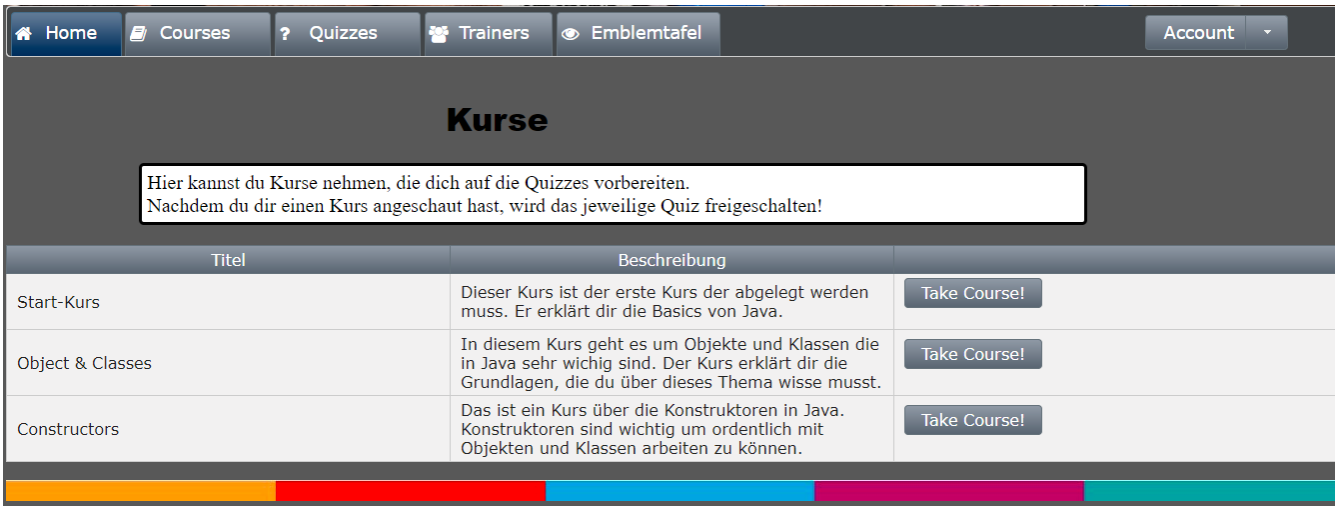


Figure 3. Courses page

Quiz pages

The quiz pages include a interface, where every takeable quiz is displayed, two pages for taking a quiz and a page where the results are shown. Every quiz has its own emblem, which can be won if they quiz is taken succesfully. This means the user has to has at least half of the questions right.

quizzes.xhtml

```
<p:dataTable var="quiz" value="#{quizController.quizzes}">
    <p:column headerText="Titel" rendered="#{quizController.isTakeable(quiz.QID,
&amp;quot;1&amp;quot;)}">
        <h:outputText value="#{quiz.titel}"></h:outputText>
    </p:column>
    <p:column headerText="Beschreibung" rendered=
"#{quizController.isTakeable(quiz.QID, &amp;quot;1&amp;quot;)}">
        <h:outputText value="#{quiz.beschreibung}"></h:outputText>
    </p:column>
    <p:column rendered="#{quizController.isTakeable(quiz.QID, &amp;quot;1&amp;quot;)}">
        <h:form>
            <h:commandButton action="#{quizController.quizUebergabe(quiz.QID)}" value
="Take Quiz!" >
            </h:commandButton>
        </h:form>
    </p:column>
</p:dataTable>
```

Here, <p:dataTable> is used again, this time to show all available quizzes. It gets a list of quizzes and displays a quiz whether or not it is takeable. This is evaluated in the isTakeable-method in the QuizController Bean.

This is how it looks like, when the user has not fulfilled the requirements to take a quiz:

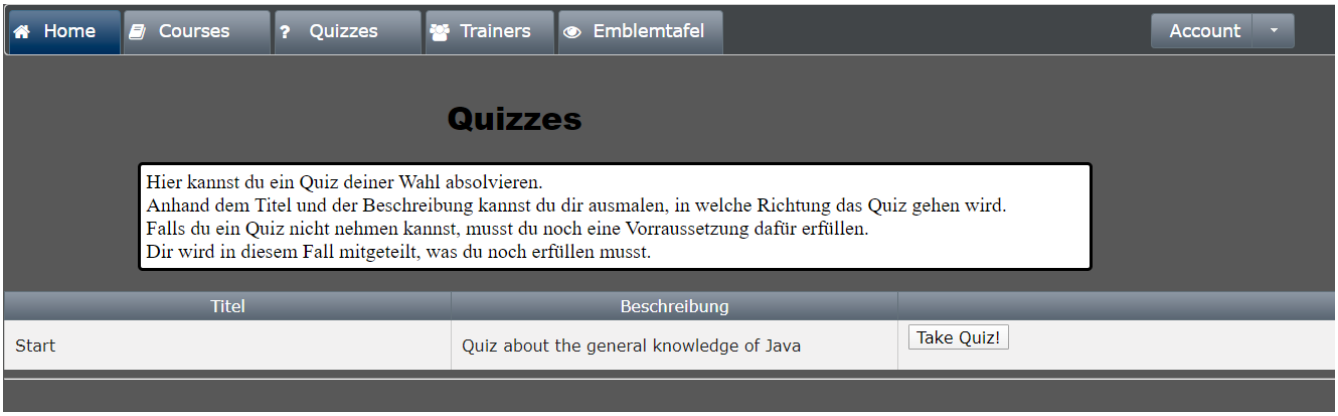


Figure 4. Quizzes page

In this picture below, the trainee has met all the requirements needed to take the other two quizzes. The second quiz requires the first first quiz to be completed succesfully and the third quiz requires the second quiz.

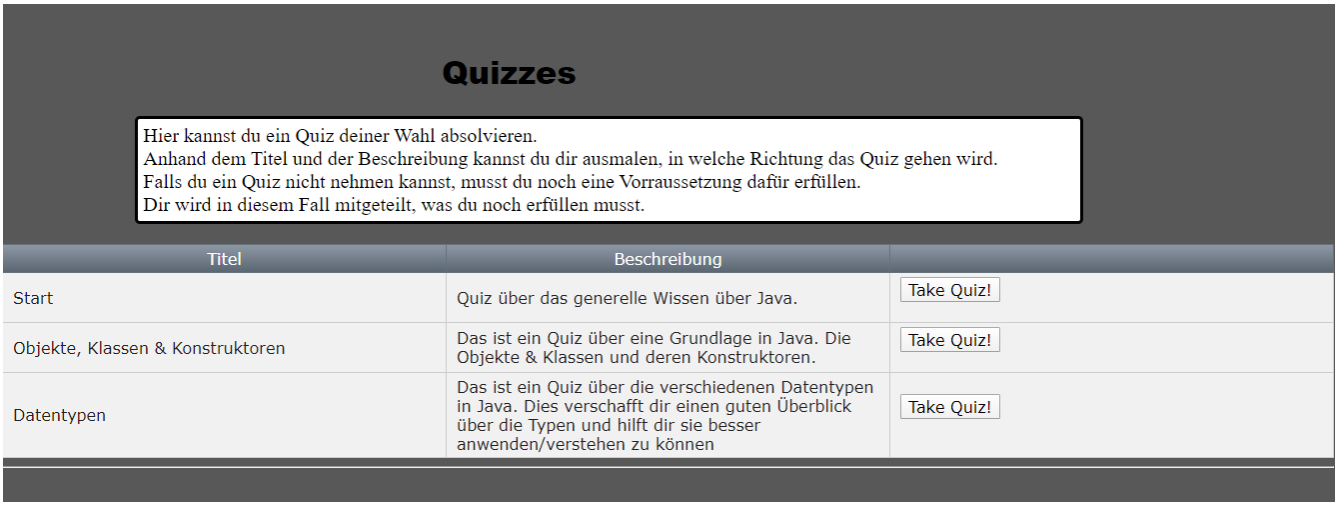


Figure 5. Quizzes page with fulfilled requirements

By clicking on the <p:commandButton>, the user can take the quiz.

takequiz.xhtml

```
<ui:repeat var="frage" value="#{quizController.fragemodell}">
    <div class="question">
    <h:outputLabel for="radio" value="#{frage.frage}"></h:outputLabel>
    <div class="answer">
        <p:selectOneRadio id="radio" value="#{frage.selectedAnswer}" layout="grid"
required="true" unselectable="true" columns="1">
            <f:selectItem itemValue="#{frage.antworten.get(0)}" itemLabel=
"#{frage.antworten.get(0)}"></f:selectItem>
            <f:selectItem itemValue="#{frage.antworten.get(1)}" itemLabel=
"#{frage.antworten.get(1)}"></f:selectItem>
            <f:selectItem itemValue="#{frage.antworten.get(2)}" itemLabel=
"#{frage.antworten.get(2)}"></f:selectItem>
            <f:selectItem itemValue="#{frage.antworten.get(3)}" itemLabel=
"#{frage.antworten.get(3)}"></f:selectItem>
        </p:selectOneRadio>
        <br/>
    </div>
    </div>
    <br/>
</ui:repeat>
```

This is the page for taking singlechoice-quizzes. The questions are repeatadly displayed by the <ui:repeat> tag. This tag runs through a given list, and displays the wanted data. The answers are displayed with the <p:selectOneRadio>, which renderes a set of buttons based on the data you set with <f:selectItem>.

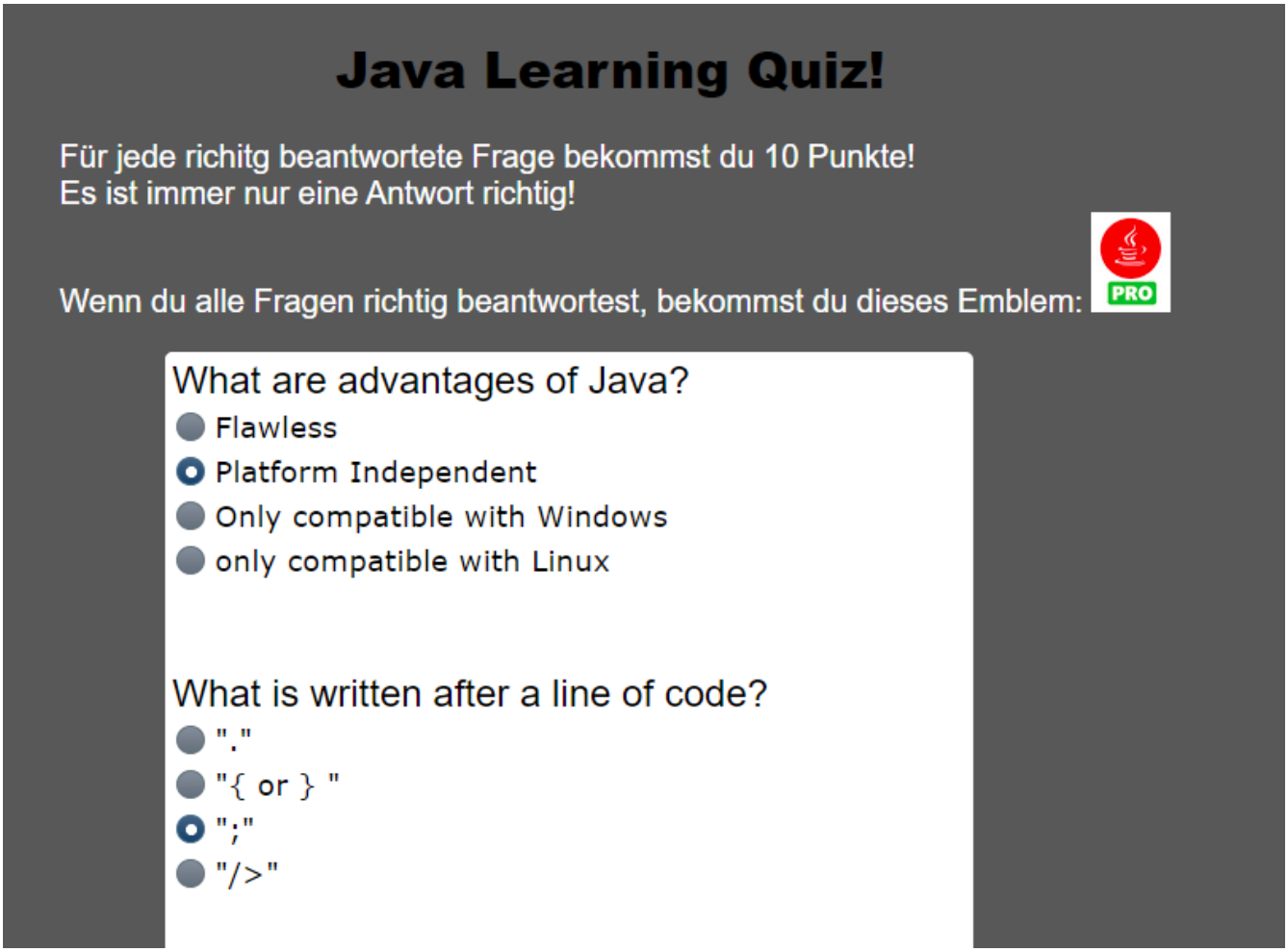


Figure 6. Single-choice quiz

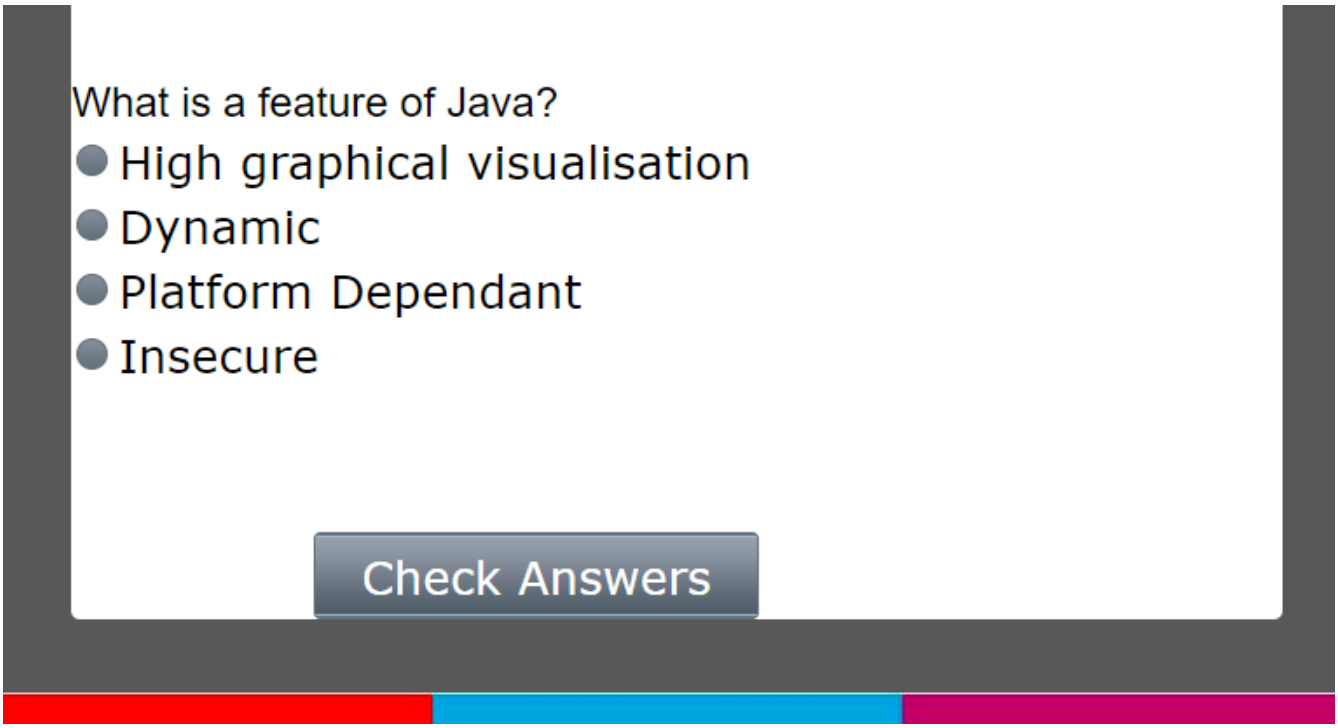


Figure 7. Bottom half of the single-choice quiz

By clicking on the "Check Answers" button, the trainee is redirected to the results page.

The page for taking multiplechoice quizzes looks almost the same, except for the buttons. I used <p:selectBooleanCheckbox> tags here, because they can be used for multiplechoice purposes. Each

of these buttons must be bound to a Boolean property.

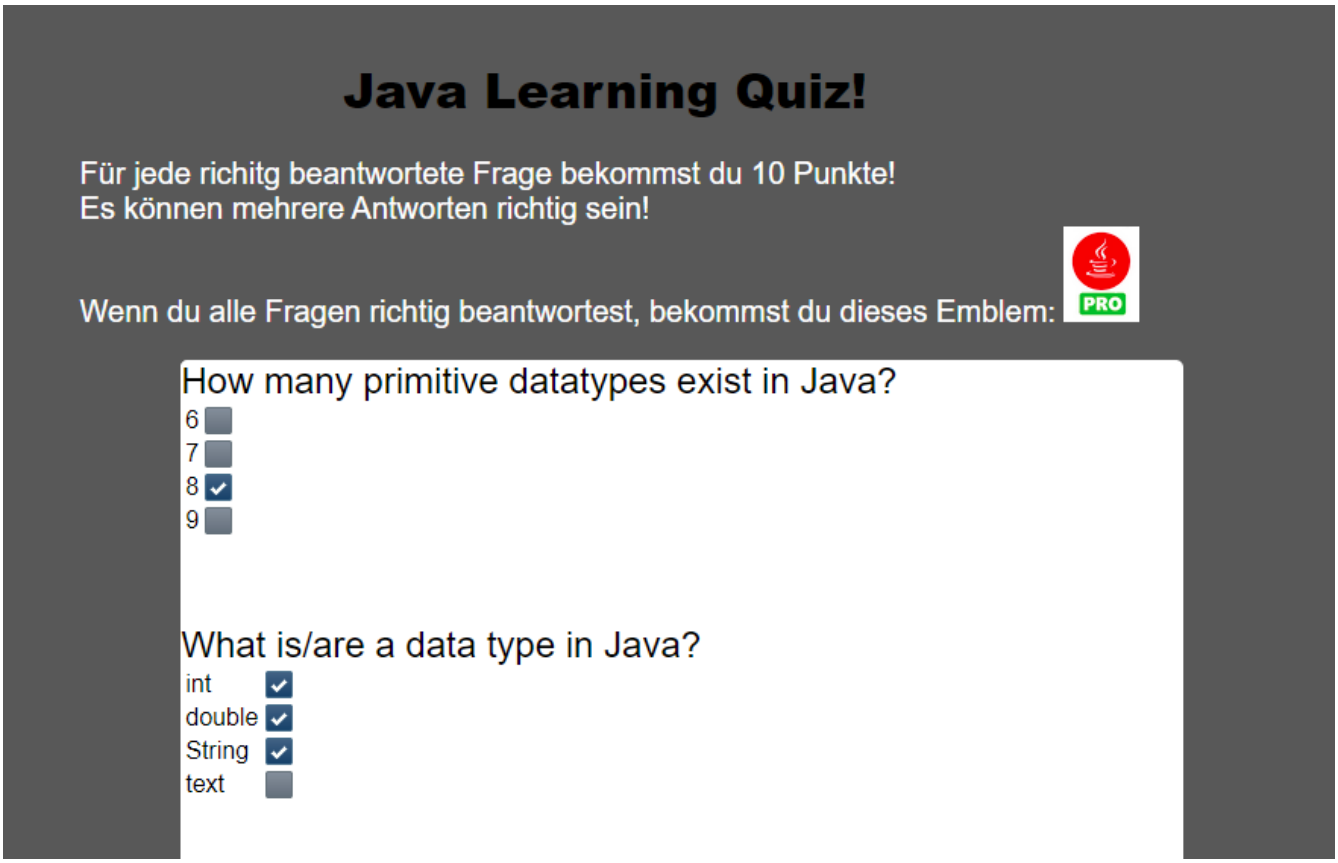


Figure 8. Multiple-choice quiz

The results page is responsible for displaying the results of the quiz taken by the trainee. The trainee is able to see how many questions he/she answered right, how many points he/she won and which questions he/she answered incorrectly. Furthermore, the user sees which answers is right for every question. If the question is green and checked, the user answered correctly. If the question is red with a X at the end, the user answered incorrectly.

Figure 9. Results page

Trainer page

The Trainers page should display all the trainers associated with the GAME platform. The trainees can contact these trainers if they need help.

trainers.xhtml

```
<p:carousel value="#{trainerController.trainers}" headerText="Trainers" var="trainer"
itemStyle="text-align:center" responsive="true">
    <p:panelGrid columns="2" style="width:100%;margin:10px 0px" columnClasses=
"label,value" layout="grid" styleClass="ui-panelgrid-blank">
        <f:facet name="header">
            <p:graphicImage library="img" name="trainer.jpg"/>
        </f:facet>
        <h:outputText value="Name:"  />
        <h:outputText value="#{trainer.name}" />
        <h:outputText value="Rolle:" />
        <h:outputText value="#{trainer.role}" />
        <h:outputText value="Abteilung:" />
        <h:outputText value="#{trainer.branch}" />
    </p:panelGrid>
</p:carousel>
```

Here I used a primefaces tag called <p:carousel>. This tag is used to create a carousel. The <p:panelGrid> tag is used to display data in a grid. The <p:graphicImage> is just like the JSF tag <h:graphicImage>. <h:outputText> is used to display text, with the function to call a Backing Bean.

For easier explanation here is a picture:

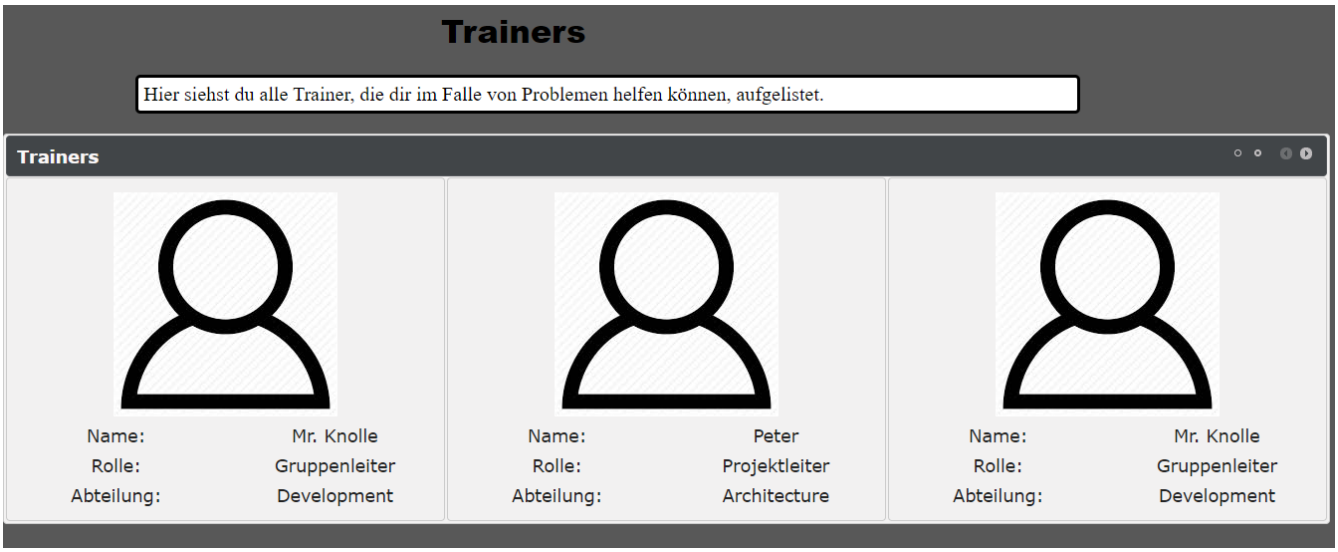


Figure 10. Trainers page

Emblemboard

The emblemboard page is used to diplay all the trainees with their names, nicknames, branches and Icons they got. You should be able to sort them by their names.

leaderboard.xhtml

```
<p:dataTable var="trainee" value="#{traineeController.trainees}">
            <p:column headerText="Name">
                <h:outputText value="#{trainee.vorname}" />
                <h:outputText value="#{trainee.nachname}" />
            </p:column>

            <p:column headerText="Nickname">
                <h:outputText value="#{trainee.nickname}" />
            </p:column>

            <p:column headerText="Abteilung">
                <h:outputText value="#{trainee.abteilung}" />
            </p:column>

            <p:column headerText="Embleme">
                <h:graphicImage value="#{trainee.embleme}"
></h:graphicImage>
            </p:column>
</p:dataTable>
```

Here, the <p:dataTable> tag is used once again. This time it display all the trainees with their emblems for the other trainees to see. Users can compare each other and are able to check out how the other users are doing.

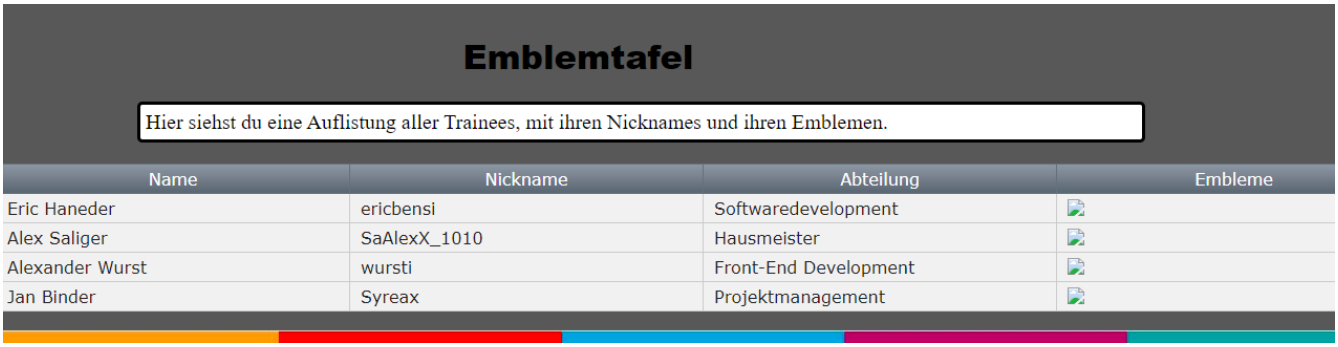


Figure 11. Emblemtafel page

Java classes

Java Classes contain business logic that are need for the application. For example, if you want to display data on a page, you have to fetch data. This is done with Java files. To be more specific, Java classes which communicate with pages are called *Backing Beans*.

Backing Beans are identified by their @Named annotations. Furthermore, every bean has to have a scope annotated. An example for this is the TrainerController class I programmed.

TrainerController.java

```
@Named
@ViewScoped
public class TrainerController implements Serializable {

    private List<Trainer> trainers;

    private Trainer selectedTrainer;

    @Inject ①
    private TrainerService service;

    @PostConstruct
    public void init() {
        trainers = service.createTrainers(6);
    }

    //Getters & Setters
}
```

① Here, the TrainerService class is injected via the Jave EE Dependy Injection System. This way, we can use everything from the injected Class, without the need of calling a contructor. The TrainerServe generates a list of trainers to be displayed by the trainers page.

The TrainerController class is used to diplay all the trainers on the trainers.xhtml page. The list of trainees is used in the <p:carousel> tag.

Another class I developed is the TraineeController. This class is used to display all the trainees on the emblemboard. It looks quite similar to the TrainerController. The only difference is that the

data is created by the TraineeEJB class.

The last class I programmed, is the biggest one. It is called QuizController.java and it handles all of the interactions regarding the quizzes. This means it is responsible for forwarding the trainee from the quizzes page to the takquiz page, and from the takequiz page to the results page. Furthermore, it evaluates if a trainee meets all the requirements to take a quiz and it evaluates the results of a taken quiz.

Quiz Classes

QuizController

```
 public void evaluateScoreMultiple() {
        List<Integer> falsche = new ArrayList<>();
        int richtige=0;
        for(int i=0; i<fragemodell.size(); i++ ) {
            List<Integer> indexrichtig = fragemodell.get(i).indexrichtig;
            for(int z=0; z<4; z++) {
                if(indexrichtig.get(z) == 1 && !fragemodell.get(i).buttons[z] ||
indexrichtig.get(z) == 0 && fragemodell.get(i).buttons[z]) {
                    falsche.add(i);
                    z=999;
                } else {
                    richtige++;
                }
            }
            if(richtige==4) {
                score+=10;
                ricounter++;
                falsche.add(9999);
            }
            richtige = 0;
        }
        checkResults(falsche);
    }
```

This method is used to evaluate the results of a MultipleChoice-Quiz. Each button is bound to a boolean-wert of the buttons[]. Every Question is checked, if every button matches the right answers. The user only gets points, if he answers the question correctly.

QuizController

```
public void evaluateScoreRadio() {
        List<Integer> falsche = new ArrayList<>();
        //fragemodell = creator.createModell(qid);
        int indexri=0;
        for(int i=0; i<fragemodell.size(); i++) {
            for(int j=0; j<fragemodell.get(i).indexrichtig.size(); j++) {
               if(fragemodell.get(i).indexrichtig.get(j) == 1)
                   indexri = j;
            }
            if(fragemodell.get(i).selectedAnswer.equals(fragemodell.get(i).antworten
.get(indexri))) {
                score+=10;
                ricounter++;
                falsche.add(9999);
            } else {
                falsche.add(i);
            }
        }
        checkResults(falsche);
    }
```

Here, the singlechoice-quizzes get evaluated. This is much easier, because the radiobuttons function differently than the normal buttons. Every set of radiobuttons is bound to one value (selectedAnswer). We only need to check if the selected Answer matches the correct Answer.

QuizController

```
public void checkResults(List<Integer> falsche) {
        results = new ArrayList<>();
        for(int i=0; i<fragemodell.size(); i++) {
            List<Integer> indexrichtig = fragemodell.get(i).indexrichtig;
            if(falsche.get(i) == i) {
                results.add(new Results(fragemodell.get(i).frage, fragemodell.get(i)
.antworten, indexrichtig, true));
            } else {
                results.add(new Results(fragemodell.get(i).frage, fragemodell.get(i)
.antworten, indexrichtig, false));
            }
        }
     trainee = traineebean.find("1");
     trainee.setProgress(trainee.getProgress()+score);
     traineebean.update(trainee);
     List<Quizbeantwortung> list =  quizbeantw.findByQIDAndMITID(qid, "1");
     list.get(0).setErreichtePunkte(score);
     if(score > fragemodell.size()*10/2) {
        list.get(0).setIstbestanden(true);
     }
     quizbeantw.update(list.get(0));
    }
```

In *checkResults*, a *Results-List* i s generated. This list is used to diplay the results on the results.xhtml page.

The QuizController is used to handle everything surrounding the action of taking a quiz. It is responsible for displaying the content on the quizzes-, takequiz- and results-page. It will forward the user from the quizzes page to the takequizpage, where he/she can take the quiz. By clicking on th Submit button, the user is forwarded to the results-site, where their results are shown.