Software-Projekt WiSe 23/24

Aufgabenblatt 4

Abgabe bis: 17.12.2023 23:59 Uhr MEZ

Autoren: Karsten Hölscher, Marcel Steinbeck

Auf diesem Aufgabenblatt werden wir einige Vereinfachungen am bisherigen Quellcode vornehmen, Passwörter nicht mehr im Klartext speichern und uns der UI intensiver widmen. Das beinhaltet neben einem einheitlichen Layout auch die Validierung von Eingabedaten.

Aufgabe 1 Redundanzen entfernen

Betrachten wir unsere beiden Service-Klassen `UserService` und `StudentService`, so stellen wir fest, dass sie sich sehr stark ähneln. Beide injizieren ein Repository, das ihre jeweiligen Entitäten `User` bzw. `Student` verwaltet und stellen Methoden zum Persistieren einer Entität (*save()*) und zur Rückgabe aller Entitäten (*findAll()*) zur Verfügung. Die Wahrscheinlichkeit ist sehr groß, dass zukünftige Service-Klassen ebenfalls diese Methoden (und noch einige weitere generelle Methoden für alle Entitäten) bereitstellen werden. Damit diese Methoden nicht immer kopiert werden müssen und dann redundant vorhanden sind, führen wir eine abstrakte Oberklasse `AbstractBaseService` im Unterpaket `service` ein.

```
1  @Transactional
2  public abstract class AbstractBaseService<T extends DBEntity,
3                                            R extends JpaRepository<T, Long>> {
4
5      @Inject
6      protected R repository;
7
8      public void save(final T t) {
9        repository.save(t);
10     }
11
12     public List<T> findAll() {
13       return repository.findAll();
14     }
15
16 }
```

Da die Methoden in einer konkreten Service-Klasse üblicherweise in einem eigenen Transaktionskontext ausgeführt werden, annotieren wir die Klasse mit der bereits bekannten Annotation `@Transactional` (Achtung: aus dem Paket `jakarta.transaction`). Damit können wir uns das in konkreten Unterklassen sparen.

Die Klasse ist abstrakt und parameterisiert über zwei Typen. Der Typ `T` spezifiziert die zu verwaltende Entität und muss ein Untertyp von `DBEntity` sein. Der Typ `R` spezifiziert den Typ des zu verwendenden Repositories und muss ein Untertyp von `JpaRepository` derselben Entität mit dem Primärschlüssel vom Typ `Long` sein.

Das Repository selbst wird von CDI injiziert (Zeilen 5 und 6).

Wie in den bisher bekannten Services delegieren wir die Aufrufe von *save()* und *findAll()* direkt an das Repository.

Jetzt müssen unsere konkreten Services nur noch diesen Basis-Service geeignet erweitern. Der `StudentService` ändert sich jetzt zu:

```
1 public class StudentService extends AbstractBaseService<Student,
2                                                          StudentRepository> {
3 }
```

Der `UserService` sieht dann so aus:

```
1 public class UserService extends AbstractBaseService<User, UserRepository> {
2
3     public Optional<User> findByEmailAndPassword(final String email,
4                                                  final String password) {
5         return repository.findByEmailAndPassword(email, password);
6     }
7
8 }
```

Da wir das Attribut für das Repository im Basis-Service mit dem Zugriffsmodifikator `protected` versehen haben, steht es den Unterklassen direkt zur Verfügung (dass `protected` durch den erlaubten Zugriff von beliebigen Klassen innerhalb desselben Pakets eigentlich konzeptionell „kaputt“ ist, können wir hier ignorieren, da wir im Unterpaket `service` ohnehin nur Service- und damit Unterklassen von `BaseService` haben sollten).

Aufgabe 2 Passwörter schützen

Wir haben auf Blatt 2 versprochen, dass wir uns um die Passwort-Verschlüsselung später kümmern und jetzt ist es soweit.

Es gibt – wie so oft – diverse Möglichkeiten, die Passwörter der Nutzer:innen zu „schützen“. Eine Methode, die in der Vergangenheit häufig Anwendung gefunden hat, besteht darin, dass Passwort mit Hilfe der Datenbank oder anderen Methoden zu *verschlüsseln*. Das wäre zwar besser als das Passwort im Klartext in der Datenbank zu speichern, aber wirklich sinnvoll ist das nicht. Denn Verschlüsselung bedeutet ja, dass das Passwort auch wieder *entschlüsselt* werden kann. Und das ist nicht mehr zeitgemäß. Aktuell werden stattdessen Passwort-*Hashes* in der Datenbank abgelegt.

Aus einem Passwort-Hash kann das ursprüngliche Passwort nicht ermittelt werden, da das Verfahren so ausgelegt ist, dass es nur in eine Richtung funktioniert (vorausgesetzt das Konzept und die Implementierung sind fehlerfrei in dieser Hinsicht).

Wenn ein Passwort aber nicht entschlüsselt werden kann – wie ist dann ein Login überhaupt möglich? Ganz einfach: das Passwort, das die Nutzer:in in der Anmeldemaske eingibt, wird mit dem gleichen Verfahren gehasht und mit dem Passwort-Hash in der Datenbank verglichen. Stimmen die Hashes überein, wird das eingegebene Passwort als korrekt angenommen.

Um anhand eines Passwort-Hashes das ursprüngliche Passwort zu bestimmen, müssten also alle möglichen Passwörter der Reihe nach *gehasht* werden und dann die Hashes mit dem Passwort-Hash in der Datenbank verglichen werden. Wenn sie übereinstimmen, ist ein Passwort ermittelt. Das muss dann auch nicht zwangsläufig das ursprüngliche Passwort sein, sondern ggf. ein Passwort, das lediglich den gleichen Hash hat.

Das Hashing-Verfahren sollte also so gestaltet sein, dass es möglichst wenig Kollisionen verursacht. Ein Hashing-Verfahren der Art *wandle die Buchstaben in Zahlen um gemäß der Position im Alphabet und addiere die Zahlen* wäre also ziemlich ungeeignet. Denn dort haben z. B. die Passwörter *abc* (1+2+3→6) und *ae* (1+5=6) den gleichen Hash-Wert.

Es empfiehlt sich, zum Hashen ein Standard-Verfahren und eine passende Bibliothek zu verwenden. Es ist niemals eine gute Idee, sich ein eigenes Verfahren auszudenken und zu implementieren.

Ein in diesem Sinne gutes Hashing-Verfahren genügt allerdings noch nicht, denn die Passwort-Hashes sollten zusätzlich auch noch *gesalzen* werden.

Passwort-Hashes werden gesalzen, um die Sicherheit von Passwörtern zu erhöhen und Angriffe wie *Rainbow Tables* und vorhersehbare Hash-Werte zu erschweren. Hier sind die Gründe, warum Salzen wichtig ist:

Verhinderung von Rainbow-Tables: Rainbow Tables sind vorberechnete Tabellen von Passwort-Hashes, die es Angreifern ermöglichen, Hash-Werte schnell mit den vorberechneten Hash-Werten abzugleichen und das ursprüngliche Passwort zu ermitteln. Wenn Passwörter gesalzen sind, müssten Angreifer für jede Salzkombination eine separate Rainbow Table erstellen, was zeitaufwändiger ist.

Erschwerung von Brute-Force-Angriffen: Bei Brute-Force-Angriffen versucht ein Angreifer, Passwörter durch systematisches Ausprobieren aller möglichen Kombinationen zu erraten. Wenn Passwörter gesalzen sind, muss der Angreifer nicht nur das Passwort selbst erraten, sondern auch die zugehörige Salzinformation kennen, um den Hash-Wert richtig zu berechnen.

Schutz vor Passwort-Wiederverwendung: Viele Benutzer verwenden dasselbe Passwort für mehrere Dienste. Wenn Passwörter gesalzen sind, führt dies dazu, dass selbst bei identischen Passwörtern unterschiedliche Hash-Werte erzeugt werden, da das Salz unterschiedlich ist. Dadurch wird vermieden, dass ein gehacktes Passwort in mehreren Diensten verwendet werden kann.

Erhöhte Sicherheit: Das Salzen von Passworten erhöht die Sicherheit, da es die Komplexität der Passwort-Hashes erhöht. Selbst wenn zwei Benutzer dasselbe Passwort haben, werden ihre Hash-Werte aufgrund unterschiedlicher Salze unterschiedlich sein.

Die Verwendung von zufälligen und eindeutigen Salzen für jedes Passwort ist bewährte Sicherheitspraxis in der Passwortverwaltung und trägt wesentlich dazu bei, die Sicherheit von Benutzerkonten zu gewährleisten.

Wir entscheiden uns hier für das Verfahren *scrypt*[1], für das es ausgereifte Java-Bibliotheken gibt.

[1] `http://www.tarsnap.com/scrypt/scrypt.pdf`

Bindet eine entsprechende Abhängigkeit in die `pom.xml` ein:

```
...
<scrypt.version>1.4.0</scrypt.version>
...
<dependency>
    <groupId>com.lambdaworks</groupId>
    <artifactId>scrypt</artifactId>
    <version>${scrypt.version}</version>
</dependency>
```

Wir können jetzt zum Hashen und zur Überprüfung des Hashes die Klasse `SCryptUtil`[2] und darin die Methoden *scrypt()* und *check()* verwenden.

Stellt sich noch die Frage, wo wir die Methoden dann aufrufen. Die Erzeugung des Hashes muss logischerweise **vor** dem Speichern in die Datenbank passieren. Da könnte es naheliegend sein, die Setter-Methode für das Passwort so zu ändern, dass sie das Passwort mittels *SCryptUtil#scrypt* hasht und dann erst dem eigenen Attribut zuweist. Das wäre allerdings problematisch, wenn ein bereits gehashtes Passwort über die Setter-Methode gesetzt werden soll – denn dann würde es ein zweites Mal gehasht. Das würde passieren, wenn ein `User`-Objekt aus der Datenbank gelesen wird – denn dann verwendet JPA die Setter-Methoden, um die Werte aus der Datenbankzeile den Attributen zuzuweisen.

Zudem wäre diese Variante vor dem Hintergrund von *separation-of-concerns* fragwürdig. Die Modell-Klasse hat ja die Aufgabe, Daten zu tragen, Zugriff über Getter- und Setter-Methoden zu ermöglichen und ggf. die Argumente an die Setter auf Plausibilität zu überprüfen. Das Hashen bestimmter Daten würde die Zuständigkeiten übersteigen und Logik in die Klasse bringen, die dort nichts verloren hat.

Es ist daher sinnvoll, das „Wissen“ über Hashing und die konkreten Details in der passenden Service-Klasse unterzubringen – auch wenn die Service-Klassen eigentlich für die Realisierung der Geschäftslogik gedacht sind und Hashing von Passwörtern eher nicht der Geschäftslogik zuzuschreiben sind.

Die Umsetzung hier wäre dann derart möglich, dass wir im `UserService` die *save()*-Methode erweitern:

```
@Override
public void save(final User user) {
    final String pwHash = SCryptUtil.scrypt(user.getPassword(), 4096, 8, 1);
    user.setPassword(pwHash);
    super.save(user);
}
```

Die *save()*-Methode des Service sollte jetzt nur für neue (d. h. noch nicht persistierte) Objekte verwendet werden (nicht etwa auch zum Aktualisieren eines Objektes). Ansonsten würde ein gehashtes Passwort erneut gehasht. Zum Aktualisieren eines Objektes sollte entsprechend eine Methode *update()* im `BaseService` implementiert werden, sobald die Funktionalität benötigt wird.

[2] `http://javadox.com/com.lambdaworks/scrypt/1.4.0/com/lambdaworks/crypto/SCryptUtil.html`

Dann müssen wir noch die Methode *findByEmailAndPassword()* anpassen:

```
public Optional<User> findByEmailAndPassword(final String email, final String
 ↪ password) {
    final Optional<User> candidate = repository.findByEmail(email);
    if (candidate.isPresent()) {
        final User user = candidate.get();
        if (SCryptUtil.check(password, user.getPassword())) {
            return candidate;
        }
    }
    return Optional.empty();
}
```

Hier können wir das übergebene Passwort nicht einfach hashen und dann den Hash für die Repository-Methode verwenden. Das liegt daran, dass der Hash ja mit einem zufälligen Salt versehen ist und das gleiche Passwort dadurch niemals den gleichen Hash bekommt. Also holen wir uns den User für die gegebene E-Mail-Adresse (die zugehörige Methode gibt es im Repository noch nicht – ihr müsst sie also noch selbst anlegen). Sollte es einen User mit der E-Mail-Adresse geben, prüfen wir über *SCryptUtil#check()*, ob das gegebene Passwort zu dem Passwort-Hash des Users passt. Wenn das so ist, geben wir den optionalen User zurück.

Deployt die Anwendung und probiert es aus. Eine Anmeldung sollte immer noch möglich sein und in der Liste der Nutzer:innen sollten jetzt kryptische Passwort-Hashes angezeigt werden.

Wen es stört, dass das Hashing des Passwortes jetzt in der Geschäftslogik eigentlich an der falschen Stelle gemacht wird, der kann sich das freiwillige(!) Zusatzblatt `ScryptDecorator` anschauen und die Vorschläge darin umsetzen.

Falls ihr das Zusatzblatt nicht bearbeiten möchtet, kann die Methode *findByEmailAndPasswort()* jetzt aus dem `UserRepository` entfernt werden, da sie nicht mehr benötigt wird.

Aufgabe 3 PrimeFaces: Wofür werden Frontend-Designer:innen eigentlich noch gebraucht?

In dieser Aufgabe wollen wir uns nun dem Frontend unserer Anwendung widmen. Es gibt für Faces umfangreiche Komponentenbibliotheken, die euch bei der Umsetzung moderner und funktionsreicher (im Sinne von Nutzer:innen-Interaktionen) Webseiten unterstützen. Unter den verfügbaren Bibliotheken ist *PrimeFaces*[3] sehr verbreitet. Hier einige Pro-Argumente:

- PrimeFaces bietet eine sehr große Auswahl an fertigen Komponenten für nahezu alle Standardfälle an – angefangen bei einfachen Eingabefeldern für Passwörter bis hin zu ganzen Kalender-Komponenten (*Schedule* genannt).
- Viele Komponenten bieten über ihren grundlegenden Funktionsumfang hinaus eine Vielzahl von zusätzlichen Konfigurationsmöglichkeiten an. Beispielsweise unterstützen PrimeFaces-Tabellen *out-of-the-box* das Sortieren und Filtern von Einträgen (und vieles mehr) – einfach durch Aktivierung der entsprechenden Option.
- Das Aussehen der Komponenten kann bequem über fertige Themes konfiguriert werden. PrimeFaces bietet sowohl kostenpflichtige (für SWP aus offensichtlichen Gründen nicht

[3] `https://www.primefaces.org`

nutzbar) wie auch kostenfreie Themes an. Für die Experten unter euch: PrimeFaces unterstützt*angeblich* auch *jQuery ThemeRoller*[4]. Wir haben es selbst zwar nicht getestet, für den einen oder anderen mag es aber durchaus interessant sein. Außerdem können alle Komponenten separat über CSS angepasst werden – PrimeFaces dokumentiert die verwendeten CSS-Klassen und -Ids sehr ausführlich.

- *Responsive Design* (also das automatische Anpassen der Bedienoberfläche an die Bildschirmgröße des Endgerätes) lässt sich mit *Grid CSS*[5] (ähnlich zu Bootstrap) vergleichsweise einfach umsetzen.
- *Client Side Validation*: PrimeFaces kann aus Attributen, die mit Annotationen der Java Bean Validation annotiert und in Faces gebunden (`value`) sind, automatisch Code ableiten, der die Validierung von Eingabedaten **zusätzlich** im Client (dem Browser) vornimmt[6]. Somit werden unnötige HTTP-Requests verhindert (was sich auch positiv auf die Reaktionsgeschwindigkeit der Oberfläche auswirkt), ohne dass die Validierung von Daten doppelt (sowohl im Frontend wie auch im Backend) implementiert werden muss.
- PrimeFaces hat eine große Community und wird aktiv – und vor allem professionell – weiterentwickelt.
- Die Integration von *Ajax*[7] ist sehr gut gelungen.
- Neben JSF unterstützt PrimeFaces auch *Angular*, *React* und *Vue.js*. Somit geht das von euch über die Zeit hart erarbeitete Wissen nicht verloren, falls ihr mal *gezwungen* – kleiner Seitenhieb – werdet, eines dieser Frameworks zu verwenden.

Auf eine Einführung von Ajax wird hier bewusst verzichtet. Zu diesem Thema gibt es sehr viel (Online-)Literatur, die ihr heranziehen könnt. Ich empfehle jedem von euch ein Grundverständnis von Ajax in Faces (und PrimeFaces) aufzubauen, da diese Technik die Grundlage vieler moderner Webanwendungen bildet – es geht zwar auch irgendwie ohne, aber solche Anwendungen möchte niemand benutzen.

Eine Übersicht (hier *showcase* genannt) aller PrimeFaces-Komponenten findet ihr unter: `https://www.primefaces.org/showcase`.

Ihr solltest euch unbedingt einen Überblick über **alle** von PrimeFaces bereitgestellten Komponenten verschaffen! Insbesondere im Hinblick auf das SWP könnt ihr euch viel Arbeit ersparen, wenn ihr ein Gefühl dafür entwickelt, was – und vor allem wie – technisch umsetzbar ist. Achtet bei jeder Komponente auch auf die verschiedenen Konfigurationsmöglichkeiten.

Aufgabe 3.1 Endlich richtige Tabellen

Um PrimeFaces nutzen zu können, müsst ihr – wie immer – die entsprechende Abhängigkeit in euer Maven-Projekt eintragen:

[4] `https://jqueryui.com/themeroller`
[5] `http://primefaces.org/showcase/ui/panel/grid.xhtml`
[6] `http://primefaces.org/showcase/ui/csv/basic.xhtml`
[7] `https://www.w3schools.com/xml/ajax_intro.asp`

```
...
<primefaces.version>13.0.0</primefaces.version>
...
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>${primefaces.version}</version>
    <classifier>jakarta</classifier>
</dependency>
```

Über das Tag `classifier` kann eine bestimmte Variante angegeben werden, falls mehrere zur Auswahl stehen.

Auf folgender Webseite findet ihr den Faces-Namensraum, in dem die von PrimeFaces bereitgestellten Komponenten zu finden sind: `https://www.primefaces.org/gettingstarted`.

Als erstes sollt ihr – endlich – die Tabelle auf der Seite `users.xhtml` durch eine PrimeFaces-Tabelle (*DataTable*) ersetzen. Nutzt für die Spaltendeklarationen innerhalb der Tabelle die PrimeFaces-Komponente `column` und setzt sinnvolle Spaltennamen durch das XML-Attribut `headerText`.

Ersetzt die Tabelle für die Prüflinge ebenfalls durch eine PrimeFaces-Tabelle.

Baut jetzt auch die Login-Seite derart um, dass darin die PrimeFaces-Komponenten `OutputLabel`[8], `InputText`[9], `Password`[10] und `CommandButton`[11] zum Einsatz kommen. Ändert auch `h:messages` in `p:messages`.

Konfiguriert die Eingabefelder so, dass sie *required* sind – also eine Eingabe dort erfolgen muss. Wenn ihr jetzt die Felder leer lasst und direkt die Schaltfläche klickt, passiert allerdings nichts. Das liegt daran, dass das Standard-Verhalten des `CommandButton` (und aller anderen PrimeFaces-Komponenten) lautet, dass Ajax zu verwenden ist. Daher wird die Seite nach dem Klick nicht neu geladen und eine eventuelle Fehlermeldung nicht angezeigt. Setzt also das Attribut `ajax` im `CommandButton` auf `false` und probiert es nochmal. Jetzt solltet ihr eine entsprechende Fehlermeldung sehen, wenn beide oder ein Eingabefeld leer ist oder ihr ungültige Anmeldedaten eingebt.

Ihr könnt auch das Attribut `ajax` im `CommandButton` wieder löschen bzw. auf `true` und dann das Attribut `update` auf den Wert `@form` setzen. Mit dem `update`-Attribut wird festgelegt, welche Teile der Seite neu gerendert werden. Durch den Wert `@form` wird das umschließende Formular komplett neu gerendert. Vergebt auch mal eine Id für die `messages`-Komponente und verwendet diese Id direkt (also ohne das `@`) im `update`-Attribut. Seht ihr den Unterschied und versteht ihr woher er kommt? Falls nicht, Tutor:in fragen!

Wenn ihr Spaß daran habt, solltet ihr euch die Code-Beispiele für die Facelets auf den Showcase-Seiten der Komponenten ansehen und die Seite ggf. ein wenig (oder auch mehr) „aufhübschen“.

[8] `https://www.primefaces.org/showcase/ui/misc/outputLabel.xhtml`
[9] `https://www.primefaces.org/showcase/ui/input/inputText.xhtml`
[10] `https://www.primefaces.org/showcase/ui/input/password.xhtml`
[11] `https://www.primefaces.org/showcase/ui/button/commandButton.xhtml`

Wenn euch etwa die Anordnung der Labels und Eingabefelder so nervt wie uns, könnt ihr das z. B. über *PanelGrid*[12] korrigieren.

Aufgabe 3.2 Layout Template

Nachdem unsere Login-Seite jetzt zumindest besser aussieht als vorher, werden wir jetzt ein Template für das Layout der Seiten „hinter“ dem Login einrichten. Wir streben hier eine vertikale Dreiteilung an in einen Kopfbereich, einen Inhaltsbereich und einen Fußbereich.

Erstellt das Verzeichnis `templates` unterhalb des Verzeichnisses `WEB-INF` und darin dann die Datei `layout.xhtml` mit dem folgenden Inhalt:

```
1  <!DOCTYPE html>
2      <html lang="en"
3          xmlns:h="jakarta.faces.html"
4          xmlns:ui="jakarta.faces.facelets"
5      >
6      <h:head>
7          <title>#{title}</title>
8          <link rel="stylesheet"
           ↪ href="https://unpkg.com/primeflex@^3/primeflex.css" />
9      </h:head>
10     <h:body>
11         <header>
12             <ui:include src="/WEB-INF/includes/layout/header.xhtml" />
13         </header>
14         <main>
15             <ui:insert name="content" />
16         </main>
17         <footer>
18             <ui:include src="/WEB-INF/includes/layout/footer.xhtml" />
19         </footer>
20     </h:body>
21 </html>
```

In Zeile 1 fehlt zunächst mal die Angabe der DTD. Die lassen wir hier weg, weil wir eigene Tags verwenden möchten, die in der allgemeinen XHTML-DTD nicht vorgesehen sind und IntelliJ anderenfalls Parse-Fehler anzeigt.

In Zeile 4 binden wir zusätzlich zu dem bereits bekannten `h`-Namespace für die HTML-Tags auch noch den `ui`-Namespace für die Faces-Tags mit ein.

Da das Template den Rahmen einer Webseite darstellt, benötigt es ein `head`- und ein `body`-Tag (Zeilen 6 und 10). Im Head-Bereich setzen wir den Titel der Seite auf einen EL-Ausdruck (Zeile 7). Darüber kann eine konkret eingebundene Seite später den Titel konfigurieren. Da wir für die Seitengestaltung *PrimeFlex*[13] verwenden möchten, referenzieren wir in Zeile 8 auf das entsprechende Stylesheet[14]. Im Body der Seite nehmen wir unsere gewünschte vertikale Dreiteilung vor. Dabei wird im Kopfbereich (eigenes Tag `header` in Zeile 11) eine entsprechende Datei inkludiert (Zeile 12). Für den Fußbereich (eigenes Tag `footer` in Zeile 17) wird analog auch dafür eine

[12] `https://www.primefaces.org/showcase/ui/panel/panelGrid.xhtml`
[13] `https://primeflex.org/`
[14] `https://primeflex.org/installation`

Datei inkludiert. Der Inhaltsbereich (eigenes Tag `main` in Zeile 14) fügt über `ui:insert` den eigentlichen Inhalt ein.

<ui:insert>: wird normalerweise in Verbindung mit einem Template verwendet und dient dazu, Platzhalter innerhalb des Templates zu erstellen, in die andere Seiten (Fragments) eingefügt werden können, die konkrete Inhalte bereitstellen.

<ui:include>: wird verwendet, um eine separate Faces-Seite in eine andere Seite zu importieren oder einzuschließen. Es ermöglicht die Wiederverwendung von Seiten oder Teilen von Seiten.

Erstellt entsprechende Unterverzeichnisse in `WEB-INF`.

Das Verzeichnis `WEB-INF` gehört nicht zum öffentlichen Dokumentbaum der Webanwendung. Daher dürfen Dateien, die unterhalb dieses Verzeichnisses existieren, nicht vom Container direkt an Clients gesendet und somit nicht von diesen eingesehen werden.

Erzeugt die Datei `header.xhtml`, so dass deren Pfad zu dem *include* (Zeile 12 oben) passt. Sie sollte zunächst folgenden Inhalt haben:

```
1  <ui:composition
2      xmlns:h="jakarta.faces.html"
3      xmlns:p="http://primefaces.org/ui"
4      xmlns:ui="jakarta.faces.facelets"
5  >
6
7      <div class="flex justify-content-between">
8          <p:linkButton value="Eksadat" href="#{request.contextPath}/"/>
9          <h:form>
10             <p:commandButton value="Logout" icon="pi pi-fw pi-sign-out"
               ↪ action="#{logoutController.logout()}" />
11         </h:form>
12     </div>
13
14 </ui:composition>
```

Das Tag `ui:composition` in Zeile 1 legt dabei eine Gruppierung von Elementen fest. So eine Komposition wird häufig im Kontext von Templates verwendet – das muss aber nicht so sein. Wir binden in Zeile 2 und 3 wieder die Namespaces für die zu verwenden Tags ein. In Zeile 6 definieren wir eine `div`, die jetzt einen Style aus dem oben erwähnten PrimeFlex-Stylesheet verwendet. In diesem Fall beschreibt die Klasse die Ausrichtung der in ihr enthaltenen Komponenten: `justify-content-between` bedeutet, dass der gesamte aktuell zur Verfügung stehende Platz in horizontaler Richtung ausgenutzt wird und dadurch die Komponenten gleichmäßig verteilt werden. In unserem Fall gibt es in der `div` zwei Komponenten. Dadurch wird die erste Komponente ganz nach links und die zweite Komponente ganz nach rechts gesetzt.

Bei der ersten Komponente handelt es sich um einen `LinkButton`[15], der als Schaltfläche dar-

[15] `https://www.primefaces.org/showcase/ui/button/linkButton.xhtml`

gestellt und zur Navigation durch Aufruf von URLs (per HTTP GET Request) verwendet wird. In Zeile 7 wird das Ziel direkt im Attribut `href` angegeben. Es handelt sich dabei um einen EL-Ausdruck, der das implizite EL-Objekt `request` verwendet, welches auf die aktuelle `HttpServletRequest`-Instanz verweist. Der `contextPath` ist die Basis-URL der Webanwendung. Wird diese Schaltfläche also betätigt, wird die Startseite der Webanwendung geladen und angezeigt.

Die zweite Komponente in Zeile 8 ist ein Formular, das ein `commandButton` von PrimeFaces enthält. Es bekommt neben der Zeichenkette `Logout` auch noch ein passendes PrimeFaces-Icon[16]. Wenn diese Schaltfläche angeklickt wird, wird die Methode *logout()* des `LogoutController`s aufgerufen. Da ein Formular üblicherweise das Facelet erneut per *Postback* lädt, wird der Rückgabewert der Methode *logout()* hinsichtlich der Navigation ignoriert. Wir müssen daher die Methode *logout()*-Methode der `LogoutController` anpassen.

Lasst euch zunächst den externen Kontext injizieren:

```
@Inject
private ExternalContext externalContext;
```

Ändert dann den Rückgabetyp der Methode in `void`. Statt einer Rückgabe tragt ihr am Ende der Methode folgendes ein:

```
externalContext.redirect(request.getContextPath());
```

Hierdurch wird am Ende der Methode ein *redirect*, d. h. ein Laden der URL im Argument, ausgelöst. Die URL lassen wir uns wie oben über den `contextPath` des Requests geben – nur hier als Java-Anweisung und nicht als EL-Ausdruck.

Die Methode *redirect()* kann eine `IOException` auslösen. Delegiert diese an den Aufrufer, indem ihr die *logout()*-Methode mit einer entsprechenden `throws`-Klausel verseht.

Erzeugt jetzt die Datei `footer.xhtml`, so dass deren Pfad zu dem *include* (Zeile 18 oben) passt. Sie sollte zunächst folgenden Inhalt haben:

```
1  <ui:composition
2      xmlns:ui="jakarta.faces.facelets"
3      xmlns:p="http://primefaces.org/ui"
4  >
5
6      <div class="flex justify-content-center">
7          <p:outputLabel value="(C) 2023 SWP"/>
8      </div>
9
10 </ui:composition>
```

Die einzige Komponente (Zeile 7) in der einzigen `div` zentrieren wir hier über die Klasse `justify-content-center` von PrimeFlex.

Aufgabe 3.3 Das Layout benutzen

Damit unser Layout auch tatsächlich verwendet wird, müssen wir unsere beiden Seiten `users.xhtml` und `students.xhtml` anpassen. Wir beschreiben das hier beispielhaft für die Seite `users.xhtml`.

[16] `https://www.primefaces.org/showcase/icons.xhtml`

Entfernt zunächst das Tag `!DOCTYPE`. Ersetzt dann das `html`-Tag wie folgt:

```
<ui:composition template="/WEB-INF/templates/layout.xhtml"
    xmlns:p="http://primefaces.org/ui"
    xmlns:ui="jakarta.faces.facelets"
>
```

Hiermit teilen wir Faces mit, dass das Facelet eine Komposition definiert, die im Rahmen des spezifizierten Templates verwendet wird. Denkt daran, das schließende `html`-Tag entsprechend durch ein schließendes `ui:composition`-Tag zu ersetzen.

Der gesamte `h:head`-Zweig wird ersetzt durch folgendes:

```
<ui:param name="title" value="Users"/>
```

Damit weisen wir der Variablen `title` den Wert `Users` zu. Wir erinnern uns: die Variable `title` wird vom Template verwendet, um den Seitentitel zu setzen.

Der eigentliche Inhalt unserer Webseite wird jetzt nicht mehr im `h:body`-Zweig angegeben (denn das Template definiert ja bereits einen solchen Zweig), sondern in einem `ui:define`-Zweig:

```
<ui:define name="content">
```

Durch den Wert im Attribut `name` „weiß“ das Template, wo der folgende Inhalt eingesetzt werden soll. Denkt auch hier daran, das schließende `h:body`-Tag geeignet zu ersetzen.

Passt jetzt das Facelet `students.xhtml` analog an und überprüft das Ergebnis im Browser (probiert auch die Schaltflächen aus). Ändert insbesondere die Größe des Browser-Fensters für die Seiten, die auf dem Template aufbauen. Die Anpassung der Schaltflächen in der Kopfzeile und der Schrift in der Fußzeile erfolgen durch die Verwendung von PrimeFlex.

Das Facelet `logout.xhtml` wird jetzt nicht mehr benötigt – löscht es also.

Aufgabe 4 Prüfungen

In unserem System zur Verwaltung von Prüfungen muss es natürlich auch Prüfungen geben. In unserem stark vereinfachten Projekt gehen wir jetzt davon aus, dass eine Prüfung neben einem Namen ein Datum und eine Uhrzeit enthält und genau eine Nutzer:in mit Rolle `USER` als Prüfer:in und genau eine Student:in als Prüfling. Natürlich darf jede Student:in an beliebig vielen Prüfungen teilnehmen und jede Nutzer:in darf beliebig viele Prüfungen als Prüfer:in abnehmen. Das folgende Diagramm veranschaulicht diese Beziehung:

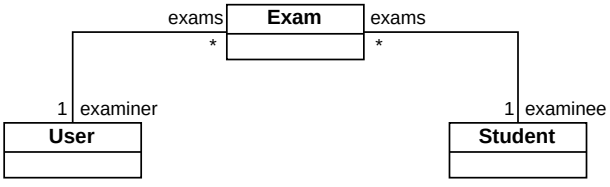

Abbildung 1: Klassendiagramm für Prüfungen

Aufgabe 4.1 Prüfungsobjekte

Erstellt also zunächst eine Klasse `Exam`, die eine Prüfung repräsentiert. Da wir Prüfungen auch persistieren möchten, handelt es sich bei einer Prüfung um eine `DBEntity`. Legt ein `String`-Attribut für den Namen der Prüfung an. Für Datum und Uhrzeit verwendet ihr ein einziges Attribut mit dem Typ `LocalDateTime`[17].

Da eine Student:in an mehreren Prüfungen teilnehmen darf, könnten wir jetzt auf die Idee kommen, der Klasse `Student` eine Sammlung von Prüfungen (z. B. `List<Exam>`) hinzuzufügen. Aber wir erinnern uns an die Lazy-Loading Problematik und wählen daher hier den Weg, den wir bei 1-zu-n Beziehungen immer gehen sollten: in Richtung der Multiplizität 1.

Sehr wahrscheinlich ist es im Kontext der Anwendung interessant, welche Prüfungen eine Student:in tatsächlich hat – spätestens wenn sich Student:innen selbst am System anmelden können und dann die Liste ihrer Prüfungen einsehen möchten. Daher erscheint es naheliegend, zusätzlich eine Sammlung von Prüfungen in der Klasse `Student` zu deklarieren. Wenn wir das aber tun, müssen wir sehr sorgfältig mit Veränderungen umgehen. Wenn sich eine Student:in dann z. B. von einer Prüfung abmeldet, müssen wir die Student:in aus dem Attribut in der Prüfung entfernen und natürlich auch die Prüfung aus der Sammlung der Prüfungen der Student:in. Wir haben dann also immer mindestens zwei Stellen im Quellcode, die wir aktuell und konsistent halten müssen. Das ist sehr fehleranfällig, daher raten wir dazu, Beziehungen zwischen Entitäten wann immer möglich *unidirektional* zu definieren und zu nutzen. Wir müssen dann in unserem Fall zwar eine Datenbank-Abfrage absetzen, um alle Prüfungen einer Student:in zu erhalten (statt einfach das Sammlungsattribut auszulesen), aber ein DBMS ist dafür hervorragend geeignet und sogar optimiert.

Legt also ein Attribut für den Prüfling vom Typ `Student` und ein Attribut für die Prüfer:in vom Typ `User` an.

Da es sich dabei nicht um integrale Datentypen handelt, die direkt in einer Tabellenspalte gespeichert werden können, müssen wir jetzt eine Fremdschlüssel-Beziehung herstellen. Da eine Student:in an beliebig vielen Prüfungen teilnehmen darf, verwenden wir eine JPA-Annotation für die n-zu-1 Beziehung direkt über der Attributdeklaration:

```
@ManyToOne
private Student examinee;
```

Neben `@ManyToOne` gibt es noch weitere Annotationen, die die Beziehungen zwischen Entitäten definieren. Ihre Namen `OneToOne`, `OneToMany`, `ManyToMany` sind treffend gewählt. Ausführliche Erläuterungen dazu findet ihr unter `https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1#a538`.

Verseht auch das Attribut für die Prüfer:in mit einer entsprechenden Annotation.

Wenn ihr eure Anwendung jetzt deployt und dann in die Tabellenstruktur schaut, solltet ihr feststellen, dass JPA eine Tabelle `EXAM` erzeugt hat mit je einem Fremdschlüssel zu Tabellen `EDUSER` und `STUDENT`.

[17] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/LocalDateTime.html`

Aufgabe 4.2 Repository und Service

Damit wir auf die in unserer Architektur üblichen Art mit den Prüfungen arbeiten können, legt ihr jetzt ein Repository und einen Service für Prüfungen an.

Aufgabe 4.3 Liste von Prüfungen anzeigen

Erstellt ein Facelet mit Backing Bean, um alle Prüfungen anzuzeigen (egal, wer die Prüfer:in ist).

Um nicht die URL im Browser manuell eingeben zu müssen, erweitert ihr das Facelet `hello.xhtml` um einen weiteren `outputLink`, so dass angemeldete Nutzer:innen mit der Rolle `USER` diesen sehen und verwenden können.

Um die beiden Links für die Nutzer:innen untereinander anzuzeigen, könnt ihr sie jeweils in einen Paragraphen setzen. Ein Paragraph wird durch die HTML-Tags `<p>` und `</p>` definiert.

Aufgabe 4.4 Prüfungen in der UI anlegen

Leider ist die Anzeige der Prüfungen eher langweilig, weil es keine gibt. Wir könnten natürlich wie bei den Student:innen und Nutzer:innen in unserer Startup Bean einige Prüfungen anlegen, aber wir möchten das jetzt über Funktionalität in der UI erledigen (zum Prüfen eurer Anzeige und der Kommunikation mit der Datebank könnt ihr natürlich trotzdem Prüfungen im Startup Code anlegen).

Aufgabe 4.4.1 Facelet and Backing Bean

Ergänzt das Facelet über oder unter der Prüfungsliste um eine Schaltfläche für das Hinzufügen neuer Prüfungen:

```
<p:linkButton icon="pi pi-plus" value="create new exam" outcome="create_exam"
 ↪ style="margin-top: 10pt"/>
```

Die Schaltfläche bekommt ein Plus-Symbol als Icon. Im `style`-Attribut wird festgelegt, dass das Element einen Abstand nach oben von 10 Punkten hat – dadurch wird es nicht direkt unter die Tabelle gesetzt (wenn ihr die Schaltfläche über die Liste setzt, könnt ihr `margin-bottom` für den Abstand nach unten verwenden).

Die Zeichenkette in `outcome` legt das Navigationsziel innerhalb der Dokumentenstruktur der Webanwendung fest. Hier wird automatisch davon ausgegangen, dass sich das Ziel im selben Verzeichnis befindet. Wir benötigen jetzt also ein neues Facelet im Unterordner `user` mit dem Namen `create_exam.xhtml`. Es beginnt mit dem üblichen Facelet-Header:

```
1 <ui:composition template="/WEB-INF/templates/layout.xhtml"
2     xmlns:h="jakarta.faces.html"
3     xmlns:p="http://primefaces.org/ui"
4     xmlns:ui="jakarta.faces.facelets"
5     xmlns:f="jakarta.faces.core"
6 >
```

Dann legen wir den Titel fest:

```
8  <ui:param name="title" value="Create an exam"/>
```

Dann geben wir an, dass wir den `content`-Bereich des Templates füllen möchten, setzen eine Überschrift und beginnen ein Formular:

```
10 <ui:define name="content">
11     <h1>New Exam</h1>
12     <h:form>
```

Unser Formular benötigt Eingabemöglichkeiten für alle Bestandteile einer Prüfung. Wir definieren der Einfachheit halber (schön ist anders) ein responsives Gitter mit drei Spalten (Label, Eingabefeld, Nachricht):

```
13 <p:panelGrid columns="3" layout="flex" contentStyleClass="align-items-baseline
    ↪ ui-fluid">
```

Als Layout wählen wir das schon bekannte *PrimeFlex* und weisen dem Gitter zusätzlich die Stile `align-items-baseline` und `ui-fluid` zu. Der erste Stil richtet die Grundlinien der Elemente im Gitter an einer horizontalen Linie aus. Der zweite Stil sorgt dafür, dass die Elemente sich im Gitter den gesamten zur Verfügung stehenden Platz nehmen. Bei drei Spalten bedeutet das, dass jedes Element ein Drittel des Platzes einnimmt. Der Platz wird hierbei durch die Größe des Browserfensters festgelegt und ändert sich bei einer Größenänderung des Browsers. Wird das Browserfenster in der Breite soweit verkleinert, dass nicht mehr alle Spalteninhalte nebeneinander passen, so werden sie untereinander dargestellt.

Als erstes hat eine Prüfung bei uns einen Namen. Wir legen also ein Textfeld zusammen mit einem Texteingabefeld sowie einem Nachrichtenfeld dafür an:

```
14 <p:outputLabel for="name" value="Name"/>
15 <p:inputText id="name" value="#{createExamController.name}"/>
16 <p:message for="name"/>
```

Der in der UI eingegebene Name soll im Attribut `name` eines Objektes vom Typ `CreateExamController` gespeichert werden. Legt also im entsprechenden Verzeichnis/Paket eine passende Backing Bean an.

Da eine Prüfung ein Datum und eine Uhrzeit hat, legen wir auch hierfür eine Eingabemöglichkeit in Form des `DatePickers` von PrimeFaces an:

```
17 <p:outputLabel for="date" value="Date and time"/>
18 <p:datePicker id="date" value="#{createExamController.dateTime}"
    ↪ showTime="true" locale="de" pattern="dd.MM.yyyy"/>
19 <p:message for="date"/>
```

Die UI der Webanwendung ist zwar aktuell in englischer Sprache, da es aber in Deutschland eingesetzt werden soll, stellen wir das Widget entsprechend ein (*Locale* und das hierzulande übliche Datumsformat in `pattern`). Legt das zugehörige Attribut mit dem bereits bekannten Datentyp `LocalDateTime` in der Backing Bean an.

Bleiben noch die Prüfer:in und die Student:in. Die Prüfer:in wird der Einfachheit halber die Nutzer:in, die die Prüfung anlegt. Dafür wird also keine Eingabemöglichkeit benötigt.

Da in unserem Beispielprojekt nur genau eine Student:in geprüft werden kann, wählen wir dafür ein Auswahlfeld, bei dem nur ein Eintrag[18] aus einer Liste ausgewählt werden kann:

```
20 <p:outputLabel for="examinee" value="Examinee"/>
21 <p:selectOneListbox id="examineeID" value="#{createExamController.examineeID}">
22   <f:selectItems value="#{createExamController.students}" var="student"
     ↪ itemLabel="#{student.email}" itemValue="#{student.id}"/>
23 </p:selectOneListbox>
24 <p:message for="examinee"/>
```

In Zeile 21 wird definiert, dass der in der *Listbox* ausgewählte Wert in das Attribut `examineeID` der Backing Bean übertragen wird. Legt dort also ein entsprechend benanntes Attribut vom Typ `String` an (denn die Listbox überträgt ohne weiteres nur Zeichenketten).

In Zeile 21 wird die Listbox konfiguriert. Die Auswahl der zur Verfügung stehenden Werte wird aus der Sammlung entnommen, die im Attribut `value` des `selectItems`-Facet angegeben ist. Das ist hier die Rückgabe einer Methode `getStudents()` in der Backing Bean. Legt also dort eine Methode an, die alle Student:innen im Datenbestand zurückgibt.

Da die Elemente, die in der Listbox zur Auswahl stehen, in einer Schleife ermittelt werden, legen wir im Attribut `var` fest, welchen Namen die Variable hat, die während der Iteration das aktuelle `Student`-Objekt enthält. Im Attribut `itemLabel` wird konfiguriert, was in der Listbox auf der Webseite konkret angezeigt wird. Hier wird der Einfachheit halber nur die E-Mail-Adresse der Student:in verwendet. Im Attribut `itemValue` wird konfiguriert, was im Fall der Auswahl konkret als Wert in die Backing Bean übertragen wird. Hier wird die Id der Entität übertragen. Denkbar wäre es aber auch, das gesamte Objekt als Wert zur Verfügung zu stellen – dann müsste allerdings ein *Konverter*[19] dafür erstellt und im `converter`-Attribut der Listbox angegeben werden.

Wenn ihr die Anwendung jetzt deployt und ausprobiert, dann sollten die E-Mail-Adressen aller im Datenbestand vorhandenen Student:innen in der entsprechenden Listbox angezeigt werden.

Die Anzeige der E-Mail-Adressen als Kriterium zur Auswahl ist natürlich nicht sinnvoll. Sinnvoller wäre vermutlich ein Anzeige-Label der Art: `Dozent, Dieter (dieter@offline.de)`. Das ist direkt mit der EL möglich[a].

[a] `https://jakarta.ee/specifications/expression-language/5.0`

Was jetzt noch fehlt, ist eine Schaltfläche zum Absenden des Formulars:

```
25 </p:panelGrid>
26 <p:commandButton value="save" icon="pi pi-check" ajax="false"
   ↪ action="#{createExamController.create}"/>
```

Wir beenden zunächst das `PanelGrid` und platzieren dann die Schaltfläche unterhalb des Grids. Sie bekommt ein Icon und den Text *save*. Ajax schalten wir wieder aus. Wird die Schaltfläche angeklickt, wird die Methode *create()* in der Backing Bean aufgerufen.

[18] `https://www.primefaces.org/showcase/ui/input/listbox.xhtml`

[19] `https://javadoc.io/doc/jakarta.faces/jakarta.faces-api/latest/jakarta/faces/convert/Converter.html`

Erstellt also eine entsprechende Methode in der Backing Bean:

```
public String create() {
    final Exam exam = new Exam();
    exam.setName(name);
    exam.setDateTime(dateTime);
```

Wir erzeugen hier zunächst ein neues Prüfungs-Objekt und setzen den Namen sowie Datum und Zeit. Die einzutragende Prüfer:in haben wir aber noch nicht. Wir können sie aus dem `ExternalContext` ermitteln über

```
@Inject
private ExternalContext externalContext;
...
externalContext.getRemoteUser()
```

Darüber erhalten wir den Login (bei uns die E-Mail-Adresse) des aktuell eingeloggten Users als `String`. Das nützt uns noch nichts, denn wir benötigen ja das `User`-Objekt. Unser `UserRepository` hat bereits eine Methode *findByEmail()*, die ein optionales `User`-Objekt zurückgibt. Damit wir aber unsere Architekturprinzipien einhalten, dürfen wir aus einer Backing Bean nicht direkt mit einem Repository kommunizieren. Stattdessen verwenden wir architekturkonform den `UserService`. Legt darin also eine entsprechende Methode *findByEmail()* an, die einfach den Rückgabewert des Repositories zurückgibt.

Lasst euch den `UserService` injizieren und erweitert die create()-Method:

```
final User examiner =
 ↪ userService.findByEmail(externalContext.getRemoteUser()).orElseThrow();
exam.setExaminer(examiner);
```

Wir verwenden hier der Einfachheit halber die Methode *orElseThrow()* für das zurückgegebene `Optional`, weil wir davon ausgehen, dass es die eingeloggte Nutzer:in im Datenbestand gibt. In einem ausgelieferten Produktivsystem müsste hier eine ausführlichere Fehlerbehandlung erfolgen als einfach eine Exception auszulösen.

Für die Student:in ergibt sich jetzt ein ähnliches Problem. Wir erhalten aus dem Facelet eine `String`-Repräsentation der Id des ausgewählten `Student`-Objektes. Wir benötigen aber auch hier das vollständige Objekt. Das können wir uns architekturkonform vom `StudentService` geben lassen. Auch hier hat das Repository bereits die passende Methode *findById()*), aber die Service-Klasse hat noch keine solche Methode. Wir könnten jetzt – wie oben – den `StudentService` direkt um eine entsprechende Methode erweitern, aber es ist sehr wahrscheinlich, dass auch andere Service-Klassen eine solche Methode benötigen werden. Also erweitern wir die Oberklasse `AbstractBaseService` folgendermaßen:

```
public Optional<T> findById(final Long id) {
    return repository.findById(id);
}
```

Damit erben jetzt alle konkreten Service-Klassen diese Methode.

Allerdings können wir hier nicht direkt den String übergeben, sondern müssen erst ein `Long`-Objekt daraus erzeugen:

```
exam.setExaminee(studentService.findById(Long.valueOf(examineeID))
                                                        .orElseThrow());
```

Auch hier verwenden wir aus den gleichen Gründen wie oben *orElseThrow()*, um an das Objekt im `Optional` zu gelangen.

Jetzt sind alle Attribute mit korrekten Werten versehen und wir können die Prüfung abspeichern:

```
examService.save(exam);
```

Nach dem Speichern möchten wir zurück zur Liste aller Prüfungen, daher führen wir ein Redirect auf das entsprechende Facelet durch, indem wir einen entsprechenden `String` zurückgeben:

```
return "exams.xhtml?faces-redirect=true";
```

Baut und deployt eure Anwendung und legt eine Prüfung mit sinnvollen Daten an. Es sollte problemlos funktionieren und ihr danach eure Prüfung in der Liste der Prüfungen sehen können. Die Darstellung ist allerdings noch wenig sinnvoll. Das liegt daran, dass die Datentabelle für Objekte deren *toString()*-Methode aufruft und deren Rückgabe ist noch nicht besonders gut lesbar. Um die Darstellung kümmern wir uns später.

Versucht jetzt eine Prüfung anzulegen in der ihr lediglich eine Student:in auswählt, d. h. ihr lasst das Eingabefeld für den Namen leer und wählt auch kein Datum aus. Wie ihr seht, funktioniert das problemlos und diese sinnlose Prüfung wird tatsächlich angelegt.

Jetzt erstellt eine komplett leere Prüfung, d. h. ihr lasst alle Eingabefelder ungenutzt – einfach direkt `create new exam` anklicken. Jetzt fliegt eine Exception und sie wird im Browser inklusive Stacktrace angezeigt.

In einem ausgelieferten Produktivsystem werden natürlich keine Exceptions und Stacktraces im Browser angezeigt. Dort wird im Fehlerfall auf konfigurierbare Fehlerseiten umgeleitet oder die Exceptions werden gefangen und eine entsprechende menschenlesbare Fehlermeldung angezeigt.

Das Problem ist hier, dass in der Methode *create()* der Backing Bean die Methode *Long.valueOf()* mit dem Argument `null` aufgerufen wird, wenn keine Student:in ausgewählt ist.

Wie stellen wir sicher, dass unsinnige (oder gar keine) Werte abgelehnt werden und die Nutzer:in ihre Eingaben revidieren kann?

Aufgabe 4.4.2 Validatoren

Um die eingegebenen Werte zu überprüfen, bietet Faces vorgefertigte *Validatoren*[20] an. Diese Validatoren werden im Facelet definiert:

```
<h:inputText value="#{someBean.someString}">
   <f:validateLength minimum="3" maximum="3" />
</h:inputText>
```

Hier wird durch den Validator sichergestellt, dass die Eingabe genau 3 Zeichen lang ist.

Wozu gibt es denn den Validator `f:validateRequired`? Die Eingabekomponenten haben doch bereits ein Attribut für `required`. Der Validator ist gedacht für zusammengesetzte Komponenten (*Composite Components*[a]).

[a] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0.html#a1515`

[20] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0.html#a1446`

Die vorhandenen Validatoren sind in vielen Fällen ausreichend – insbesondere `f:validateRegex` ist sehr mächtig wenn es um die kontextunabhängige Überprüfung einer Eingabe geht – aber es könnte sein, dass eine Eingabe aufgrund ihres Kontextes nicht isoliert geprüft werden kann. Daher können auch eigene Validatoren implementiert werden.

Ein Beispiel aus der Praxis: Wenn sich jemand mit E-Mail-Adresse für eine Webseite registriert hat und die E-Mail-Adresse später ändern möchte, sollte sichergestellt sein, dass die neu eingegebene E-Mail-Adresse nicht bereits im Datenbestand existiert. Das kann über einen eigenen Validator dann z. B. so geprüft werden:

```
1  @FacesValidator(value="de.unibremen...UniqueEmailValidator", managed=true)
2  public class UniqueEmailValidator implements Validator<String> {
3
4    @Inject
5    private UserService userService;
6
7    @Override
8    public void validate(FacesContext ctx, UIComponent cmpt, String email)
9    throws ValidatorException {
10     if (email == null || email.isEmpty()) {
11       return; // Let @NotNull or required=true handle this.
12     }
13     String oldEmail = (String) ((UIInput) cmpt).getValue();
14     if (!email.equals(oldEmail) && userService.exists(email)) {
15       throw new ValidatorException(new FacesMessage("Email already in use"));
16     }
17   }
18 }
```

Wir müssen dazu das `Validator`-Interface implementieren (Zeile 2) und den Validator entsprechend als solchen anmelden (über die Annotation in Zeile 1). Die Überprüfung, ob es die eingegebene E-Mail-Adresse bereits gibt, darf natürlich nur erfolgen, wenn beim Editieren der Daten die E-Mail-Adresse auch tatsächlich geändert wurde. Daher wird der alte Wert (vor der Änderung) aus der zu validierenden Komponente des Facelets extrahiert (Zeile 13) und die Überprüfung nur durchgeführt, wenn sich die E-Mail-Adresse geändert hat (Zeile 14).

Bevor wir jetzt allerdings über Validatoren die Eingabe unsinniger Werte verhindern, noch ein kleiner Exkurs:

Aufgabe 4.4.3 Bean Validation

Die Faces Validatoren aus dem letzten Abschnitt kommen zum Einsatz, wenn die Eingaben aus einem Facelet überprüft werden sollen. Es kann in der Praxis allerdings sinnvoll sein, die Eingaben auch generell zu überprüfen. Stellen wir uns ein System vor, an dem einerseits Daten über Facelets im Browser eingeben werden können, aber gleichzeitig die Möglichkeit der Übermittlung von Daten z. B. durch eine REST-Schnittstelle gegeben ist. Dann müssten wir die Überprüfung an zwei Stellen konsistent durchführen. Zusätzlich könnten – wie in unserer Startup Bean beim Erzeugen der User und Studis – im Quellcode direkt Daten bereitgestellt werden. Und auch diese sollten natürlich überprüft werden, damit nicht ein Programmierfehler zu fehlerhaften Daten im

Datenbestand führt. Das wären dann also schon drei Stellen, an denen Daten konsistent validiert werden müssen.

Um eine zentrale Stelle zu schaffen, um Eingabewerte zu überprüfen, bietet Faces die Möglichkeit der *Bean Validation*[21]. Constraints für Eingaben können dabei durch Annotationen festgelegt werden. Ihr findet eine Übersicht in der Spezifikation[22].

Die Annotationen werden in unserem Fall direkt über die entsprechenden Attribute geschrieben, z. B. in der Backing Bean:

```
@NotBlank
@Size(min = 3, max = 20)
private String name;
```

Das bedeutet dann, dass die Eingabe nicht nur aus Whitespaces bestehen darf und mindestens 3 Zeichen aber höchstens 20 Zeichen lang ist.

Setzt Annotationen für Datum und Uhrzeit, so dass das Attribut nicht `null` sein darf und ein Datum in der Zukunft verlangt.

Setzt eine Annotation für die Id des Prüflings, die leere Zeichenketten verhindert.

Stellt dabei sicher, dass die Annotationen aus dem Paket `jakarta.validation.constraints` kommen.

Optional: Validatoren sinnvoll platzieren

Die obige Argumentation hilft allerdings nicht weiter, denn jetzt werden ja nur die Attribute der Backing Bean validiert und damit auch wieder nur die Eingabe über das Facelet.

Sinnvoller ist es, die Attribute der Datenklassen entsprechend zu annotieren. Dann erfolgt die Überprüfung immer, wenn die Daten des entsprechenden Objektes geändert werden sollen.

Wenn ihr das auch sinnvoll findet, solltet ihr die Attribute der Datenklasse `Exam` entsprechend annotieren. Damit dann die Validierung auch zentral stattfindet, verwendet ihr in der Backing Bean statt der einzelnen Attribute direkt nur ein Attribut als Objekt vom Typ `Exam`:

```
private Exam exam;
```

Das Objekt erzeugt ihr in einer `@PostConstruct`-Methode, weist es dem Attribut zu und setzt auch schon mal dessen Attribut `examiner`. Wie das geht, wisst ihr alles schon.

Im Facelet werden dann die Attribute des `Exam`-Objektes direkt referenziert, z. B.:

```
<p:inputText id="name" value="#{createExamController.exam.name}"/>
```

Deployt die Anwendung und probiert aus, ob fehlerhafte Eingaben wie gewünscht verhindert werden. Ihr werdet feststellen, dass das für Datum und Uhrzeit und den Namen der Prüfung wie erwartet funktioniert, aber für die Auswahl des Prüflings nur, wenn kein Prüfling ausgewählt wird.

Wird ein Prüfling ausgewählt, kommt es zu einem Konvertierungsfehler, der als Nachricht direkt angezeigt wird. Es können also jetzt gar keine Prüfungen mehr erzeugt werden. Na toll.

[21] `https://jakarta.ee/specifications/bean-validation/3.0/`

[22] `https://jakarta.ee/specifications/bean-validation/3.0/jakarta-bean-validation-spec-3.0.html#builtinconstraints`

Woher kommt der Konvertierungsfehler? Aus der Tatsache, dass zwischen Facelet und Controller Beans nur mit Zeichenketten gearbeitet wird. Das Facelet trägt also in das Attribut nicht etwa das ausgewählte `Student`-Objekt selbst ein, sondern die *toString()*-Repräsentation des Objektes. Tatsächlich weiß das Facelet überhaupt nichts von dem Objekt, denn es wird durch unsere EL-Ausdrücke in Zeichenketten umgewandelt (`itemValue="#{student}"}`) und nur damit arbeiten die Facelets bei der Ausführung.

Wieso funktioniert das aber für das `LocalDateTime`-Objekt für Datum und Uhrzeit der Prüfung? Das liegt daran, dass Faces für eine Reihe von Klassen vorgefertigte *Konverter* bereitstellt, die in entsprechenden Fällen automatisch zum Einsatz kommen. Die Spezifikation enthält eine Liste[23] mit Konvertern für Typen, die automatisch eingesetzt werden, wenn zwischen Facelet und Backing Bean kommuniziert wird.

Eine offensichtliche Lösung des Problems wäre der Rückbau auf ein Attribut für die Id des Prüflings in der Controller Bean. Dann muss das Attribut zusätzlich durch Faces validiert werden und vor dem Speichern – wie oben – aus der Id das eigentliche Objekt ermittelt und in die Prüfung eingetragen werden.

Eine Alternative besteht in der Verwendung eines eigenen Konverters[24]. Wenn ihr das ausprobieren möchtet, legt ein Unterpaket `converter` an und erzeugt darin eine Klasse, die das Interface `jakarta.faces.convert.Converter<Student>` implementiert. Implementiert dann darin die beiden geforderten Methoden *getAsObject()* und *getAsString()*. Das könnt ihr so gestalten, dass ihr mit der Id des Objektes als String arbeitet.

Annotiert die Klasse wie folgt:

```
@FacesConverter(forClass=Student.class, managed = true)
```

Über das Attribut `forClass` legt ihr fest, für welche Klasse der Konverter automatisch eingesetzt werden soll und über das Attribut `managed` sorgt ihr dafür, dass CDI-Injektionen möglich werden. Der Default-Wert ist `false`, was bedeutet, dass `@Inject` nicht funktioniert.

Wenn ihr den Konverter derart registriert habt, dann wird zwischen den Facelets und den Backing Beans an passenden Stellen mit den Objekten selbst gearbeitet. Daher müsst ihr das Attribut `itemValue` der Listeneinträge der Listenauswahlbox im Facelet `create_exam.xhtml` von `student.id` auf `student` ändern, da es sonst durch die Einmischung des Konverters zu Fehlern kommt.

Nach einem Redeployment sollte jetzt eine gültige Prüfung angelegt werden können.

Versucht jetzt, in der Startup Bean Prüfungen mit unsinnigen Werten anzulegen. Wird das – wie oben versprochen – verhindert?

Aufgabe 4.5 Prüfungsdarstellung

Wie oben bereits erwähnt ist die Darstellung von Prüfer:innen und Prüflingen in der Datentabelle nicht sehr sinnvoll. Sorgt für eine sinnvolle Darstellung in den Spalten, z. B.

```
Nachname, Vorname (E-Mail-Adresse)
```

[23] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0.html#standard-converter-implementations`

[24] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0.html#a1258`

Wenn ihr das für sinnvoll haltet, könnt ihr einfach die *toString()*-Methode der betroffenen Klassen entsprechend überschreiben. Wir geben allerdings zu bedenken, dass es bei einer expliziten Trennung von Model und View in der Architektur eher seltsam ist, dass das Model-Objekt festlegt, wie es dargestellt wird. Das muss dieser Lösung aber nicht widersprechen, denn es kann ja sein, dass die *toString()*-Repräsentation zufällig passt.

Alternativ könnt ihr das wie oben mit EL-Ausdrücken machen.

Eine weitere Alternative besteht in einer Methode in der Backing Bean, die ein Objekt als Argument erhält und eine Zeichenkette für genau das Facelet zurückgibt, in dem das Objekt dargestellt werden soll. Da die Backing Bean stark mit dem Facelet gekoppelt ist, wäre das hier konzeptionell passend.