Software-Projekt WiSe 23/24

Aufgabenblatt 2

Abgabe bis: 03.12.2023 23:59 Uhr MEZ

Autoren: Karsten Hölscher, Marcel Steinbeck

Während es in dem letzten Aufgabenblatt primär darum ging, die Entwicklungsumgebung einzurichten, das Projekt aufzusetzen und euch mit einigen grundlegenden Konzepten von Java EE vertraut zu machen, soll es in diesem Blatt darum gehen, die ersten Komponenten der Webanwendung zu implementieren.

Hierbei werden wir auf eine sogenannte Schichtenarchitektur zurückgreifen, die sich in der Praxis bewährt hat, da sie klare Schnittstellen definiert, sodass Aufgaben gut aufgeteilt werden können. In diesem Zusammenhang werden wir einen ersten vertikalen Durchstich durch die gesamte Schichtenarchitektur wagen, sodass ihr einen Eindruck davon bekommt, wie die einzelnen Komponenten zusammenarbeiten. Weiterhin werden wir uns mit der Bibliothek *Lombok* beschäftigen, die uns die Arbeit insofern erleichtern wird, als dass sie uns das lästige Schreiben von *Boilerplate Code* abnimmt.

Wir erweitern in diesem Aufgabenblatt unsere Anwendung aus dem ersten Aufgabenblatt um Funktionalität, die es ermöglicht, alle im Datenbestand vorhandenen Prüflinge anzuzeigen und bestehende Nutzer im System anzumelden und anzuzeigen.

Aufgabe 1 Personen

Wir werden in der späteren Webanwendung mindestens zwei Arten von Personen verwalten. Da sind einerseits die Prüflinge, die sich für eine Prüfung anmelden. Andererseits gibt es aber auch die eigentlichen Nutzer:innen der Webapplikation, die z. B. Prüfungstermine anlegen und verwalten. Diese zwei Personengruppen haben offensichtlich Gemeinsamkeiten, nämlich mindestens einen Vor- und Nachnamen, sowie eine E-Mail-Adresse. Es ist also sinnvoll, eine abstrakte Basisklasse `Person` zu implementieren.

Aufgabe 1.1 Abstrakte Leute

Eine Person ist technisch ein Teil des Datenmodells unserer Applikation. Eine solche Klasse sollte in einem eigenen Paket abgelegt werden. Erstellt also ein entsprechendes Unterpaket mit dem Namen *model* und darin die besagte abstrakte Klasse `Person`. Wenn ihr im vorherigen Blatt nicht von den Vorgaben abgewichen seid, dann würdet ihr dieses Unterpaket also im Paket `de.unibremen.cs.swp.eksadat` anlegen. Allgemein gilt: Wenn nicht anders angegeben, dann beziehen wir uns im Folgenden bei dem Begriff *Unterpaket*, immer auf Pakete die unter dem Paket `de.unibremen.cs.swp.eksadat` liegen. Implementiert die oben angegebenen Attribute mit sinnvollen Datentypen und wie üblich dem Zugriffsmodifikator `private`. Ein expliziter Konstruktor ist nicht nötig – wir verwenden jetzt ersmtal den parameterlosen Standardkonstruktor, den der Compiler automatisch einfügt.

Eigentlich müssten wir jetzt ebenfalls Getter- und Setter-Methoden gemäß der üblichen Namenskonventionen selbst erstellen oder durch IntelliJ generieren lassen (Menüpunkt `Code` → `Generate`). Ebenso müssten wir noch die *equals()*-Methode überschreiben, sodass zwei Personen äquivalent sind, wenn ihre E-Mail-Adressen übereinstimmen. Für konsistentes

Verhalten müsste dann auch die *hashCode()*-Methode passend dazu überschrieben werden. Das alles lassen wir jetzt jedoch erstmal weg.

Aufgabe 1.2 Studis

Implementiert nun – ebenfalls im Paket `model` – eine konkrete Unterklasse von `Person` vom Typ `Student`. Eine Student:in benötigt offensichtlich mindestens ein Attribut für die Matrikelnummer. Legt es mit einem sinnvollen Datentyp Java-konform an und kümmert euch auch hier nicht um Getter- und Setter-Methoden.

Aufgabe 2 Lombok ist auch eine Insel

Eine sehr nützliche Bibliothek, die hier vorgestellt werden soll (*Projekt Lombok*[1], kurz *Lombok*), unterscheidet sich von anderen Bibliotheken insofern, als dass das erklärte Ziel nicht die Bereitstellung spezifischer Funktionalitäten ist, sondern die Erweiterung der Programmiersprache Java. Über verschiedene Annotationen (und entsprechende Codegeneratoren) soll *Boilerplate Code* (also repetitiver Programmcode, der immer und immer wieder erstellt werden muss) reduziert oder gar ganz vermieden werden. Das folgende Beispiel verdeutlicht dies:

```
1 @Getter
2 @Setter
3 public class SomeClass {
4
5     private String someText;
6
7     private int aNumber;
8     ...
```

Das lästige Erstellen von Getter-/Setter-Paaren (auch wenn die IDE dabei unterstützen kann) lässt sich mithilfe der Annotationen `@Getter` und `@Setter` einfach automatisieren. Werden die Annotationen auf Klassenebene angewendet (wie hier im Beispiel), werden automatisch für alle nichtstatischen Felder die gewünschten Methodenpaare (unter Einhaltung der Kodierrichtlinien für Getter- und Setter-Methoden) generiert.

Die Prüfung der Parameterwerte in den Setter-Methoden wird derart allerdings nicht mit generiert. Für Prüfungen auf `null` gibt es die Annotation `@NonNull`. Die wird an das zu prüfende Attribut geschrieben:

```
1 @NonNull
2 private String someText;
```

Leider löst Lombok hier eine unsinnige `NullPointerException`(NPE) aus, wenn der Setter-Methode der Wert `null` übergeben wird. Das ist hier unsinnig, weil eine NPE ausgelöst werden soll, wenn eine `null`-Referenz *dereferenziert* wird. Das ist bei einer Zuweisung eines Argumentes an ein Attribut nicht der Fall. Glücklicherweise können wir die konkret ausgelöste Exception konfigurieren – dazu später mehr.

Lombok kann nicht nur Getter- und Setter-Methoden generieren, sondern bietet eine Vielzahl von Erweiterungen für unterschiedliche Zwecke an. Sehr nützlich sind z. B. auch die Annotatio-

[1] `https://projectlombok.org`

nen `@EqualsAndHashCode` (zur Generierung des Methodenpaares *equals()* und *hashCode()*) und `@ToString` (zur Generierung der Methode *toString()*).

Das obige Beispiel lässt sich wie folgt erweitern:

```
1  @Getter
2  @Setter
3  @EqualsAndHashCode
4  public class SomeClass {
5
6      @NonNull
7      private String someText;
8
9      private int aNumber;
10
11 }
```

Durch die Annotationen `@EqualsAndHashCode`[2] (4) werden Implementierungen für *equals()* und *hashCode()* generiert, die alle Attribute einbeziehen, die nicht statisch oder transient sind. Möchten wir nicht alle, sondern nur bestimmte Attribute einbeziehen, dann können diese aber auch explizit konfiguriert werden, indem wir die Annotation entsprechend konfigurieren:

```
1      @EqualsAndHashCode(onlyExplicitlyIncluded = true)
```

Dann müssen diejenigen Attribute, die für die Prüfung auf *equals* und die Generierung des Hashcodes einbezogen werden sollen, jeweils mit der Annotation `@EqualsAndHashCode.Include` versehen werden.

Die von Lombok generierte *equals()*-Methode vergleicht nicht nur Objekte des exakt gleichen Typs (über *getClass()*), sondern auch Subtypen (über `instanceof`). Das kann den Kontrakt bezüglich der Symmetrie von *equals* brechen. Eine sinnvolle Implementierung von *equals()* ist gar nicht so leicht (siehe `https://www.artima.com/articles/how-to-write-an-equality-method-in-java`). Lombok stellt allerdings die im Artikel vorgeschlagene Lösung über `canEqual` zur Verfügung.[2]

Die Annotation `@AllArgsConstructor`[3] generiert einen Konstruktor, der Parameter für jedes Attribut der Klasse erhält. Die Annotation `@NonNull` wird dabei berücksichtigt. Es gibt auch noch eine Annotation `@RequiredArgsConstructor` – näheres dazu in der Dokumentation.[3] Kopierkonstruktoren werden von Lombok leider nicht unterstützt.

Eine Liste der als stabil gekennzeichneten Features findet ihr unter `https://projectlombok.org/features/all`. Darüber hinaus gibt es noch eine Liste mit experimentellen Features: `https://projectlombok.org/features/experimental/all`. Diese sollten nur mit Bedacht eingesetzt werden, da das Entwickler:innenteam von Lombok sich vorbehält, diese Features jederzeit zu ändern bzw. ersatzlos zu streichen.

[2] `https://projectlombok.org/features/EqualsAndHashCode`
[3] `https://projectlombok.org/features/constructor`

Die von Lombok generierten Getter- und Setter-Methoden können weder flache, noch tiefe Kopien von Objekten erstellen. In vielen Situationen soll jedoch der interne Zustand einer Klasse durch die simple Herausgabe bzw. Übernahme von Objekten nicht exponiert werden. Hier gibt es verschiedene Lösungen: i) Die notwendige Logik zum Erstellen von Kopien wird selbst implementiert, ii) es wird eine Bibliothek für die Erstellung von Kopien eingebunden oder iii) die zu kopierenden Objektstrukturen werden serialisiert (Java-Serialisierung, JSON/JSOG, CSV etc.) und deserialisiert – was prinzipbedingt eine tiefe Kopie erstellt. Falls in iii) sehr große Objektstrukturen kopiert werden müssen, könnte ein Blick auf die Bibliothek *FST* (siehe `https://github.com/RuedigerMoeller/fast-serialization`) lohnenswert sein.

Um Lombok in euer Projekt einzubinden, sind mehrere Schritte notwendig. Der erste Schritt sollte wenig überraschend sein: Bindet Lombok wie unter `https://projectlombok.org/setup/maven` beschrieben in euer Maven-Projekt (denkt allerdings an die Konvention, dass wir die konkrete Versionsnummer einer Abhängigkeit über eine Variable konfigurieren) ein. Zusätzlich müsst ihr ggf. in eurer IDE ein Plugin installieren, damit der von Lombok generierte Code auch von der IDE „verstanden“ wird. Eine Anleitung findet ihr unter `https://www.baeldung.com/lombok-ide` (diese URL wird auf der offiziellen Seite von Lombok gelistet).

Habt ihr Lombok erfolgreich eingerichtet, solltet ihr nun zur Übung die Klassen im Paket `model` so umbauen, dass entsprechende Lombok-Annotationen zum Einsatz kommen.

Um die Überprüfung von Parametern auf `null` innerhalb von normalen (nicht Getter-/Setter-) Methoden von Lombok erzeugen zu lassen, könnt ihr übrigens direkt vor die zu prüfenden Parameter die Annotation `@NonNull` schreiben.

Wie oben erläutert löst Lombok eine NPE aus, wenn ein Argument mit dem Wert `null` überprüft wird. Um die Exception in diesem Fall umzustellen, könnt ihr eine Datei `lombok.config`[4] erzeugen und in das Verzeichnis `src/main/java` platzieren. In der Datei befindet sich dann die Zeile:

```
1 lombok.nonNull.exceptionType = IllegalArgumentException
```

Konfiguriert euer Projekt entsprechend.

Zum Abschluss dann noch ein Hinweis auf eine Annotation, die im Kontext von Logging nützlich sein könnte: `@Slf4j`.[5] Schaut euch an, wie die Annotation funktioniert und verwendet wird und passt das Logging in der Klasse `HelloController` entsprechend an.

Die automatische Generierung von getter- und setter-Methoden, Konstruktoren und Methoden wie *equals()* und *hashCode()* usw. werden seit Java 16 auch von der `Record`-Klasse unterstützt.[a] Diese Records sind allerdings noch nicht mit JPA kompatibel und da wir das bald einsetzen möchten, verzichten wir hier auf eine explizite Einführung.

[a] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html`

[4] mehr zur Konfiguration unter `https://projectlombok.org/features/configuration`
[5] `https://projectlombok.org/features/log`

Aufgabe 3 Schichtenarchitektur anlegen

Für eine monolithische Webanwendung, die ihr in einem Team von bis zu 6 Personen entwickeln sollt, hat sich eine Schichtenarchitektur mit klaren Zuständigkeiten pro Schicht bewährt. Es gibt dabei die folgenden Schichten:

User Interface Das User-Interface beinhaltet die Facelets und die Controller-Klassen und hat die Aufgabe, mit den Nutzer:innen zu interagieren (Eingaben entgegennehmen und Ausgaben erzeugen).

Service Die Service-Schicht realisiert die Applikationslogik, in der die den Anwendungsfällen zugrundeliegenden Aktionen ausgeführt werden.

Persistenz Die Persistenz-Schicht kümmert sich um das Persistieren (Speichern und Laden) der Datenklassen.

Model Die Datenklassen selbst.

Die einzelnen Schichten haben also klar geregelte Zuständigkeiten und werden auch als *Komponenten* bezeichnet. In einer Schichtenarchitektur liegen die einzelnen Schichten übereinander und eine Schicht kann nur Funktionalität von Modulen (Klassen) derselben Schicht oder der direkt darunterliegenden Schicht nutzen. Es ist nicht erlaubt (und auch nicht sinnvoll) eine Schicht zu überspringen – so sehr das manchmal auch reizen mag.

Abbildung 1 zeigt die Schichtenarchitektur unserer Webanwendung in Form eines Komponentendiagramms.

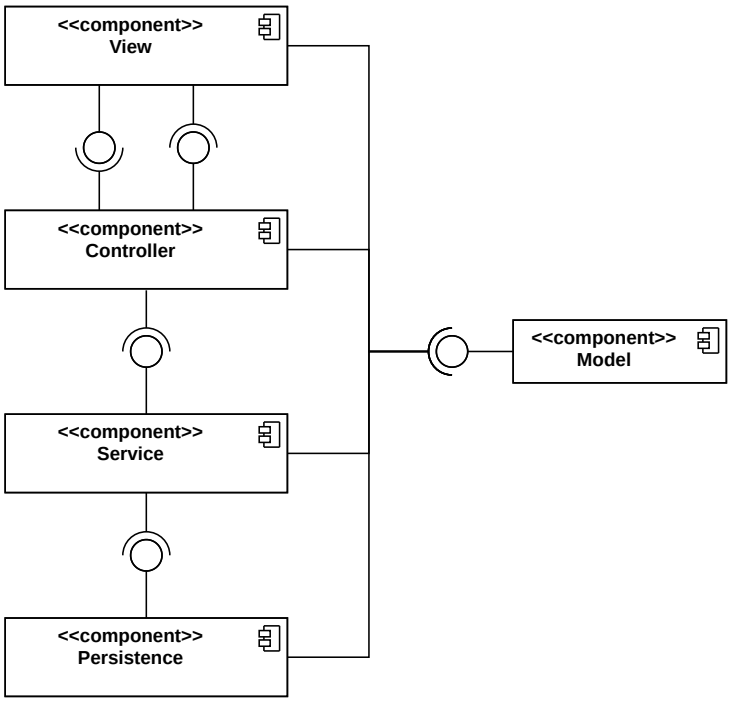

Abbildung 1: empfohlene Schichtenarchitektur

Der Lollipop (Kreis mit anhängender Kante) bedeutet, dass die Komponente eine Schnittstelle *anbietet*, während die Zange bedeutet, dass die Komponente eine Schnittstelle *verwendet*.

Eine Komponente besteht aus *Modulen*, die jeweils für eine Aufgabe innerhalb der Komponente zuständig sind. Ein Modul in der Komponente `Service` könnte z. B. für die Verwaltung von Prüfungsterminen zuständig sein, während ein anderes Modul für die Verwaltung von Prüflingen zuständig ist. In großen Projekten können Module aus mehreren Klassen bestehen, aber in kleinen Projekten – wie bei unseren Übungen – besteht ein Modul typischerweise aus genau einer Klasse.

In einem Java-Backend (alles außer `View` im Diagramm in Abbildung 1) empfiehlt es sich, die Komponenten durch Java-Pakete zu realisieren. Da sich Module einer Komponente gegenseitig beliebig verwenden dürfen, kann dann der Zugriffsmodifikator `package` sinnvoll eingesetzt werden.

Die angebotene Schnittstelle einer Komponente ergibt sich aus den öffentlichen Bestandteilen aller Klassen, die der Komponente zugeordnet sind. Das sind dann insbesondere alle `public` Methoden der Klassen eines Paketes.

Wenn wir das Diagramm betrachten, fällt zunächst auf, dass die Schichtungsregeln bei den Komponenten `View` und `Controller` nicht eingehalten werden. Hier erfolgt der Zugriff bzw. die Kommunikation nicht in einer Richtung, sondern bidirektional. Das ist aber nicht weiter problematisch, da `Controller` und `View` sehr eng miteinander verknüpft sind und sich die Aufgabenbereiche teilweise überschneiden.

Eine weitere Auffälligkeit besteht in der Tatsache, dass alle Komponenten die Klassen aus der Komponente `Model` kennen und verwenden. Auch das ist konzeptionell kein Problem, da Datenklassen eine Sonderrolle in einem derartigen Projekt einnehmen. Einerseits tragen sie die zu verarbeitenden Daten – und diese werden ja gerade in den anderen Modulen ausgelesen oder verändert. Andererseits enthalten sie gar keine Logik, sodass ihre Schnittstellen sowieso nur aus Getter- und Setter-Methoden (plus *equals*, *hashCode()*, *toString()*) bestehen, die letztlich ja nur Datenzugriff ermöglichen.

Der große Vorteil einer solchen Architektur besteht darin, dass hervorragend parallel implementiert werden kann. Änderungen in einer Komponente betreffen dann nur die direkt darüber liegende Komponente – außer für die Datenklassen der Komponente `Model`. Das ist dann auch ein wichtiger Grund dafür das Datenmodell möglichst früh zu erstellen und zu implementieren. Durch die geringe Kopplung der Klassen (z. B. ist den Klassen in der Komponente `Persistence` jede Änderung in den darüberliegenden Komponenten völlig egal) verringern sich auch Konflikte in der Versionskontrolle.

Prüflinge anzeigen

Wir wollen jetzt zunächst ein recht simples Feature implementieren – nämlich die Anzeige aller im Datenbestand vorhandenen Prüflinge. Im Datenmodell existiert bereits eine Klasse `Student`, sodass wir uns auf die anderen Komponenten konzentrieren können. Wir arbeiten jetzt *bottom-up*, d. h. wir beginnen bei der untersten Schicht im obigen Diagramm: `Persistence`.

Aufgabe 3.1 Prüflinge im Datenbestand

Da wir noch keine Datenbank verwenden, gibt es aktuell auch keinen Datenbestand. Wir müssen diesen also erst herstellen.

Aufgabe 3.1.1 Komfortkonstruktor für Prüflinge

Um die Erstellung von `Student`-Objekten ohne Datenbank-Anbindung zu erleichtern, solltet ihr der Klasse `Person` die Lombok-Annotation `RequiredArgsConstructor` hinzufügen. Zusätzlich

solltet ihr auch die `NoArgsConstructor` annotieren, damit auch der parameterlose Standardkonstruktor generiert wird. Leider kann Lombok in der Unterklasse `Student` den entsprechenden Konstruktor mit allen benötigen Argumenten nicht automatisch erzeugen (da zum Zeitpunkt der Code-Generierung durch Lombok die Klassenhierarchie nicht zur Verfügung steht und damit die Oberklasse nicht analysiert werden kann), sodass ihr ihn dort selbst erstellen müsst[6]. Auch hier solltet ihr allerdings `NoArgsConstructor` annotieren für die Generierung des parameterlosen Standardkonstruktors.

Aufgabe 3.1.2 Ein Repository für Prüflinge

Für die Persistenz, d. h. die Verwaltung der zu speichernden Daten, empfehlen wir den Einsatz des *Repository Musters*[7].

Das Hauptziel des Repository-Musters besteht darin, die Datenzugriffslogik von anderen Teilen der Anwendung zu trennen und somit die Wartbarkeit, Testbarkeit und Flexibilität der Software zu verbessern. Die Hauptvorteile sind:

Trennung von Verantwortlichkeiten: Durch die Verwendung des Repository-Musters kann die Geschäftslogik der Anwendung von den Details der Datenbankkommunikation getrennt werden. Dadurch wird der Code übersichtlicher und einfacher zu warten.

Wiederverwendbarkeit: Das Muster ermöglicht es, Datenzugriffsoperationen in einem zentralen Repository zu kapseln. Dadurch können diese Operationen in verschiedenen Teilen der Anwendung wiederverwendet werden, ohne den Datenbankcode an vielen Stellen duplizieren zu müssen.

Testbarkeit: Repositories können durch Mocking oder Stubbing ersetzt werden, um isolierte Komponententests zu erleichtern. Dadurch kann z. B. die Geschäftslogik getestet werden, ohne tatsächlich auf eine Datenbank zugreifen zu müssen.

Datenbankunabhängigkeit: Indem Datenbankzugriffscodes in Repositories gekapselt werden, kann sich die Datenbanktechnologie hinter dem Repository ändern, ohne die übrige Anwendungslogik ändern zu müssen. Das erleichtert den Wechsel von einer Datenbank zu einer anderen.

Zentrale Verwaltung von Abfragen: Komplexe Abfragen und Datenzugriffslogik können im Repository organisiert werden, anstatt in verschiedenen Teilen der Anwendung verstreut zu sein. Dies verbessert die Wartbarkeit und erleichtert die Anpassung von Abfragen.

Üblicherweise wird das Muster so eingesetzt, dass es pro Datenklasse ein entsprechendes Repository gibt. Wir erstellen also im Unterpaket `persistence` ein Repository für die Datenklasse `Student`:

[6] Hinweis: Eine negative Matrikelnummer ist nicht sinnvoll.
[7] `https://martinfowler.com/eaaCatalog/repository.html`

```
10  @Named
11  @ApplicationScoped
12  public class StudentRepository implements Serializable {
13
14    private final List<Student> students = new LinkedList<>();
15
16    public void save(final Student student) {
17      students.add(student);
18    }
19
20    public List<Student> findAll() {
21      return students;
22    }
23
24  }
```

Da wir noch keine Datenbank einsetzen, speichert unser Repository temporär alle `Student`-Objekte in einem entsprechenden Attribut. Über die *save()*-Methode wird das übergebene Objekt persistiert (aktuell nur bis zum Neustart der Webanwendung – daher wird der Application-Scope gewählt) und mittels *findAll()* wird eine Liste mit allen Objekten zurückgegeben.

Aufgabe 3.1.3 Datenbestand beim Start erzeugen

Wir wollen unseren Datenbestand – solange wir keine Datenbank verwenden – temporär möglichst früh erzeugen. Daher benötigen wir eine Art „Startup"-Bean mit einer Methode, die beim Starten der Webanwendung ausgeführt wird. Es ist später ohnehin nützlich, eine solche Klasse zu haben um dort ggf. Dinge zu konfigurieren.

Legt im Paket `de.unibremen.cs.swp.eksadat` eine Klasse `EksadatConfig` an mit folgenden Annotationen (deren Bedeutung wir bereits kennen):

```
14  @Named
15  @ApplicationScoped
16  @Slf4j
17  public class EksadatConfig {
```

Fügt die folgende Methode hinzu:

```
19  public void contextInitialized(
20          @Observes @Initialized(ApplicationScoped.class) ServletContext context) {
21      log.trace("Context with path {} initialized.", context.getContextPath());
22      initApp();
23  }
```

Die Annotationen vor dem Parameter `context` bedeuten, dass die Methode die Initialisierung des Application-Scopes beobachtet und somit beim Starten des Scopes aufgerufen wird. Logischerweise wird der Application-Scope direkt beim Starten der Webanwendung initialisiert, sodass wir damit den gewünschten Effekt einer Startup-Methode erzielen.

Fügt jetzt die Methode *initApp()* hinzu, die nur ihre Ausführung protokolliert und dann die Methode zur Erzeugung von `Student`-Objekten aufruft:

```
23 private void initApp() {
24     log.trace("initApp executing.");
25     createStudents();
26 }
```

Um den Inhalt dieser Methode kümmern wir uns später. Wenn euch stört, dass das Projekt in diesem Zustand nicht übersetzbar ist, könnt ihr die Zeile mit dem Aufruf der Methode *createStudents()* vorerst auskommentieren.

Die Unterteilung in mehrere Methoden muss natürlich nicht so gemacht werden, ist aber der Übersichtlichkeit und damit der Verständlichkeit und folglich der Wartbarkeit zuträglich.

Aufgabe 3.2 Ein Service für Prüflinge

Für die `Service`-Komponente empfehlen wir eine analoge Strukturierung wie für die Repositories. Für jede Datenklasse sollte es eine entsprechende Service-Klasse geben, die die gesamte Logik für ihre Datenklasse kapselt und bereitstellt. Für unser Feature ist das jetzt eher langweilig, da wir in der entsprechenden Service-Methode einfach die passende Repository-Methode aufrufen und deren Rückgabe – sofern vorhanden – weiterreichen.

Erstellt also im Unterpaket `service` die Klasse `StudentService` mit folgendem Inhalt:

```
10  public class StudentService {
11
12    @Inject
13    private StudentRepository studentRepository;
14
15    public List<Student> findAll() {
16      return studentRepository.findAll();
17    }
18
19    public void save(final Student student) {
20      studentRepository.save(student);
21    }
22
23  }
```

Da das Repository mit `@Named` annotiert ist, können wir es uns in unserer Service-Klasse einfach injizieren lassen. Wir müssen uns also wie bei den Backing Beans der Facelets nicht um die Objekterzeugung und Verwaltung kümmern.

Wir lassen uns also in Zeile 12 das Repository von CDI injizieren. Die Service-Methoden *findAll()* und *save()* haben keine eigene Funktionalität, sondern rufen direkt die entsprechenden Repository-Methoden auf und leiten eine Rückgabe an einen Aufrufer weiter. Warum wir hier keinen Scope festlegen, erläutern wir unten.

Hier sieht es jetzt so aus, als würden wir durch die Service-Komponente nur eine weitere Indirektion in unsere Architektur einbauen. Dem ist aber nicht so, wie wir später noch sehen werden. Es gibt nämlich in der Geschäftslogik durchaus auch kompliziertere Dinge, die dann in einer Service-Methode kombiniert und atomar (als *Transaktion*) ausgeführt werden müssen (z. B. das gleichzeitige Anlegen eines Prüflings mit seinem Prüfungstermin). Durch die Kapselung der Geschäftslogik in einem Service wird es auch möglich, Funktionalität nicht nur durch eine Webseite, sondern z. B. auch durch eine REST-Schnittstelle[a] anzubieten. Die Komponente, die dann für REST zuständig ist, kann dann direkt die Service-Klassen verwenden.

[a] `https://www.codecademy.com/article/what-is-rest`

Bisher mussten wir alle Klassen manuell über die Annotation `@Named` als CDI-Beans registrieren und müssten das eigentlich auch bei dieser Klasse tun. Um uns das jetzt und in Zukunft zu ersparen, ändern wir in der Datei `beans.xml` das Attribut `bean-discovery-mode` von `"annotated"` auf `"all"`. Damit werden fast[8] Java-Klassen in unserem Projekt von CDI gefunden. Ihr könnt dann auch die Annotation `@Named` an den Klassen `StudentRepository` und `EksadatConfig` entfernen. Denkt daran, auch die Import-Anweisungen für die Annotation zu entfernen, denn wir möchten natürlich aufgeräumten Quellcode haben. IntelliJ kann die Import-Anweisungen automatisch für euch aufräumen und organisieren über den Menüpunkt `Code→Optimize Imports` bzw. dessen konfigurierte Tastenkombination.

Wir ergänzen nun unsere Startup-Bean `EksadatConfig` um ein Attribut, dessen Wert wir uns von CDI injizieren lassen:

```
10 @Inject
11 private StudentService studentService;
```

Die Methode *createStudis()* können wir jetzt so implementieren, dass sie die erstellten Objekte direkt zum Speichern an den Service übergeben werden:

```
10 private void createStudents() {
11     log.trace("Creating students.");
12     studentService.save(new Student(
13                                 "Anton", "Aalglatt", "anton@offline.de", 9901));
14     studentService.save(new Student(
15                                 "Berta", "Belanglos", "bertan@offline.de", 9902));
16     studentService.save(new Student(
17                                 "Mona", "Clickclocken", "mona@offline.de", 9903));
18 }
```

Solltet ihr den Aufruf der Methode *createStudents()* zuvor auskommentiert haben, könnt ihr ihn jetzt wieder „einkommentieren“.

Aufgabe 3.3 Prüflinge in der Backing Bean

Wenn wir die Liste aller Prüflinge in einem Facelet anzeigen möchten, dann benötigen wir eine Backing Bean mit einer Methode, die diese Liste bereitstellt.

[8] `https://jakarta.ee/specifications/cdi/4.0/jakarta-cdi-spec-4.0.html#what_classes_are_beans`

Erzeugt also im Unterpaket `controller` die Backing Bean `StudentsController` mit folgendem Inhalt:

```
10 @Named
11 @RequestScoped
12 public class StudentsController {
13
14     @Inject
15     private StudentService studentService;
16
17     public List<Student> getStudents() {
18         return studentService.findAll();
19     }
20
21 }
```

Wir registrieren auch diese Klasse über `@Named` bei Faces und geben ihr den Request-Scope. Die Service-Klasse, die uns letztlich alle Prüflinge aus dem Datenbestand besorgt, lassen wir von CDI injizieren. In der eigentlichen Getter-Methode wird wieder einfach die Rückgabe der Service-Methode direkt weitergereicht.

Die Backing Beans der Facelets benötigen die Annotation `@Named` auch dann, wenn wir in der `beans.xml` den Attributwert von `bean-discovery-mode` auf `"all"` geändert haben. Das liegt daran, dass Faces einen anderen Mechanismus verwendet, der zwar mit CDI zusammenarbeitet aber nicht ersetzt.

Wir haben oben für die Service-Klasse `StudentService` keinen expliziten Scope konfiguriert. Dadurch erhält sie den Scope *Dependent*, was bedeutet, dass sie den gleichen Scope erhält, wie die Bean, in die sie injiziert wird. Da die Controller-Bean den Request-Scope hat, erhält die Service-Bean in diesem Fall also ebenfalls den Request-Scope.

Generell lautet die Empfehlung von erfahrenen Faces-Entwickler:innen, immer zunächst den „kleinsten“ Scope für eine Backing Bean zu verwenden (der kleinste Scope ist dabei derjenige mit der kürzesten Lebensdauer). Sollte die Seite sich dann nicht wie gewünscht verhalten, wird der Scope schrittweise „erhöht“, d. h. der Scope mit der nächst längeren Lebensdauer wird probiert).

Aufgabe 3.4 Prüflinge im Facelet anzeigen

Wir benötigen jetzt noch ein Facelet, das zunächst nur die Aufgabe hat, alle Prüflinge anzuzeigen.

Legt dazu im Verzeichnis `src/main/webapp` ein Facelet namens `students.xhtml` an und verseht es analog zum `hello.xhtml`-Facelet mit dem identischen `DOCTYPE` und den `html`-, `head` und `body`-Zweigen. Wählt einen geeigneten Titel für die Seite und schon mal eine `h1`-Überschrift, die über der Liste der Prüflinge dargestellt wird.

Fügt dann in den `body`-Zweig unterhalb eurer `h1`-Überschrift folgendes ein:

```
13 <h:dataTable value="#{studentsController.students}" var="student">
14
15     <h:column>
16         ${student.lastName}
17     </h:column>
18
19     <h:column>
20         ${student.firstName}
21     </h:column>
22
23     <h:column>
24         ${student.email}
25     </h:column>
26
27     <h:column>
28         ${student.matriculationNo}
29     </h:column>
30
31 </h:dataTable>
```

Sollten eure Attribute bzw. die Getter-Methoden anders benannt sein (z. B. nur `name` statt `lastName`) müsst ihr die EL-Ausdrücke entsprechend anpassen (z. B. `${student.name}` statt `${student.lastName}`.

Die Idee einer solchen Datentabelle ist, dass es pro Eintrag einer Liste eine Zeile in der Tabelle gibt. Die Liste selbst wird im Attribut `value` festgelegt. Wir erinnern uns: Dort wird die Methode *getStudents()* eines Objektes der Controller-Klasse aufgerufen. Im nächsten Schritt werden die einzelnen Spalten konfiguriert. Dazu ist es nötig, Zugriff auf das jeweils passende Objekt der Liste zu haben. Daher wird im Attribut `var` der Name der Objekt-Variablen festgelegt. Das `dataTable`-Widget verhält sich hier also vergleichbar zu einer `for-each`-Schleife, bei der die Sammlung, die sich aus dem Methodenaufruf `studentsController.getStudents()` ergibt, von vorne nach hinten durchlaufen wird, wobei das jeweils aktuelle Objekt in der Variablen `student` referenziert wird.

Für jede Spalte, die wir in unserer Datentabelle haben möchten, spezifizieren wir ein `h:column`-Element. Zwischen öffnendem und schließendem Tag wird der Text angegeben, der in der Spalte angezeigt werden soll. Hier kommen wieder EL-Ausdrücke zum Einsatz, die die entsprechenden Getter-Methoden auf dem `Student`-Objekt aufrufen. Dadurch werden deren Rückgabewerte in den Spalten angezeigt. Eine Besonderheit ist hier die Syntax, denn die EL-Ausdrücke werden hier nicht mit einer Raute (`#`), sondern mit einem Dollarsymbol (`$`) eingeleitet. Das liegt daran, dass wir die EL-Ausdrücke hier nicht als Attributwerte verwenden (wie im `hello`-Facelet), sondern direkt die Rückgabe in unsere Webseite einbinden. In Zeile 13 verwenden wir die Raute (`#`), da der EL-Ausdruck dort als Attributwert für das Element `h:dataTable` verwendet wird.

Der Hauptunterschied zwischen EL-Ausdrücken, die mit `$` und solchen, die mit `#` beginnen, liegt in ihren Auswertungskontexten.

EL-Ausdruck, der mit `$` beginnt (`${`*`Ausdruck`*`}`): Dieser Typ von EL-Ausdruck wird sofort ausgewertet, wenn der Komponentenbaum aufgebaut wird. Der Komponentenbaum (*Component Tree*) ist ein zentrales Konzept in Faces. Er repräsentiert die Hierarchie der UI-Komponenten (wie Textfelder, Buttons, Tabellen usw.) innerhalb einer Faces-Seite und bildet die Struktur der Benutzeroberfläche ab. Der Komponentenbaum ermöglicht es Faces, den Zustand der UI über mehrere Anfragen hinweg zu verwalten und zu synchronisieren. Dieser Ansatz unterstützt die Entwicklung komplexer, zustandsbasierter Anwendungen, ohne dass Entwickler:innen den Zustand manuell verwalten müssen. Der Komponentenbaum ermöglicht auch die Wiederverwendung von Komponenten, die in verschiedenen Teilen der Anwendung verwendet werden können.

Wenn Daten direkt in der Ansicht abgerufen und angezeigt werden, sollte dieser EL-Ausdruck verwendet werden.

EL-Ausdruck, der mit `#` beginnt (`#{`*`Ausdruck`*`}`): Dieser Typ von EL-Ausdruck wird für eine verzögerte (*deferred evaluation*)Auswertung verwendet. Er wird erst ausgewertet, wenn der Wert eines Attributs tatsächlich abgerufen wird. Im `hello`-Facelet wird im `action`-Attribut der Schaltfläche angegeben, welche Methode auf welchem Objekt beim Betätigen der Schaltfläche aufgerufen wird. Wenn die Webseite angezeigt wird, wird dieser Methodenaufruf gar nicht benötigt. Er muss erst durchgeführt werden, wenn die Schaltfläche betätigt wird (und auch erst dann muss der entsorechende EL-Ausdruck ausgewertet werden).

In JSP (*Java Server Pages*) – eine Alternative zur Erstellung dynamischer Webseiten mit Java und in diesem Sinne gewissermaßen ein Vorgänger von Faces – dürfen EL-Ausdrücke dieses Typs nur bei Attributen angewendet werden (und auch nur bei solchen, die eine verzögerte Auswertung erlauben[a]). In Faces ist das weniger streng geregelt. Dennoch wendet IntelliJ in seinen Überprüfungen der Faceletes die gleiche Regel an wie für JSP, sodass dieser Typ eines EL-Ausdrucks in Attributen von Faces-Tags verwenden werden sollte. Nur dann funktioniert übrigens die Navigation über CTRL-Klick aus dem Facelet in die Controller Bean reibungslos.

[a] `https://jakarta.eaworld.io/specifications/expression-language/5.0/jakarta-expression-language-spec-5.0.html#syntax-restrictions`

Probiert das Ergebnis eurer Änderungen aus, indem ihr die URL `http://localhost:8080/eksadat/students.xhtml` im Browser aufruft. Ihr könnt auch mal das Attribut `border` der Datentabelle auf den Wert "`1`" setzen und euch anschauen, was passiert.

Es ergibt wenig Sinn, sich mit dem `h:dataTable`-Widget jetzt intensiver zu beschäftigen, da wir bald eine Bibliothek kennenlernen werden, die sehr viel mächtigere Widgets zur Verfügung stellt (die überdies auch optisch ansprechender gestaltet sind).

Aufgabe 4 Nutzer:innen und ihre Rollen

Wie oben erwähnt, soll es in unserer Webanwendung auch Nutzer:innen geben, die die Prüfungstermine und auch die Prüflinge verwalten. Diese Nutzer:innen haben die Eigenschaften von Personen, aber zusätzlich ein Passwort, mit dem sie sich am System anmelden können.

Aufgabe 4.1 Nutzer:innen

Erstellt also eine konkrete Unterklasse von `Person` mit dem Namen `User`, die eine Nutzer:in realisiert. Ihr einziges Attribut vom Typ `String` ist das Passwort[9].

Analog zu den Prüflingen soll es ein Facelet `users.xhtml` geben, das eine Liste aller Nutzer:innen in einer Datentabelle anzeigt. Erstellt das Facelet und dann die nötigen Klassen in den verschiedenen Schichten und legt beim Starten der Webanwendung zwei verschiedene Nutzer:innen im Datenbestand an.

Aufgabe 4.2 Rollen

Nutzer:innen verwenden unsere Webanwendung in unterschiedlichen *Rollen*. Je nach Rolle ändern sich die Aufgaben und auch die Berechtigungen. Der Einfachheit halber unterscheiden wir zunächst nur zwei Rollen: `ADMIN` und `USER`. Nutzer:innen mit der Rolle `ADMIN` sollen als Administrator:innen später mal Nutzer:innen verwalten (anlegen, ändern, löschen) können. In der aktuellen Fassung sollen Administrator:innen die Liste der Nutzer:innen sehen dürfen. Nutzer:innen mit der Rolle `USER` dürfen sich nur die Liste der Prüflinge ansehen.

Legt dazu eine Aufzählung (`enum`) `Role` im Unterpaket `model` an. Sie enthält die Konstanten `ADMIN` und `USER`.

Erweitert dann die Klasse `User` um ein Attribut für die Rollen der Nutzer:in. Da diese mehrere Rollen haben können, wird der Typ `Set<Role>` verwendet. Stellt sicher, dass es auch Getter- und Setter-Methoden für das Attribut gibt.

Ändert jetzt die Erzeugung der initialen Nutzer:innen in der Startup-Bean derart, dass eine von ihnen die Rolle `ADMIN`[10] und eine die Rolle `USER` bekommt.

Aufgabe 5 Bitte melde dich (an)

In dieser Aufgabe sollt ihr jetzt den Zugriff auf die Nutzer:innen- und Prüflingslisten einschränken auf am System angemeldete Nutzer:innen mit den entsprechenden Rollen (siehe oben). Wir führen das im Folgenden für die Rolle `ADMIN` durch.

Aufgabe 5.1 Login im Frontend

Tragt in die Datei `web.xml` zunächst die gewünschte Rolle ein:

```
<security-role>
    <role-name>ADMIN</role-name>
</security-role>
```

[9] Über die Passwort-Verschlüsselung machen wir uns später Gedanken

[10] Das geht ganz angenehm z. B. mit `new HashSet(Set.of(Role.ADMIN))`

Über ein URL-Muster kann jetzt ebenfalls in der `web.xml` der Zugriff auf eine Ressource eingeschränkt werden. Die einfachste Lösung wäre, direkt das für die Rolle freigegebene Facelet anzugeben:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>user list</web-resource-name>
        <url-pattern>/users.xhtml</url-pattern>
    </web-resource-collection>
    <auth-constraint>
         <role-name>ADMIN</role-name>
    </auth-constraint>
</security-constraint>
```

Unterhalb des Zweiges `web-resource-collection` können wir dort beliebige viele Facelets konkret innerhalb des Tags `url-pattern` angeben. Das ist allerdings selten sinnvoll. In der Praxis wird es häufig so geregelt, dass die Webseiten für eine dedizierte Rolle alle in einem eigenen Unterverzeichnis zu finden sind. Für unsere Administrator:innen-Rolle bietet sich das Verzeichnis `admin` an, das ihr jetzt direkt unterhalb von `src/main/webapp` anlegt. Verschiebt das Facelet `users.xhtml` in das soeben angelegte Verzeichnis. Passt dann das URL-Muster entsprechend an:

```
    <url-pattern>/admin/*</url-pattern>
```

Damit passen jetzt alle Dateien und Unterverzeichnisse des Verzeichnisses `admin` auf dieses Muster und sind damit nur noch für angemeldete Nutzer:innen mit der Rolle `ADMIN` zugreifbar. Natürlich ändert sich auch die URL für die Liste der Nutzer:innen zu

`http://localhost:8080/eksadat/admin/users.xhtml`.

Jetzt müssen wir natürlich noch eine Möglichkeit schaffen, damit sich Nutzer:innen mit ihrer E-Mail-Adresse und ihrem Passwort am System anmelden können. Dazu erstellt ihr das Facelet `login.xhtml` mit der üblichen Struktur.

In den `body`-Zweig tragt ihr dann folgendes ein:

```
1  <h1>Login</h1>
2  <h:form>
3      <div>
4          <h:outputText value="Email-Adress"/>
5          <h:inputText id="email" value="${loginController.email}"/>
6      </div>
7      <div>
8          <h:outputText id="label_pwd" value="Password"/>
9          <h:inputSecret id="pwd" value="${loginController.pwd}"/>
10     </div>
11     <div>
12         <h:commandButton value="Login" action="#{loginController.login}"/>
13     </div>
14     <h:messages />
15 </h:form>
```

In Zeile 1 erstellen wir wieder eine Überschrift. Da wir eine Eingabe an den Server übermitteln möchten, erstellen wir wieder ein Formular (Zeilen 2 bis 15). In Zeile 4 lassen wir einen Ausgabetext anzeigen und platzieren ein Text-Eingabefeld (Zeile 5) direkt dahinter. Was dort

eingegeben wird, wird in die Property `email` der Backing Bean übertragen (per entsprechender Setter-Methode).

In den Zeilen 8 und 9 wird analog eine Text-Ausgabe und Eingabe für das Passwort definiert. Die Bedeutung von `h:inputSecret` in Zeile 9 könnt ihr vermutlich schon erahnen – ob eure Ahnung zutrifft, könnt ihr später in der deployten Webseite überprüfen.

Schließlich wird in Zeile 12 eine Schaltfläche definiert, die bei Anklicken die Methode *login()* der Backing Bean aufruft.

Die Ausgabe-Eingabe-Paare und die Schaltfläche sind allerdings jeweils von `div`-Elementen eingerahmt. Ein `div`-Zweig ist ein generischer Container für Flussinhalte, der noch weiter gestaltet werden kann (später mehr). Hier ist es nützlich, weil es als sogenanntes *Blockelement* einerseits Elemente gruppiert und andererseits standardmäßig einen Zeilenumbruch vor und nach dem Element erzeugt. In unserem Fall werden dadurch die Ausgabe-Eingabe-Paare und die Schaltfläche untereinander dargestellt, statt alle direkt hintereinander in einer Zeile. Eine neue Zeile in einem HTML-Quelltext wird nicht als neue Zeile von den Browsern gerendet!

In Zeile 14 definieren wir dann noch einen Platzhalter für globale Nachrichten.

Wie wir dem Facelet entnehmen können, benötigen wir offensichtlich eine `LoginController`, die die eingegebene E-Mail-Adresse und das Passwort speichert bzw. bereitstellt. Sie benötigt auch die Methode *login()*, die beim Anklicken der Schaltfläche aufgerufen wird. Legt also im Paket `controller` eine Backing Bean `LoginController` an, die die `String`-Attribute `email` und `pwd` hat und stellt sicher, dass es Getter- und Setter-Methoden für beide gibt. Gebt der Bean den Request-Scope.

Für die Login-Methode benötigen wir drei weitere Attribute, die wir uns von CDI injizieren lassen (und für die weder Getter- noch Setter-Methoden benötigt werden):

```
1  import jakarta.security.enterprise.SecurityContext;
2  import jakarta.servlet.http.HttpServletRequest;
3  import jakarta.servlet.http.HttpServletResponse;
4  ...
5  public class LoginController {
6      ...
7      @Inject
8      private SecurityContext securityContext;
9
10     @Inject
11     private ExternalContext externalContext;
12
13     @Inject
14     private FacesContext facesContext;
15     ...
```

In der Login-Methode fragen wir zunächst beim Security-Kontext den Authentifizierungsstatus für die Kombination aus E-Mail-Adresse und Passwort ab:

```
1  public void login() {
2      final AuthenticationStatus authStatus = securityContext.authenticate(
3                  (HttpServletRequest) externalContext.getRequest(),
4                  (HttpServletResponse) externalContext.getResponse(),
5                  AuthenticationParameters.withParams().credential(
6                                  new UsernamePasswordCredential(email, pwd))
7                  );
```

Die Methode *authenticate()* erwartet drei Parameter: den HTTP-Request, die HTTP-Response und ein *Credential*-Objekt, in dem die Anmeldeinformationen gespeichert sind. Da wir neben der E-Mail-Adresse als Username ein Passwort zur Authentifizierung verwenden, erstellen wir ein entsprechendes Objekt vom Typ `UsernamePasswordCredential` mit den im Facelet eingebenen Werten.

Der Aufruf dieser Methode signalisiert dem Container (programmatisch ausgelöst), dass er einen Web-/HTTP-basierten Authentifizierungsdialog mit der Aufrufer:in starten oder fortsetzen soll. Programmatisches Auslösen bedeutet, dass der Container so reagiert, als hätte die Aufrufer:in versucht, auf eine eingeschränkte Ressource zuzugreifen. Der Container handelt, indem er den konfigurierten Authentifizierungsmechanismus aufruft.

In einem `switch`-Statement reagieren wir jetzt auf den zurückgegebenen Authentifizierungsstatus:

```
1  switch (authStatus) {
2      case SEND_CONTINUE -> facesContext.responseComplete();
3      case SEND_FAILURE -> facesContext.addMessage(null,
4                               new FacesMessage(FacesMessage.SEVERITY_ERROR,
5                               "Login failed", null));
6      case SUCCESS -> facesContext.addMessage(null,
7                           new FacesMessage(FacesMessage.SEVERITY_INFO,
8                           "Login succeeded", null));
9      case NOT_DONE -> { // doesn't happen
10                       }
11 }
```

In Zeile 2 reagieren wir auf den Status `SEND_CONTINUE`. Dieser bedeutet, dass eine Authentifizierung bereits im Gange ist. In diesem Fall halten wir uns aus allem heraus und beenden den Lebenszyklus unserer Backing Bean durch den Aufruf einer entsprechenden Methode im Faces Context.

In Zeile 3 reagieren wir darauf, dass die Authentifizierung fehlgeschlagen ist (z. B. durch ein falsches Passwort). In diesem Fall zeigen wir eine Nachricht im Facelet an, die auf diese Tatsache hinweist. Da wir keine *Client-ID* mitgeben, handelt es sich um eine globale Nachricht[11].

In Zeile 6 reagieren wir darauf, dass die Authentifizierung funktioniert hat. Auch hier zeigen wir erstmal lediglich eine entsprechende Nachricht im Facelet an.

Zeile 9 behandelt den Fall, dass der Container bzw. der Authentifizierungsmechanismus eine Authentifizierung nicht für nötig hält und dementsprechend auch nicht bearbeitet. Das kann hier allerdings nicht passieren, was durch den entsprechenden Kommentar verdeutlicht wird.

Aufgabe 5.2 Login im Backend

Um die Anmeldedaten einer Nutzer:in zu verifizieren, wird eine Implementierung des Interfaces `IdentityStore`[12] benötigt. Ihr findet eine (funktionierende) Basisimplementierung in der Datei `EksadatIdentityStore.java` bei Stud.IP. Sie hat folgenden Inhalt:

[11] `https://jakarta.ee/specifications/faces/4.0/apidocs/jakarta/faces/context/facescontext#addMessage(java.lang.String,jakarta.faces.application.FacesMessage)`

[12] `https://jakarta.ee/specifications/security/3.0/apidocs/jakarta.security/jakarta/security/enterprise/identitystore/identitystore`

```
1  @Slf4j
2  @RequestScoped
3  public class EksadatIdentityStore implements IdentityStore {
4
5      @Inject
6      private UserService userService;
7
8      @Override
9      public CredentialValidationResult validate(final Credential credential) {
10         if (!(credential instanceof UsernamePasswordCredential login)) {
11             log.error("""
12                       Given credentials are of type {}
13                       instead of expected type UsernamePasswordCredential.
14                       Credentials are not validated.""",
15                           credential.getClass().getName());
16             return CredentialValidationResult.NOT_VALIDATED_RESULT;
17         }
18         String email = login.getCaller();
19         String password = login.getPasswordAsString();
20         Optional<User> optionalUser = userService.findByEmailAndPassword(
21                                                                 email, password);
22         if (optionalUser.isPresent()) {
23             final User user = optionalUser.get();
24             final Set<String> rolesAsStrings = user.getRoles()
25                                                 .stream()
26                                                 .map(Enum::toString)
27                                                 .collect(Collectors.toSet());
28             return new CredentialValidationResult(user.getEmail(),
29                                                                 rolesAsStrings);
30         } else {
31             return CredentialValidationResult.INVALID_RESULT;
32         }
33     }
34 }
```

Wir überschreiben hier lediglich die Methode *validate()* in einer für unsere Webanwendung geeigneten Art und Weise. Um die gegebenen Daten zu überprüfen, benötigen wir Zugriff auf den Datenbestand. Dieser wird uns in unserer Architektur über den entsprechenden Service gewährt, so dass wir in Zeile 6 den `UserService` injizieren lassen. In Zeile 10 wird dann geprüft, ob das gebenene `Credential`-Objekt dem von uns erwarteten Typ entspricht. Wenn das nicht der Fall ist, protokollieren wir eine entsprechende Fehlermeldung und beenden die Methode mit einem Rückgabewert, der anzeigt, dass eine Validierung der Daten nicht versucht wurde.

Ansonsten ermitteln wir die Email-Adresse und das Passwort aus dem gegebenen Objekt und lassen uns ein optionales `User`-Objekt zu diesen Daten vom User-Service geben. Sollte der Service ein passendes Objekt zurückgeben, wird ein `CredentialValidationResult`-Objekt erzeugt und zurückgegeben. Dabei ist zu beachten, dass der Konstruktor[13] dieses Objektes für die Rollen

[13] `https://jakarta.ee/specifications/security/3.0/apidocs/jakarta.security/jakarta/security/enterprise/identitystore/credentialvalidationresult#<init>(java.lang.String,java.util.Set)`

eine Menge von Strings benötigt. Daher wird die Rückgabe von `User#getRoles()` entsprechend konvertiert (Zeilen 23 bis 26).

Wenn kein passendes Objekt vom User-Service zurückgegeben wurde, bedeutet das, dass die Kombination aus Email-Adresse und Passwort nicht im Datenbestand vorhanden ist. In diesem Fall wird in Zeile 31 eine entsprechende Konstante zurückgegeben, die diese Tatsache anzeigt.

Kopiert die Datei `EksadatIdentityStore` in das Unterpaket `security` und implementiert dann die fehlende Methode *findByEmailAndPassword()* in der Klasse `UserService`.

WildFly konfigurieren

Per default nutzt WildFly ein eigenes Security-System, so dass wir die Nutzung von Jakarta Security erst aktivieren müssen. Nutzt dazu eine interaktive Sitzung im *command line interface (CLI)* des WildFly[14]. Nach Eingabe von `connect` (der WildFly-Server muss dafür gestartet sein und laufen) gebt ihr dann folgendes ein (alles in eine Zeile **OHNE** den roten Pfeil)

```
/subsystem=undertow/application-security-domain=other:write-attribute(
 ↪ name=integrated-jaspi, value=false)
```

Damit deaktivieren für die Security-Domäne *other* das in WildFly integrierte *JASPI* (Java Authentication Service Provider Interface)[15]. Die Security-Domäne *other* wird innerhalb von WildFly für Jakarta EE Security verwendet.

Durch die Verwendung von Jakarta Security stellen wir sicher, dass die Anwendung auch in einen anderen Application Container, der Jakarta EE bereitstellt, umgezogen werden kann.

Sorgt dann durch Eingabe von `reload` dafür, dass die geänderte Konfiguration im laufenden WildFly-Prozess geladen wird (oder startet den WildFly neu).

Aufgabe 5.3 Login to continue

Unsere Webanwendung „weiß“ jetzt allerdings noch nicht, welchen Authorisierungsmechanismus sie verwendet. Das bringen wir ihr jetzt bei, indem wir unserer Startup-Bean `EksadatConfig` folgende Annotation auf Klassenebene hinzufügen:

```
@CustomFormAuthenticationMechanismDefinition(loginToContinue =
                    @LoginToContinue(
                              loginPage = "/login.xhtml",
                              errorPage = ""
                    )
)
```

Wir legen damit generell fest, dass wir ein selbsterstelltes Formular (*Custom Form*) für den Authentifizierungsmechanismus verwenden. Gleichzeitig konfigurieren wir *login to continue* und geben dafür an, wo unser Formular zur Eingabe der Anmeldedaten liegt und wie es heißt (auf

[14] `https://docs.wildfly.org/29/Admin_Guide.html#Command_Line_Interface`
[15] Was JASPI ist und wie die Dinge historisch entstanden sind, erklärt Arjan hier ganz gut: `https://www.eclipse.org/community/eclipse_newsletter/2019/may/EEsecurity.php`

eine Fehlerseite bei fehlgeschlagener Anmeldung verzichten wir zunächst). „Login to continue“ ist ein nützliches Feature, das dazu führt, dass wir im Falle des Aufrufs einer geschützten URL auf die Seite mit dem Anmeldeformular umgeleitet werden. Im Fall einer erfolgreichen Anmeldung werden wir dann auf die ursprüngliche URL weitergeleitet.

Probiert das jetzt mal aus, indem ihr eure Webanwendung deployt und direkt die URL für die Anzeige der Nutzer:innen `http://localhost:8080/eksadat/admin/users.xhtml` aufruft. Ihr solltet direkt auf die Login-Seite umgeleitet werden. Wenn ihr dort die Daten für die Nutzer:in mit der Rolle `ADMIN` eingebt sollte eine weitere Umleitung auf die Anzeige der Nutzer:innen erfolgen.

Abmelden können wir uns aktuell noch nicht, daher müsst ihr die Webanwendung neu deployen oder den WildFly neu starten, um eine Abmeldung durchzuführen.

Leider gibt es einen lange bekannten, aber sehr komplexen Bug in WildFly[a], durch den zwei Warnungen ausgelöst werden, wenn durch *login-to-continue* die umgeleiteten Seiten geladen werden. Die Warnung hat zwar keinen Einfluss auf die Funktionalität der Webanwendung, aber sie wird protokolliert und nervt daher möglicherweise. Ihr findet am Ende des Aufgabenblattes im Abschnitt *Zusatz: Log-Nachrichten filtern* eine Lösung dafür.

[a] `https://issues.redhat.com/browse/WFLY-15972`

Aufgabe 5.4 Liste der Prüflinge schützen

Ihr wisst jetzt, wie ihr Facelets bzw. ganze Unterverzeichnisse vor unbefugtem Zugriff schützen könnt. Erzeugt ein Unterverzeichnis `user` im Verzeichnis `webapp`, verschiebt das `students.xhtml`-Facelet dort hinein und konfiguriert die Webanwendung so, dass nur Nutzer:innen mit der Rolle `USER` die Liste der Prüflinge einsehen können[16].

Überprüft dann, ob der rollenbasierte Zugriff tatsächlich wie gewünscht funktioniert.

Aufgabe 6 Rollenbasierte Zugriffskontrolle im Frontend

Abschließend schauen wir uns noch an, wie eine Zugriffskontrolle auf bestimmte Teile eines Faceletes konfiguriert werden kann.

Aufgabe 6.1 Verweise anzeigen und schützen

Dazu ändern wir unsere Startseite `hello.xhtml`. Löscht alles aus dem `h:body`-Zweig und fügt dort stattdessen folgendes ein:

```
<h1>Welcome to Eksadat!</h1>
<div>
    <h:outputLink value="admin/users.xhtml">List of users</h:outputLink>
</div>
<h:outputLink value="user/students.xhtml">List of students</h:outputLink>
```

[16] Tipp: Es sind zusätzliche `security-roles` und `security-constraints` nötig

Das Tag `h:outputLink` erzeugt einen statischen Verweis (*Link*) auf die im Attribut `value` angegebene URL. Es kann immer dann genutzt werden, wenn eine direkte Navigation gewünscht ist (ohne Aufruf einer Methode in der Backing Bean). Wird von einer Nutzer:in der Link angeklickt, so lädt der Browser direkt die URL aus dem `value`-Attribut.

Löscht jetzt auch die Klasse `HelloController`, da sie nicht mehr benötigt wird.

Wir möchten jetzt erreichen, dass in unserem `hello`-Facelet nur die Links angezeigt werden, die von der jeweiligen Rolle einer Nutzer:in auch verwendet werden können. Dazu gibt es im Wesentlichen zwei Möglichkeiten:

1. Das Attribut `rendered`
2. Jarkata Tag `c:if`

Das Attribut `rendered` gibt an, unter welchen Bedingungen eine Komponente gerendert, also angezeigt wird. Nur wenn der Ausdruck, der als `rendered`-Wert angegeben wird, zu `true` ausgewertet wird, wird die Komponente angezeigt. In unserem Fall könnten wir eine EL-Expression verwenden:

```
<h:outputLink rendered="#{request.isUserInRole('ADMIN')}"
              value="admin/users.xhtml">List of users</h:outputLink>
```

In unserer EL-Expression rufen wir die Methode *isUserInRole()* auf dem HTTP-Request (genauer: dem `HttpServletRequest`[17], der uns in EL-Expressions als Objekt zur Verfügung steht) auf. Diese Methode erwartet einen Parameter vom Typ String – nämlich die zu prüfende Rolle. Wir verwenden hier für das String-Literal statt Gänsefüßchen einfache Anführungszeichen, da der EL-Ausdruck selbst in Gänsefüßchen eingeschlossen ist[18]. Die Methode liefert `true` genau dann, wenn der zum Request gehörende User die entsprechende Rolle hat.

Das `rendered`-Attribut wird in jeder Phase im Lebenszyklus des Facelets[19] ausgewertet.

Wenn Jakarta Tags verwendet werden, so findet die Auswertung nur ein Mal statt und zwar beim Aufbau der Webseite. Jakarta Tags manifestieren sich auch nicht in der ausgelieferten HTML-Seite, die durch ihre Verwendung also „schlanker“ wird.

Die Nutzung sieht dann in unserem Fall so aus:

```
<c:if test="#{request.isUserInRole('ADMIN')}">
    <h:outputLink value="admin/users.xhtml">List of users</h:outputLink>
</c:if>
```

Natürlich müssen wir den Namensraum für Tags mit dem Präfix `c` in unserem Facelet bekannt machen:

```
xmlns:c="jakarta.tags.core"
```

Sorgt jetzt dafür, dass das `Hello`-Facelet den Verweis auf die User-Liste nur für Nutzer:innen mit der Rolle `ADMIN` und den Verweis auf die Liste der Prüflinge nur für die Nutzer:innen mit der Rolle `USER` anzeigt. Fügt einen dritten Verweis ein, der auf das `Login`-Facelet verweist und der

[17] `https://jakarta.eaworld.io/specifications/platform/10/apidocs/jakarta/servlet/http/HttpServletRequest.html`

[18] Ihr könnt die Anführungszeichen auch genau umtauschen, d. h. den EL-Ausdruck insgesamt in einfache Anführungszeichen (') und Strings wie in Java üblich in Gänsefüßchen setzen (")

[19] `https://jakarta.eaworld.io/specifications/faces/4.0/jakarta-faces-4.0.html#a401`

nur angezeigt wird, wenn die Nutzer:in noch nicht angemeldet ist. Dazu könnt ihr die folgende EL-Expression nutzen:

```
"#{empty request.userPrincipal}"
```

Damit fragt ihr den Fall ab, dass noch niemand angemeldet ist („Die Nutzer:in im Request ist leer“). **Hinweis:** Schreibt ihr noch ein `not` davor, könnt ihr abfragen, ob eine Nutzer:in allgemein angemeldet ist – unabhängig von ihren konkreten Rollen.

Ändert jetzt die Methode *login()* in der `LoginController` derart, dass sie einen String zurückgibt. Im Fall des erfolgreichen Logins soll sie den String `"hello"` zurückgeben und in allen anderen Fällen den Wert `null`. Darüber wird die Navigation auf eine neue bzw. die gleiche Seite geregelt. Wenn der Rückgabewert `null` ist, wird die aktuelle Seite neu geladen. Ansonsten wird die Seite geladen, die durch den Rückgabewert benannt ist.

Aufgabe 6.2 Logout

Analog zum `Login`-Facelet erstellt ihr jetzt ein Facelet mit passender Backing Bean für den Logout. Im `Logout`-Facelet gibt es nur eine Schaltfläche, die beim Betätigen eine entsprechende Methode in der Backing Bean aufruft. In dieser Methode werden die folgenden Methoden aufgerufen:

```
request.logout();
request.getSession().invalidate();
```

Der `request` hat dabei den Typ `HttpServletRequest`. Das entsprechende Objekt könnt ihr euch einfach von CDI injizieren lassen.

Durch die erste Anweisung (die eine `ServletException` auslösen kann – behandelt sie durch eine entsprechende `throws`-Klausel an der Methodensignatur) wird nur der Status der aktuellen Session des Security Frameworks gelöscht. Mit der zweiten Anweisung wird sichergestellt, dass auch die zugehörige Session sofort invalidiert wird.

In vielen Webanwendungen wird nach einem Logout wieder die Login-Seite angezeigt. Das nötige Wissen zur Umsetzung dieses Features habt ihr bereits. Ihr könnt das allerdings hier machen wie der Pfarrer Assmann. Der macht es so wie der Pastor Nolte. Und der Pastor Nolte, der machte immer, was er wollte[20].

Wenn ihr euch für ein *Redirect* auf die Login-Seite entschieden habt, werdet ihr vielleicht bemerken, dass zwar die Login-Seite angezeigt wird, die URL im Browser aber noch auf `.../logout.xhtml` steht. Wenn ihr einen „echten“ Redirect auslösen möchtet, dann hängt ihr an den Rückgabewert noch das Suffix `?faces-redirect=true` an. Dann wird auch die URL im Browser korrekt aktualisiert (weil ihr tatsächlich eine neue Anfrage mit dem Ziel `login.xhtml` gesendet habt).

Zusatz: Log-Nachrichten filtern

Der WildFly kann so konfiguriert werden, dass bestimmte Log-Nachrichten unterdrückt werden (tatsächlich können Log-Nachrichten sogar angepasst werden, indem bestimmte Muster darin

[20] nach Bernd Stromberg in `https://de.wikipedia.org/wiki/Stromberg_(Fernsehserie)`

durch andere ersetzt oder entfernt werden – z. B. wenn sensitive Informationen wie Passwörter durch verwendete Frameworks protokolliert werden).

Aktuell ist es nur möglich, den protokollierten Text (also die Log-Nachricht) zu verwenden – es kann also nicht z. B. nach Exception o. ä. gefiltert werden.

Um die beiden für uns relevanten Log-Ausgaben zu unterdrücken, müssen wir zunächst mal definieren, um welche Nachrichten es sich handelt. Dazu bietet die WildFly-Konfiguration ein oder mehrere Filter-Spezifikationen an. Darin kann ein Regulärer Ausdruck aus Java verwendet werden[21].

```
match("WELD-000717:.*\\[ GET /eksadat/login.xhtml \\]")
```

Das Schlüsselwort `match` zeigt an, dass ein Muster folgt. Alle Zeichenketten, die auf das Muster passen, werden von diesem `match` „gefunden“. Hier soll die Zeichenkette mit der exakten Buchstabenfolge `WELD-000717:` beginnen. Dann folgen durch `.*` beliebige Zeichen. Damit wir nur die Log-Ausgaben finden, die sich konkret auf unser Login-Facelet beziehen, muss die Zeichenkette sich nur darauf beziehen und entsprechend folgendes enthalten: (`[ GET /eksadat/login.xhtml ]`). Da die eckigen Klammern in regulären Ausdrücken eine eigene Bedeutung haben, müssen sie *gequoted* werden. Das geschieht in Java (und vielen anderen) regulären Ausdrücken mittels Backslash (\). Da es sich bei dem Muster um einen Java-String handelt, muss der Backslash allerdings durch einen Backslash erneut gequoted werden.

Der reguläre Ausdruck für die zweite Log-Ausgabe sieht ganz ähnlich aus:

```
match("WELD-000335:.*\\[ POST /eksadat/login.xhtml \\]")
```

Da die Zeichenketten auf eines der beiden Muster passen sollen, können wir sie über `any` zusammenfassen:

```
any(<filter_expression, <filter_expression>, ...)
```

Um die Nachrichten tatsächlich zu unterdrücken, negieren wir das Ergebnis des *Pattern Matchings*:

```
not(<filter_expression)
```

Durch `not` legen wir fest, dass alle Log-Ausgaben, die auf das folgende Muster passen, unterdrückt werden. Weitere Informationen zu den Filter-Ausdrücken[22] findet ihr im *Admin Guide* des WildFly.

Insgesamt ergibt sich also ein Filterausdruck der Art:

```
not(any(match(...), match(...)))
```

Es gibt verschiedene Möglichkeiten, die Filter-Ausdrücke zu konfigurieren. Die vermutlich einfachste besteht darin, direkt die Konfiguration des `standalone`-WildFly anzupassen. Dazu öffnet ihr die Datei `standalone.xml` im Verzeichnis `standalone/configuration` eures WildFly-Installationsverzeichnisses. Darin findet ihr neben vielen anderen Konfigurationsoptionen auch Zweige für die Logger, z. B.

```
<console-handler name="CONSOLE">
```

[21] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/Pattern.html`
[22] `https://docs.wildfly.org/29/Admin_Guide.html#filter-expressions`

Darunter könnt ihr dann die Filter-Spezifikation eintragen (alles in eine Zeile ohne die roten Pfeile):

```
1 <filter-spec value="not(any(match(&quot;WELD-000717:.*\\[ GET
  ↪ /eksadat/login.xhtml \\]&quot;),match(&quot;WELD-000335:.*\\[ POST
  ↪ /eksadat/login.xhtml \\]&quot;)))"/>
```

Die Anführungszeichen müssen dabei durch `&quot;` ersetzt werden, da die Anführungszeichen selbst Attribute in XML-Elementen einschließen.

Wenn die Log-Ausgaben auch in den Protokoll-Dateien unterdrückt werden sollen, tragt ihr sie genauso in die anderen Zweige ein, z. B. unterhalb von

```
<periodic-rotating-file-handler name="FILE" autoflush="true">
```

Der WildFly verändert die Konfigurationsdatei während des Betriebes. Er darf also während einer entsprechenden Eintragung in die `standalone.xml` nicht ausgeführt werden.

Alternativ könnt ihr die Filter-Spezifikation auch über das CLI (s. o.) für die jeweiligen Handler konfigurieren:

```
/subsystem=logging/console-handler=CONSOLE:write-attribute(name=filter-spec,
  ↪ value="not(any(match(\"WELD-000717...),...))"
```