Software-Projekt WiSe 23/24

Aufgabenblatt 5

Abgabe bis: 07.01.2024 23:59 Uhr MEZ

Autoren: Karsten Hölscher, Marcel Steinbeck

Auf diesem Aufgabenblatt werden wir allgemeine Fehlerseiten und ein angemessenes Feedback für Nutzer:innen im Falle von Exceptions betrachten. Im zweiten Teil schauen wir uns isolierte Komponententests mit Mockito und das Messen der Testüberdeckung an.

Aufgabe 1 Fehlerseiten

Bisher haben wir Fehler nicht weiter betrachtet. Das soll sich jetzt ändern. Um zu sehen, was aktuell im Fehlerfall passiert, ändern wir die Entität `Exam` derart ab, dass die Namensspalte eindeutig sein muss:

```
@Column(unique = true)
private String name;
```

Legt jetzt nach einem Deployment der Änderungen zwei Prüfungen mit demselben Namen an. Dadurch, dass die Namen von Prüfungen gemäß unserer obigen Änderung jetzt eindeutig sein müssen, verletzen wir einen Constraint im Datenbestand. Daher kommt es zur einer Exception beim Speichern der zweiten Prüfung. Im Browser-Fenster sehen wir neben einer Fehlernachricht auch den Stack-Trace - ähnlich zu Abbildung 1.

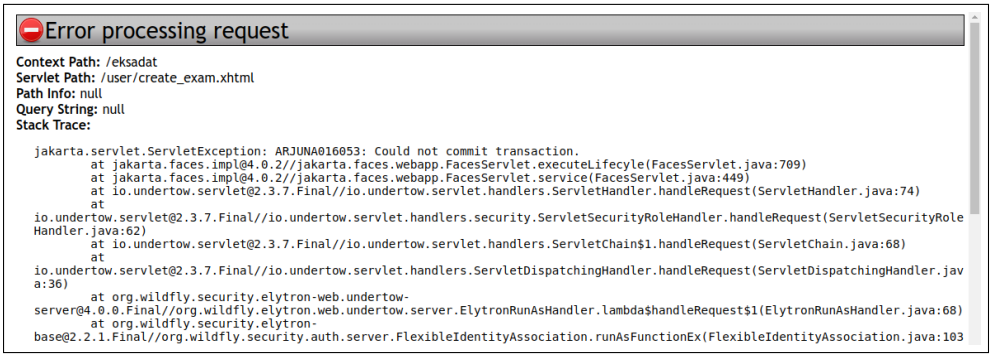

Abbildung 1: Fehlermeldung im Browserfenster

Hier sehen wir, dass die Transaktion nicht committed werden konnte. Ausgelöst wird dieser „letzte“ Fehler in der Kette ursprünglich (im Browser ganz nach unten scrollen) durch eine `JdbcSQLIntegrityConstraintViolationException`, die aus dem H2-Treiber kommt.

Eine Nutzer:in kann natürlich überhaupt nichts mit einer derartigen Fehlerseite anfangen. Zudem hilft die angezeigte Seite auch nicht weiter im Bezug auf das weitere Vorgehen, denn es gibt keinerlei Navigationshilfen o. ä.

Um diese Situation zu verbessern, können im Deployment-Descriptor (`web.xml`) Fehlerseiten für verschiedene Fehlersituationen definiert werden. Eine Exception im Backend wird von Faces als

interner Serverfehler interpretiert und die Antwortseite entsprechend mit dem *Status-Code*[1] 500 für *Internal Server Error* ausgeliefert.

Erstellt also ein Facelet `500.xhtml` für eine Fehlerseite und bietet darin z. B. einen Link zu Übersichtsseite (`hello.xhtml`) an. Konfiguriert die Fehlerseite dann im Deployment-Descriptor wie folgt:

```
<error-page>
    <error-code>500</error-code>
    <location>/WEB-INF/errorpages/500.xhtml</location>
</error-page>
```

Der Kontext dieser Fehlerseite entspricht dem Kontext der Webseite, in der der Fehler aufgetreten ist. Wenn also der Fehler – wie hier – auf der Seite `/user/create_exam.xhtml` auftritt, dann bezieht sich der Kontext der Fehlerseite auf das Verzeichnis `user` in unserer Webanwendung. Ein `outputLink` der Form `<h:outputLink value="hello.xhtml">` würde dann auf die URL `http.../eksadat/user/hello.xhtml` verweisen. Ein führender Slash, `<h:outputLink value=/hello.xhtml">`, würde auch nicht helfen – denn das würde die URL `http://localhost:8080/hello.xhtml` bedeuten. Das wäre eine Webseite außerhalb unserer Anwendung.
Die Basis-URL für unsere Anwendung erhalten wir im Facelet durch einen EL-Ausdruck: `"#{facesContext.externalContext.requestContextPath}/..."` Darüber können wir absoulte Pfade im Facelet spezifizieren.

Legt nach einem erneuten Deployment der Änderungen zwei Prüfungen mit demselben Namen an. Stellt sicher, dass der Inhalt der Seite `500.xhtml` im Browser angezeigt wird.

Aufgabe 2 Fehlermeldungen

Wenn wir uns nochmal die Entstehung des Fehlers anschauen, dann stellen wir fest, dass es sich dabei nicht um einen technischen und unvorhersehbaren Fehler handelt, sondern um einen Fehler, der sich aus der Fachdomäne ergibt. Die Fachdomäne bzw. die Kund:innen möchten eindeutige Namen für die Prüfungen (in einer „richtigen“ Anwendung würde das vermutlich auf eindeutige Namen pro Veranstaltung o. ä. aufgeweicht). Statt einer allgemeinen Fehlerseite, die keine spezifischen Gründe für das Auftreten des Fehlers benennt, wäre es hier besser, den Nutzer:innen das Feedback zu geben, dass es den Namen für die zu erzeugende Prüfung bereits gibt. Dann ist klar, wo das Problem liegt und der Fehler kann in Zukunft vermieden werden.

Jetzt könnten wir versuchen, den konkreten Fehler aus der Exception bzw. der Exception-Kette zu ermitteln. Die ursprünglich auslösende Exception (s. o.) ist recht sprechend: Es wird ein „Integrity Constraint“ verletzt. Im Kontext unserer bisherigen Anwendung ist dann klar, dass das mit hoher Wahrscheinlichkeit bedeutet, dass die globale Eindeutigkeit des Namens der Prüfung verletzt ist. Das lässt sich auch aus der Message der Exception herauslesen. Die Nutzung dieser Information ist aber in der Praxis schwierig. Mit einem Versionsupdate in einer verwendeten Bibliothek mag sich die Message ändern. Und was passiert, wenn wir später mal ein anderes DBMS einsetzen als H2? Dann ändert sich möglicherweise sogar der Typ der Exception.

[1] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses`

Es ist hier also besser, den Fehler vorherzusehen und vor dem Persistieren zu überprüfen. Ein sinnvoller Ort dafür ist in unserer Architektur die Service-Klasse. Diese muss in der *save()*-Methode zunächst prüfen, ob es bereits eine Prüfung mit dem gegebenen Namen gibt. Wenn das so ist, sollte dann eine Exception unserer Webanwendung ausgelöst werden.

Kümmert euch zunächst um die Exception. Erstellt dazu das Unterpaket `exception` und darin dann eine Klasse `DuplicateNameException`, die `RuntimeException` erweitert und einen Konstruktor mit einem Parameter für die Nachricht hat.

Erweitert dann das `ExamRepository` um eine Methode, die eine optionale Prüfung zu einem gegebenen Namen zurückgibt.

Die eigentliche Überprüfung findet dann in der Service-Klasse `ExamService` statt. Überschreibt dort die *save()*-Methode derart, dass sie vor dem Aufruf der *save()*-Methode der Superklasse prüft, ob es bereits eine Prüfung mit dem Namen der zu speichernden Prüfung gibt und in diesem Fall die obige Exception auslöst.

Jetzt könnt ihr den `CreateExamController` anpassen. Prüft dort, ob beim Speichern der Prüfung die obige Exception ausgelöst wird und zeigt in diesem Fall die Message im Facelet an. Das geht z. B. so:

```
FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR, "Error occurred",
            exception.getMessage()));
```

Die Methode *addMessage()*[2] der Klasse `FacesContext` erwartet als ersten Parameter den Bezeichner der Anzeigekomponente für die Fehlermeldung. Wir setzen ihn hier auf `null`, da wir in Kürze im Facelet eine Komponente für globale Fehlermeldungen deklarieren. Es folgt ein Parameter für die Nachricht vom Typ `FacesMessage`[3]. Der hier verwendete Konstruktor erwartet eine Konstante, die die Schwere (*severity*) der Nachricht angibt – in unserem Fall ist es ein Fehler (*error*). Dann folgt eine Zeichenkette mit einer kurzen Zusammenfassung und dann eine Zeichenkette mit Details der Nachricht.

Damit wir diese Nachricht jetzt sehen können sind zwei Dinge nötig. Damit wir die Nachricht überhaupt zu sehen bekommen, dürfen wir nicht über einen Redirect eine neue Seite laden. Zudem ist es generell sinnvoll, bei gescheiterten Eingaben nicht von der entsprechenden Eingabeseite weg zu navigieren – da die Eingabe ja korrigiert werden kann, wenn der Fehler verstanden wird. Daher solltet ihr unter der obigen Zeile die Ausführung der Methode mit

```
return null;
```

beenden. Damit wird weiterhin die ursprüngliche Seite im Browser angezeigt.

Jetzt könnten wir die Nachricht sehen, aber es ist im Facelet noch nicht definiert, wie und wo sie angezeigt werden soll. Ergänzt daher das Facelet derart, dass ihr am Anfang des Formulars

```
<p:growl globalOnly="true" />
```

einfügt. Damit verwenden wir ein *Growl*[4] (eine Nachricht, die in einem *Overlay* angezeigt wird) von PrimeFaces, um die Nachricht anzuzeigen. Das Attribut `globalOnly` setzen wir auf `true`, damit im Growl nur Nachrichten angezeigt werden, die kein explizites Ziel haben.

[2] `https://jakarta.ee/specifications/faces/4.0/apidocs/jakarta/faces/context/facescontext#addMessage(java.lang.String,jakarta.faces.application.FacesMessage)`
[3] `https://jakarta.ee/specifications/faces/4.0/apidocs/jakarta/faces/application/facesmessage`
[4] `https://www.primefaces.org/showcase/ui/message/growl.xhtml`

Probiert jetzt aus, was passiert, wenn ihr zwei Prüfungen mit dem gleichen Namen erzeugt. Wir ihr seht, werden die Details nicht angezeigt. Das können wir leicht beheben, indem wir das Attribut `showDetail` des Growls ebenfalls auf `true` setzen. Passt also das Growl entsprechend an.

Aktuell bleibt das Growl sehr lange im Browser sichtbar (6 Sekunden). Konfiguriert daher über das Attribut `life` eine kürzere Anzeigedauer von nur 3 Sekunden.

Aufgabe 3 Modul-/Komponententests mit Mockito

Bei dieser Art von Tests (auch *Unit-Tests* genannt) sollen Module oder auch ganze Komponenten weitgehend isoliert voneinander getestet werden, d. h. Interaktionen mit anderen Modulen/Komponenten sollen – sofern möglich – *simuliert* werden. Daraus ergibt sich der unmittelbare Vorteil, dass Fehler, die in Tests aufgedeckt wurden, eindeutig einem Modul/einer Komponente zugeordnet werden können – sofern die Tests selbst nicht fehlerhaft sind. Die Anforderung an die Isolation der zu testenden Module/Komponenten lässt sich mit Hilfe von *Mock-Objekten*[5] (im Folgenden auch *Mocks* genannt) realisieren. Mocks sind Stellvertreter der Fremdmodule/Fremdkomponenten, mit der eine zu testende Einheit interagiert. Sie sind schnittstellenkompatibel zu ihren Originalen, ihr Verhalten wird jedoch im Rahmen der Tests statisch festgelegt, was schließlich zur Entkopplung und damit zur gewünschten Isolation führt.

Beispiel: Soll eine Klasse X im Rahmen eines Modultests getestet werden und X hängt von einer anderen Klasse Y ab, dann *sollte* für Y ein Mock erstellt und in X verfügbar gemacht werden. Das Verhalten von Y wird dabei statisch im Test festgelegt, wobei nur genau so viel Logik implementiert wird, wie für den konkreten Testfall notwendig ist – das Verhalten von Mocks soll möglichst *trivial* sein. Unter der Annahme, dass das Verhalten von Y korrekt ist, können aufgedeckte Fehler nun eindeutig der Klasse X zugeordnet werden.

(?) Muss jede Abhängigkeit gemockt werden?

Mocks werden üblicherweise für die Module/Komponenten der eigenen Implementierung erstellt. So ist es z. B. nicht sinnvoll, die Klasse `String` der Java-Standardbibliothek zu mocken. Außerdem bieten sich Mocks immer dann an, wenn Fremdmodule/Fremdkomponenten die automatische Ausführung von Tests erschweren oder gar unmöglich machen – Datenbanken sind häufig ein Beispiel hierfür. Grundsätzlich gilt: Je isolierter ein Modul/eine Komponente getestet wird, desto besser können Fehler eingegrenzt werden.

Mocks eignen sich darüber hinaus auch sehr gut zum Testen von Modulen und Komponenten einer Applikation, die in einem Anwendungsserver läuft (z. B. eine JavaEE-Anwendung). Indem die vom Anwendungsserver bereitgestellten Ressourcen (z. B. FacesContext) gemockt – d. h. es wird ein Mock-Objekt für diese Ressource (Klasse) erstellt – und in die Testobjekte injiziert werden, können Module und Komponenten auch ohne eine Instanz des Anwendungsservers getestet werden.

Für Java gibt es eine Reihe verschiedener Mocking-Bibliotheken und -Frameworks[6] und wie immer gilt auch hier: Jede Bibliothek/jedes Framework hat ganz eigene Vor- und Nachteile. In der Vergangenheit haben wir gute Erfahrungen mit *Mockito*[7] gemacht und können es sowohl für JavaEE- als auch JavaSE-Anwendungen empfehlen.

[5] `https://de.wikipedia.org/wiki/Mock-Objekt`
[6] Eine Auflistung findet ihr unter `https://de.wikipedia.org/wiki/Mocking_Framework#Java`
[7] `https://site.mockito.org`

Aufgabe 3.1 Isolierter Login-Test

Zunächst müssen, wie üblich, die Abhängigkeiten in die `pom.xml` eingetragen werden. Wir beginnen mit JUnit, das die Tests letztendlich ausführt:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

Zu beachten ist hier der Scope `test`. Das bedeutet, dass diese Abhängigkeit nur für Tests besteht. Im Produktivsystem wird JUnit also nicht mit ausgeliefert oder erwartet.

Wir benötigen neben JUnit dann noch den Kern von Mockito selbst:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Und damit JUnit die Mockito Tests auch korrekt ausführt, benötigen wir eine Brücke zwischen den beiden Frameworks:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Erstellt jetzt unterhalb des Verzeichnisses `src` ein Unterverzeichnis `test` und darin ein weiteres Unterverzeichnis `java`. IntelliJ unterstützt euch dabei sinnvoll, wenn ihr in der Project-Ansicht im Kontextmenü (üblicherweise Rechtsklick) des Verzeichnisses `src` erst `New` und dann `Directory` wählt. Maven erwartet nämlich die Testklassen nicht im gleichen Verzeichnis wie die Quelltext-Dateien des auszuliefernden Produktivsystems, sondern in separaten Unterverzeichnissen mit eben den obigen Namen.

Da unsere Testklassen in den gleichen Paketen liegen sollen wie die Prüflinge, solltet wir jetzt die dafür nötige Paket-Struktur erzeugen.

Das könnt ihr umständlich über einzelnes Anlegen der Verzeichnisse machen oder euch auch hier von IntelliJ helfen lassen. Öffnet dazu in der Project-Ansicht das Kontextmenü des Hauptpaketes unserer Webanwendung (`de.unibremen...eksadat`) und wählt dann `Copy Path/Reference` und dann `Copy Reference`. Öffnet dann das Kontextmenü des Verzeichnisses `java` unterhalb von `test` und wählt `New` und dann `Package`. Dort könnt ihr dann die zuvor kopierte Referenz einfügen.

Wir möchten jetzt die *login()*-Methode der Klasse `LoginController` isoliert mit Mockito testen. Da diese im Paket `controller` liegt, erzeugt ihr das Unterpaket `controller` ebenfalls im

Testzweig. Ihr findet bei Stud.IP die Testklasse `LoginControllerTest`. Kopiert sie in das eben erstellte Paket.

Da es sich bei dem folgenden Test um einen Whitebox-Test handelt und eure Implementierung der *login()*-Methode möglicherweise anders aussieht, seht ihr hier die im Folgenden getestete Implementierung:

```
1  public String login() {
2
3      final AuthenticationStatus authStatus = securityContext.authenticate(
4                              (HttpServletRequest) externalContext.getRequest(),
5                              (HttpServletResponse) externalContext.getResponse(),
6                              AuthenticationParameters.withParams().credential(
7                                      new UsernamePasswordCredential(email, pwd)));
8      String forward = null;
9      switch (authStatus) {
10         case SEND_CONTINUE -> facesContext.responseComplete();
11         case SEND_FAILURE -> facesContext.addMessage(null,
12            new FacesMessage(FacesMessage.SEVERITY_ERROR, "Login failed", null));
13         case SUCCESS -> forward = "hello";
14         case NOT_DONE -> { // doesn't happen
15         }
16     }
17     return forward;
18 }
```

Damit können wir uns jetzt der Test-Klasse widmen. Sie sieht ohne Import-Anweisungen so aus:

```
30 @ExtendWith(MockitoExtension.class)
31 public class LoginControllerTest {
32
33     @Mock
34     private SecurityContext securityContext;
35
36     @Mock
37     private ExternalContext externalContext;
38
39     @InjectMocks
40     private LoginController loginController;
```

Die Annotation in Zeile 30 sorgt dafür, dass die Testklasse auch tatsächlich unter Verwendung des Mockito-Frameworks ausgeführt wird.

In Zeile 40 definieren wir ein Attribut für den Prüfling, also die zu testende Klasse. Da wir die Methoden dieser Klasse isoliert testen möchten, müssen wir alle Abhängigkeiten mocken. Das passiert in den Zeilen 33 bis 37, in denen wir Attribute definieren, die Mock-Objekte der dort angegebenen Typen sind. Über die Annotation in Zeile 39 weisen wir das Framework an, die Attribute im Prüfling durch unsere Mock-Objekte zu ersetzen. Gleichzeitig wird dadurch ein Objekt erstellt, das wir direkt nutzen können, ohne es selbst instanziieren zu müssen.

Es geht weiter mit der Testmethode, die ein erfolgreiches Login testet:

```
42 @Test
43 public void testSuccessfulLogin() {
44
45     try (MockedConstruction<UsernamePasswordCredential> anonCredentials =
       ↪ mockConstruction(UsernamePasswordCredential.class);
46     MockedStatic<AuthenticationParameters> authenticationParameters =
       ↪ mockStatic(AuthenticationParameters.class)) {
```

Der Rumpf der Testmethode besteht aus einem `try-with-resources`-Block, in dem zunächst die Klasse `UsernamePasswordCredential` inklusive ihres Konstruktors gemockt wird (Zeile 45). Das ist hier für eine komplette Isolation nötig, denn im Prüfling wird ein Objekt dieser Klasse erzeugt. Zusätzlich wird für den `try`-Block ein statischer Mock für die Klasse `AuthenticationParameters` definiert. Das ist nötig, weil im Prüfling eine statische Methode auf dieser Klasse aufgerufen wird.

Wenn diese Art des Mockings durchgeführt wird, empfiehlt es sich, das wie hier in einem `try-with-resources`-Block zu tun. Da die jeweiligen Objekte das Interface `AutoCloseable` implementieren, werden sie am Ende des `try`-Blocks geschlossen. Das bedeutet hier insbesondere, dass sich die gemockten Klassen außerhalb des `try`-Blocks normal verhalten. Ohne `try-with-resources`-Block könnten die Klassen sich auch außerhalb der Testmethode wie die Mocks verhalten, was nicht immer erwünscht sein wird.

Als nächstes stellen wir die Voraussetzungen für den durchzuführenden Test her.

```
48 // Given
49 HttpServletRequest request = mock(HttpServletRequest.class);
50 HttpServletResponse response = mock(HttpServletResponse.class);
51 AuthenticationParameters withParams = mock(AuthenticationParameters.class);
52 AuthenticationParameters credentials = mock(AuthenticationParameters.class);
53
54 when(externalContext.getRequest()).thenReturn(request);
55 when(externalContext.getResponse()).thenReturn(response);
56 authenticationParameters.when(AuthenticationParameters::withParams)
57                                                     .thenReturn(withParams);
58 when(withParams.credential(any())).thenReturn(credentials);
59 when(securityContext.authenticate(request, response, credentials))
60                                     .thenReturn(AuthenticationStatus.SUCCESS);
```

Zunächst mal erstellen wir Mock-Objekte für die lokalen Variablen in der zu testenden Methode (Zeilen 49 bis 52). Dann definieren wir, wie sich die Mock-Objekte verhalten sollen.

In Zeile 54 wird z. B. festgelegt, dass ein Aufruf von *getRequest()* auf dem oben gemockten Attribut `externalContext` unser in Zeile 49 erzeugtes Mock-Objekt zurückgibt. In Zeile 56 wird die statische Methode *withParams()* der Klasse `AuthenticationParameters` gemockt, so dass sie das in Zeile 51 erzeugte Mock-Objekt zurückgibt. In Zeile 58 sehen wir die Definition des Verhaltens einer Methode, die einen Parameter erwartet. Hier könnten wir konkrete Werte einsetzen, wenn wir das möchten (und möglicherweise verschiedene Aufrufe verschieden konfigurieren möchten). Hier ist uns egal, mit welchem Parameter die Methode *credential()* aufgerufen wird, daher verwenden wir `any()`, was auf jedes Argument passt. Das kann dann so gelesen werden: „Wenn auf

dem Mock-Objekt `withParams` die Methode *credential* mit irgendeinem Argument aufgerufen wird, dann wird das Objekt `credentials` zurückgegeben".

In Zeile 59 sehen wir die Nutzung von konkreten Argumenten. Dort spezifizieren wir, was ein Methodenaufruf mit genau den dort angegebenen Objekten zurückgeben soll. Dort könnten wir auch einfach für jedes Argument `any()` einsetzen.

In Zeile 60 sehen wir, dass dort ein Rückgabewert einer anderen Klasse konfiguriert wird. Damit ist formal die Isolation unseres Prüflings nicht mehr gegeben. Das ist aber generell immer dann kein Problem, wenn Konstanten oder triviale Objekte (z. B. aus dem Datenmodell) zum Einsatz kommen. Die könnten dann ebenfalls gemockt werden, damit es wirklich gar keine Verwendung anderer Klassen gibt – aber der Aufwand ist selten gerechtfertigt. Das gilt eigentlich auch schon für die Klasse `UsernamePasswordCredential` von oben, aber wir fanden es eine gute Idee, hier mal das Mocken eines Konstruktors vorzustellen.

Der Kommentar in Zeile 48 soll verdeutlichen, dass wir hier die Gegebenheiten konfigurieren. Dazu später mehr.

Es geht weiter mit dem Aufruf des eigentlichen Prüflings:

```
61 // When
62 final String result = loginController.login();
```

Dort wird das „wenn" konfiguriert. Wenn also jemand auf dem `LoginController`-Objekt die Methode *login()* aufruft...

```
64 // Then
65 verify(externalContext, times(1)).getRequest();
66 verify(externalContext, times(1)).getResponse();
67 verifyNoMoreInteractions(externalContext);
68 assertSame(1, anonCredentials.constructed().size());
69 verify(withParams, times(1)).credential(anonCredentials.constructed().get(0));
70 verify(securityContext, times(1)).authenticate(
71                                 eq(request), eq(response), eq(credentials));
```

...dann verifizieren wir, ob verschiedene Dinge passiert sind. In Zeile 65 prüfen wir z. B., ob die Methode *getRequest()* ohne Parameter genau ein Mal auf unserem Mock-Objekt `externalContext` aufgerufen wurde. In Zeile 66 wird das analog für die Methode *getResponse()* durchgeführt. In Zeile 67 prüfen wir, ob es keine weiteren Interaktionen mit dem Objekt gegeben hat. In Zeile 68 wird überprüft, ob genau ein Objekt vom Typ `UsernamePasswordCredential` erzeugt wurde. In Zeile 66 wird verifiziert, ob die Methode *credential()* mit unserem Mock-Objekt vom Typ `UsernamePasswordCredential` genau ein Mal auf dem Objekt `withParams` aufgerufen wurde. In den Zeilen 67 und 68 prüfen wir, ob die Methode *authenticate()* mit genau den dort angegebenen Argumenten genau ein Mal auf dem Objekt `securityContext` aufgerufen wurde.

Da die Methode einen Rückgabewert hat, prüfen wir noch – wie von JUnit bekannt – ob dieser Rückgabewert der Erwartung entspricht:

```
73 assertEquals("hello", result);
```

Ihr könnt den Test ausführen, indem ihr in der Project-Ansicht das Kontextmenü der Testklasse aufruft und dort dann `Run LoginControllerTest` auswählt. Wenn der Test keine Fehler anzeigt,

könnt ihr testweise mal Dinge in der Verifizierung ändern, z. B. den erwarteten Rückgabewert. Dann seht ihr, ob der Test überhaupt Fehler findet.

Durch das Ausführen der Tests der Testklasse entsteht eine neue Run-Konfiguration und sie wird der neue Default. Das bedeutet, dass ein Klick auf Ausführen bzw. Debuggen den Test erneut ausführt. Wenn ihr später wieder auf den Server deployen möchtet, müsst ihr im Drop-Down-Feld die entsprechende Run-Konfiguration vorher wieder auf WildFly ändern.

In dem oben gelisteten Beispiel wird die Testmethode mit Hilfe von *Given-When-Then*-Abschnitten strukturiert. Diese Art der Strukturierung stammt aus dem *Behavior Driven Development* (kurz *BDD*) Prozess der agilen Software-Entwicklung. Einen kurzen Artikel zu diesem Thema findet ihr auf der Webseite von Martin Fowler: `https://martinfowler.com/bliki/GivenWhenThen.html`. Wenn euch dieses Thema – unabhängig von Java – interessiert, solltet ihr einen Blick auf *Cucumber*[a] werfen.

[a] `https://cucumber.io`

Aufgabe 3.2 Testabdeckung messen und Testklasse erweitern

Wir möchten mit unseren Tests natürlich eine möglichst hohe Abdeckung des Quellcodes erreichen. Diese können wir auf verschiedene Arten messen (lassen). Da wir davon ausgehen, dass IntelliJ eingesetzt wird, schauen wir uns erstmal den dort eingebauten Mechanismus an.

Abdeckung in IntelliJ

Öffnet dazu das Kontextmenü der Klasse `LoginControllerTest` in der Project-Ansicht. Wählt dort den Eintrag `Run 'LoginControllerTest' with Coverage` aus. Möglicherweise ist der Eintrag unterhalb von `More Run/Debug` „versteckt“. Der Test wird jetzt ganz normal ausgeführt, aber dabei wird die Abdeckung des Prüflings gemessen. Es öffnet sich daher das `Coverage`-Panel rechts (siehe Abbildung 2).

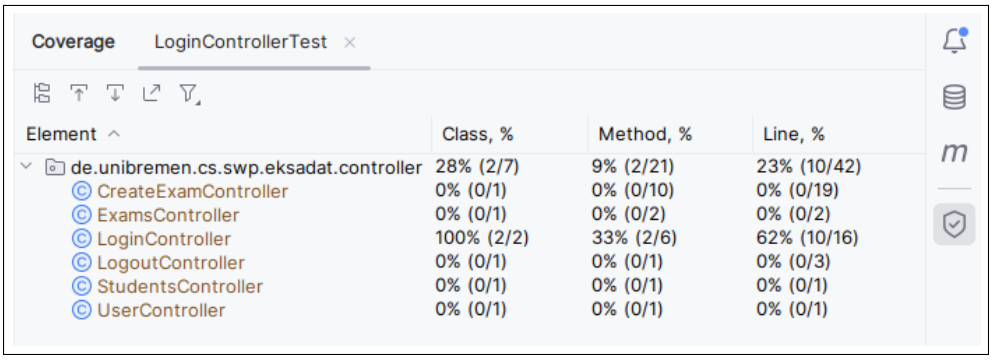

Abbildung 2: Coverage in IntelliJ

Hier sehen wir ein paar Statistiken zur Testabdeckung. Oben links befinden sich einige Schaltflächen. Interessant für uns ist hier die vierte Schaltfläche von links mit dem Tooltip *Generate Coverage Report...*. Sie dient dazu, einen HTML-Report der Abdeckung zu erzeugen, die wir dann

im Browser (oder im IntelliJ eigenen HTML Preview) anschauen können. Klickt die Schaltfläche an und übernehmt das vorgeschlagene Ausgabeverzeichnis. Setzt einen Haken bei *Open generated HTML in browser* und klickt *save*. Jetzt erstellt IntelliJ einen Report und öffnet die Hauptseite in eurem Standard-Browser. Wenn ihr im Report zur Klasse `LoginController` navigiert, dann seht ihr dort den Quellcode mit entsprechenden Markierungen (siehe Abbildung 3).

Abbildung 3: Coverage im Browser

Die grün hinterlegten Codezeilen sind durch den oder die Tests abgedeckt und die Rot hinterlegten nicht. Das passt auch zu unserem Test, denn wir haben die anderen Fälle neben `SUCCESS` ja noch nicht getestet.

In IntelliJ wird die Abdeckung übrigens im Quellcode auch direkt angezeigt – und zwar ganz links neben den Zeilennummern im Editor (siehe Abbildung 4).

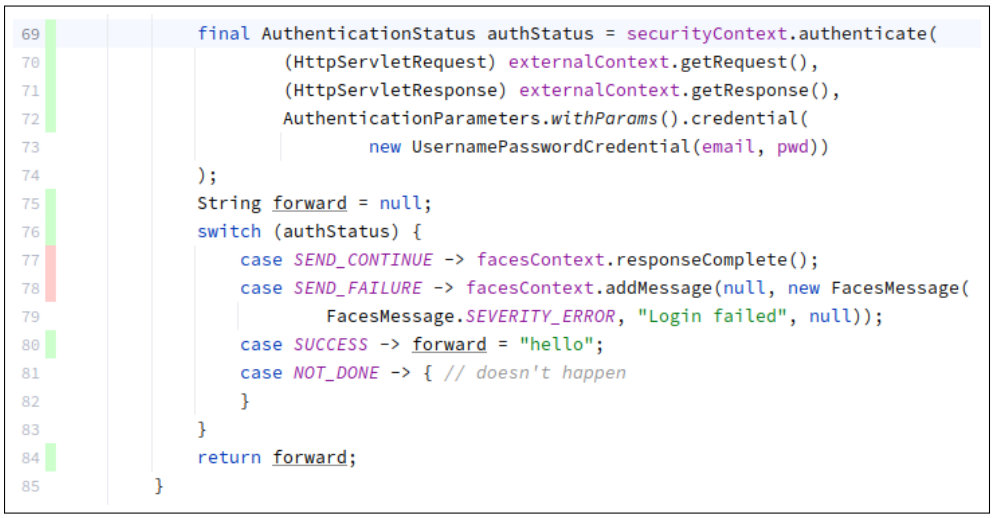

Abbildung 4: Coverage im Quellcode

Wenn ihr in der Klasse weiter nach oben scrollt, seht ihr, dass die getter- und setter-Methoden für die Attribute nicht abgedeckt sind. Es ist hier auch zu erkennen, dass der Coverage-Mechanismus scheinbar vorbildlich festgestellt hat, dass der Fall `NOT_DONE` nicht abgedeckt werden muss. Daher ist er nicht Rot markiert. Ehrlicherweise liegt das allerdings nicht an ausgeklügelter Berechnung, sondern einfach daran, dass es dort keine Anweisung gibt.

Warum meldet ein laufender WildFly, dass sich Klassen geändert haben, wenn die Tests ausgeführt werden?
Das liegt daran, dass die class-Dateien durch das Abdeckungsmessframework verändert (instrumentiert[a]) werden. Sie werden mit Bytecode ergänzt, der es ermöglicht, festzustellen, ob die Anweisungen tatsächlich ausgeführt wurden.

[a] `https://de.wikipedia.org/wiki/Instrumentierung_(Softwareentwicklung)`

Abdeckung unabhängig von der IDE

Wir können die Abdeckung auch unabhängig von der IDE messen (lassen). JaCoCo[8] ist eine weit verbreitete Bibliothek zur Abdeckungsmessung in der Java-Welt. Passenderweise gibt es für JaCoCo ein Maven-Plugin, mit dessen Einbindung die Erstellung eines Testreports automatisiert werden kann (z. B. durch den Aufruf von `mvn clean test`). Fügt das Plugin wie folgt zur `pom.xml` hinzu:

```
1  <plugin>
2      <groupId>org.jacoco</groupId>
3      <artifactId>jacoco-maven-plugin</artifactId>
4      <version>${maven-jacoco-plugin.version}</version>
5      <executions>
6          <execution>
7              <goals>
8                  <goal>prepare-agent</goal>
9              </goals>
10         </execution>
11         <execution>
12             <id>report</id>
13             <phase>test</phase>
14             <goals>
15                 <goal>report</goal>
16             </goals>
17         </execution>
18     </executions>
19 </plugin>
```

Damit wird beim Aufruf des Maven `test`-Goals JaCoCo automatisch mit ausgeführt. Die Ergebnisse sind anschließend im Verzeichnis `target/site/jacoco` zu finden. Um einen schnellen Überblick über die Testabdeckung zu erhalten, könnt ihr die Datei `index.html` darin öffnen. Auch dort könnt ihr bis zum markierten Quellcode navigieren. Es gibt dann allerdings neben der Anweisungsabdeckung auch Angaben zur Zweigabdeckung (siehe Abbildung 5).

JaCoCo und Lombok

Bei der Messung der Anweisungs- und Zweigabdeckung des Prüflings berücksichtigt JaCoCo (wie oben erwähnt) auch die von Lombok generierten getter- und setter-Methoden. Es ist aber

[8] `https://www.jacoco.org/jacoco/index.html`

Abbildung 5: Coverage im JaCoCo-Report

eigentlich ziemlich sinnlos, automatisch generierte Dinge zu testen – wir nehmen an, dass der generierte Code hinreichend während der Entwicklung von Lombok getestet wird.

Dieses Problem betrifft aber nicht nur Lombok, sondern alle Bibliotheken/Frameworks, die Programmcode zur Übersetzungszeit generieren. Daher werden in JaCoCo alle Elemente, die mit der Annotation `Generated`[9] annotiert sind, von der Analyse ausgeschlossen. Ihr müsst also Lombok anweisen, die generierten Programmteile entsprechend zu *markieren*. Dies lässt sich wieder mit Hilfe der Konfigurationsdatei `lombok.config` realisieren. Fügt dort folgende Zeile hinzu:

```
lombok.addLombokGeneratedAnnotation = true
```

Nun sollte der von Lombok generierte Programmcode von JaCoCo nicht mehr berücksichtigt werden – achtet darauf, die *veralteten* Testreports durch einen Aufruf des Maven-Goals `clean` (z. B. mit `mvn clean test`) zu löschen, bevor ihr die Tests erneut ausführt.

Aufgabe

Erstellt jetzt in der Testklasse weitere isolierte Tests (d. h. für alle anderen Klassen als `LoginController` müssen Mocks verwendet werden), die eine hunderprozentige Zweigabdeckung für die Klasse erreichen, wobei generierter Code dabei nicht beachtet wird.

Wir haben den ursprünglich ausgelieferten Test so gestaltet, dass er exakt den gegebenen Quellcode-Ausschnitt überprüft (durch die *verify()*-Methoden). Das haben wir getan, um euch die Möglichkeiten von Mockito aufzuzeigen. In der Praxis ist das natürlich nicht sinnvoll, denn dann könnte der Quelltext ja nahezu nicht mehr verändert werden – selbst wenn die Änderung keine Verhaltensänderung bewirkt. Es ist daher für einen Test zu überlegen, welche Methodenaufrufe auf gemockten Objekten wirklich sinnvoll zu verifizieren sind. Dazu gehört sicher, dass im Falle eines gescheiterten Logins eine Nachricht erzeugt und angezeigt wird. Oder z. B. auch, ob eine Controller-Methode ggf. eine Methode aus der entsprechenden `Service`-Klasse aufruft. Dadurch kann ein Stück weit die Einhaltung der Architekturrichtlinien automatisiert getestet werden.

[9] `https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/annotation/processing/Generated.html`

Optional: Konstruktor-Argumente prüfen

Überprüft in euren Testfällen, ob Konstruktor-Aufruf der `FacesMessage` mit den korrekten Argumenten durchgeführt wurde.

Nehmen wir an, es wird in der zu testenden Methode irgendwo ein Konstruktor einer anderen Klasse `Foo` aufgerufen, der zuerst einen Parameter vom Typ `int` und dann einen Parameter vom Typ `String` erwartet. Wenn ihr jetzt sicherstellen möchtet, dass der Konstruktor in eurem Testfall mit den Argumenten 42 und `"Hello"` aufgerufen wird, dann könnt ihr das folgendermaßen realisieren:

```
1 MockedConstruction<Foo> fooMockedConstruction = mockConstruction(Foo.class,
2             (mock, context) -> {
3               assertSame(42, context.arguments().get(0));
4               assertEquals("Hello", context.arguments().get(1));
5             })
```

Hier wird ein Lambda-Ausdruck an Mockito übergeben, der ausgeführt wird, wenn die Klasse `Foo` instanziiert wird. Dieser überprüft mithilfe des `context`-Objektes und üblichen asserts, ob die Argumente, die der Konstruktor erhalten hat, mit den gewünschten Werten übereinstimmen.

Aufgabe 3.3 Service Test

Erstellt weitere Tests, die die Methode *save()* der Klasse `ExamService` sinnvoll testen und dabei eine hundertprozentige Zweig-Abdeckung erreichen.

Hier stehen wir vor dem Problem, dass Mockito kein vollständiges CDI-System ist (und auch nicht sein möchte) und daher nicht in der Lage ist, das in der Superklasse deklarierte Repository direkt in den Prüfling zu injizieren. Wenn wir uns die Superklasse `AbstractBaseService` anschauen, dann sehen wir, dass dort ein konkretes Repository über generische Parameter per CDI initialisiert wird. Das kann Mockito als Testframework nicht leisten. Daher müssen wir uns darum selbst kümmern.

Da es keine setter-Methode für das Repository gibt und auch keine Konstruktor-Parameter, bleibt uns nur, das Attribut manuell über Reflection[10] zu setzen. Dazu nutzen wir eine Methode, die vor jedem Test aufgerufen wird:

```
@Mock
private ExamRepository examRepository;

@InjectMocks
private ExamService examService;

@BeforeEach
private void setRepositoryMock() throws Exception {
  final Field repository =
           examService.getClass().getSuperclass().getDeclaredField("repository");
  repository.set(examService, examRepository);
}
```

[10] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/package-summary.html`