Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

My Extraordinary Thesis

Bachelorarbeit

vorgelegt von

Jon Snow

03. Dezember 2019

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: tbd

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 03. Dezember 2019

_____

Jon Snow

Zusammenfassung

Fassen Sie hier die Fragestellung, Motivation und Ergebnisse Ihrer Arbeit in wenigen Worten zusammen.

Die Zusammenfassung sollte den Umfang einer Seite nicht überschreiten.

Danksagung

Im Falle, dass Sie Ihrer Arbeit eine Danksagung für Ihre Unterstützer (Familie, Freunde, Betreuer) hinzufügen möchten, können Sie diese hier platzieren.

Dieser Part ist optional und kann im Quelltext auskommentiert werden.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Hier kommt die Einleitung hin.

2 Theoretische Grundlagen

Kurze Erklärung.

2.1 Die testgetriebene Entwicklung (TDD)

Hier kommt der Theorieteil zu Test-driven Development hin.

2.2 Fußballerische Grundlagen und Konzepte

Hier kommt der Theorieteil zu den fußballerischen Grundlagen hin.

3 Die testgetriebene Entwicklung des Spielzeitenplaners

Nachdem die theoretischen Grundlagen geklärt und erläutert worden sind, kann nun im folgenden Kapitel näher auf die testgetriebene Entwicklung des Spielzeitenplaners eingegangen werden. Dafür werden zunächst sein grundsätzlicher Aufbau und seine Funktionsweise skizziert, ehe im Anschluss das konkrete Testing in den unterschiedlichen Schichten der Architektur dokumentiert und verdeutlicht wird.

3.1 Aufbau und Funktionsweise des Spielzeitenplaners

Der Spielzeiten-Planer ist grundsätzlich in vier verschiedene Bereiche eingeteilt: Team, Recap, Spielzeiten planen und Einstellungen. Diese Bereiche und die damit verbundenen Grundfunktionen des Spielzeiten-Planers sollen im Folgenden kurz beschrieben und erläutert werden.
Als Erstes ist der Team-Bereich zu nennen, der sich im Wesentlichen aus zwei Teilen zusammensetzt: der Team-Seite und der Spieler-Seite. Auf Ersterer lässt sich ein Teamname festlegen und speichern oder ändern. Außerdem wird eine Liste mit allen Spielern im Team und ihren Daten (Name, Position, Trikotnummer, etc.) angezeigt. Für jeden Spieler gibt es die Möglichkeit, diesen entweder zu löschen oder zu bearbeiten. Mit einem Klick auf den Löschen-Button wird der entsprechende Spieler gelöscht, durch den Klick auf den Bearbeiten-Button hingegen gelangt man zur Spieler-Seite.

Diese enthält sowohl die Spieler-Daten wie auch eine Anzeige der Spieler-Scores. Hier können Vor-, Nachname, Position und Trikotnummer des ausgewählten Spielers geändert werden. Wählt man auf der Team-Seite den Button zum Erstellen eines neuen Spielers, so wird die Spieler-Seite mit einem leeren Formular aufgerufen, sodass ein neuer Spieler erstellt und anschließend gespeichert werden kann.
Der Bereich Einstellungen enthält – wie der Name bereits suggeriert – einige grundsätzliche Einstellungen, die insbesondere für den Bereich zum Planen der Spielzeiten von Bedeutung sind. Unter anderem besteht hier die Möglichkeit, eine eigene Formation zu erstellen. Dafür sind die Angabe eines Namens – zum Beispiel *4-2-3-1* oder *4-3-3* – sowie die Bezeichnungen der einzelnen Positionen notwendig.
Über dem Abschnitt zur Formation befindet sich der Kriterien-Abschnitt. Hier können Kriterien erstellt, bearbeitet und gelöscht werden. Diese sind von zentraler Bedeutung bei der Bewertung der Spieler im Recap-Bereich. Für die Erstellung eines Kriteriums wird ein Name bzw. eine Bezeichnung, eine Abkürzung (ein bis zwei Buchstaben) und eine Gewichtung benötigt. Die Summe aller Gewichte sollte stets eins ergeben, ein einzelnes Gewicht im Bereich zwischen null und eins liegen. Die wohl gebräuchlichsten Kriterien sind zum Beispiel die Trainingsbeteiligung und die Leistung.
Schließlich gibt es noch die Scores-Einstellungen, die ganz oben auf der Seite zu finden sind. In diesem Bereich lässt sich der Zeitraum festlegen, auf dessen Grundlage die Scores für die einzelnen Kriterien berechnet werden. Für das Kriterium der Trainingsbeteiligung gibt es nochmal besondere Einstellungen: Zum einen kann zwischen einer kurzfristigen und langfristigen Trainingsbeteiligung unterschieden, zum anderen können spezifische Gewichte für die Kurz- und Langfrist festgelegt werden.
Folgendes Beispiel soll zur Verdeutlichung des Sachverhaltes herangezogen werden: Ein Trainerteam entscheidet sich dazu, dass die Trainingsbeteiligung 50 Prozent des Gesamtscores ausmachen soll. Innerhalb der Trainingsbeteiligung wird dann nochmals festgelegt, dass die letzten drei Wochen für die Kurzfrist herangezogen werden sollen und die letzten acht Wochen für die Langfrist. Da das Trainerteam etwas mehr Wert auf eine langfristige Teilnahme am Training legt, werden die Gewichte auf 0.4 für die Kurzfrist und 0.6 für die Langfrist festgelegt. Somit berechnet sich der Score für dieses Kriterium zu 60 Prozent aus der langfristigen Trainingsbeteiligung und zu 40 Prozent aus der Kurzfrist. So kann ein Spieler, der grundsätzlich immer am Training teilnimmt, ein kurzfristiges Fehlen aufgrund von Krankheit oder schulischer Verpflichtungen durch einen hohen Score in der Langfrist korrigieren.
Der dritte große Bereich der Anwendung ist das Recap. Im Wesentlichen geschieht hier Folgendes: Nach jedem Training wird eine Bewertung jedes Spielers zu jedem Kriterium vorgenommen. Die Bewertungen werden gespeichert und zur Ermittlung der Scores für jedes Kriterium sowie des Gesamtscores herangezogen. Letzterer wiederum ist von wesentlicher Relevanz bei der Planung der Spielzeiten – also der Spielminuten – für das kommende Spiel. Um zur Recap-Seite zu gelangen, werden in einem ersten Schritt all diejenigen Spieler ausgewählt, die am Training teilgenommen haben. Diese Information wird dann auf der eigentlichen Bewertungsseite genutzt, um eine Vorsortierung der Spieler vorzunehmen.
Grundsätzlich ist die Recap-Seite nach den vorhandenen Kriterien gegliedert, das bedeutet, dass für jedes Kriterium eine Liste mit Spielern angezeigt wird, für die dann eine Bewertung

abgegeben wird. Die Bewertung erfolgt auf einer Skala von eins bis fünf, wobei eine Drei den Durchschnitt bildet, eine Eins eine deutlich unterdurchschnittliche Bewertung darstellt und eine Fünf die bestmögliche Bewertung ist. Dementsprechend ist eine Vier als tendenziell überdurchschnittlich und eine Zwei als tendenziell unterdurchschnittlich zu betrachten.
Da der jeweilige Wert serverseitig als Double abgebildet wird, ist es den Nutzenden möglich, weitere Abstufungen vorzunehmen, beispielsweise eine 2.5 oder 4.5 zu vergeben. Für eine schnelle und effiziente Bewertung ist es jedoch ratsam, bei den Bewertungen eins, zwei, drei, vier und fünf zu bleiben. Für einen Spieler, der beim Training nicht anwesend ist, wird standardmäßig eine 0.0 vergeben. Die Null-Bewertungen werden dann serverseitig herausgefiltert und nicht gespeichert, da ein häufiges Fehlen sonst nicht nur den Score der Trainingsbeteiligung, sondern auch alle anderen Scores verringern würde, was einer gleich mehrfachen Abwertung gleichkäme.
Schließlich ist dann noch der Bereich der Spielzeitenplanung zu nennen. Hier können die Nutzenden vor einem Spiel planen, wie viele Minuten Spielzeit jeder einzelne Spieler basierend auf dem Gesamtscore erhalten soll. Die Spielzeitenplanung gestaltet sich als ein mehrstufiges Verfahren, durch das die Nutzenden vom Spielzeitenplaner geleitet werden.
In einem ersten Schritt werden zunächst alle Spieler ausgewählt, die für das kommende Spiel zur Verfügung stehen, die also nicht krank oder verletzt sind oder aufgrund von privaten Terminen und anderen Gründen abgesagt haben. Aus den verfügbaren Spielern wird dann in einem zweiten Schritt ermittelt, welche Spieler es in den Kader geschafft haben und welche nicht. Die vorgeschlagene Aufteilung kann dabei übernommen oder aber manuell durch die Nutzenden überarbeitet werden, sodass das Trainerteam die Kontrolle über die Spielzeitenplanung behält.
Nachdem der Kader feststeht und das entsprechende Formular durch die Benutzenden abgeschickt worden ist, wird in einem dritten Schritt aus dem Kader die Startelf bestimmt. Auch bei diesem Schritt können die Nutzenden Einfluss nehmen, indem Positionen getauscht oder Spieler von der Bank in die Startelf gesetzt werden. Ist die Startelf ermittelt, kommt es zum letzten Schritt der Spielzeitenplanung: dem Eintragen der Wechsel. Dieser letzte Planungsschritt ist essenziell für die Bestimmung der Spielzeiten, denn mit dem Feststehen der Wechsel bzw. der Wechsel-Zeitpunkte steht ebenfalls fest, welcher Spieler wie viele Minuten auf dem Platz steht.
Wie bei den vorherigen Planungsschritten besitzen die Nutzenden auch hier die volle Kontrolle: sowohl Einwechsel- wie Auswechselspieler aber auch die konkrete Spielminute kann bestimmt werden. Die voraussichtliche Anzahl der Spielminuten für jeden einzelnen Spieler wird auf Basis der gespeicherten Wechsel berechnet und auf der Seite angezeigt. Außerdem wird vom Spielzeitenplaner eine erwartete Spielzeit berechnet. Diese errechnet sich maßgeblich aufgrund des Gesamtscores eines Spielers und stellt diejenige Spielzeit dar, die basierend auf den Bewertungen des entsprechenden Spielers als fair erachtet wird.
Nun können Nutzende so lange wie nötig Anpassungen vornehmen – das heißt neue Wechsel eintragen, Wechsel löschen oder die Wechselzeitpunkte anpassen – bis die voraussichtliche Anzahl der Spielminuten eines jeden Spielers ungefähr mit der Anzahl der erwarteten Spielminuten übereinstimmt. Ein weiteres Mal ist es dem Trainerteam selbst überlassen zu entscheiden, ob erwartete und voraussichtliche Spielzeit beispielsweise bis auf fünf, zehn oder fünfzehn Minuten übereinstimmen müssen, dennoch sollten die beiden Kennzahlen so

eng wie möglich beieinander liegen, da große Abweichungen die Sinnhaftigkeit die Spielzeitenplanung infrage stellen.
Ist schließlich eine faire Aufteilung der Spielzeiten unter allen beteiligten Akteuren gefunden, so kann die Spielzeitenplanung als erfolgreich abgeschlossen betrachtet werden und der Einsatz der Startelf sowie die Wechsel wie geplant durchgeführt werden.

3.2 Testing der Web-Pages

Nachdem Aufbau und Funktionsweise des Spielzeitenplaners beschrieben und erklärt worden sind, soll nun im Detail auf die testgetriebene Entwicklung der Anwendung eingegangen werden. Im Folgenden wird sich dabei zunächst auf das Testen des Web-Interfaces konzentriert.
Das Testing des Web-Interfaces ist grundsätzlich zweigeteilt: Zum einen werden die konkret ausgelieferten Webseiten mit ihren Inhalten und HTML-Elementen überprüft, zum anderen gibt es spezielle Testklassen für die Controller, mithilfe derer eine grundsätzliche Funktionsüberprüfung der Web-Steuereinheiten erfolgt. Die Testklassen für Ersteres sind im Verzeichnis `de/bathesis/spielzeitenplaner/templates` zu finden. Eine solche Testklasse ist grundsätzlich folgendermaßen aufgebaut:

```
@WebMvcTest(Controller.class)
class PageTest {
    @Autowired
    MockMvc mvc;
    ...
    Document page;
    ...
    @BeforeEach
    void setUpPage() throws Exception {
        ...
        page = RequestHelper
                  .performGetAndParseWithJSoup(mvc, "/path");
    }
    ...
}
```

Durch die Verwendung der `WebMvcTest`-Annotation wird ein spezieller Testkontext initialisiert, der sich nur auf das Laden und Konfigurieren von Komponenten der Web-Schicht konzentriert **(QUELLE !!!)**. Darüber hinaus wird in Klammern notiert, welche spezifische Controller-Klasse für den Test benötigt wird, sodass nur diese für den Test geladen und bereitgestellt wird. Überlicherweise enthält jede WebPage-Testklasse eine mit `BeforeEach` annotierte `setUp`-Methode, die vor jedem Test ausgeführt wird. Da das Vorgehen vor jedem Test gleich ist, bietet sich die Extraktion dieser Logik an, um die einzelnen Test

übersichtlicher und wartbarer zu gestalten.
Das Ziel der `setUp`-Methode ist es, mithilfe des MockMvc eine Anfrage über einen bestimmten Pfad zu simulieren, das Ergebnis mit Jsoup zu parsen (siehe `RequestHelper.java`) und schließlich in einer Instanzvariable – hier `page` genannt – zu speichern. Auf diese Weise kann ein Abbild derjenigen Seite erzeugt werden, die im Browser zurückgegeben wird. Diese kann dann detaillierten Testungen unterzogen werden.
Ein wichtiger Bestandteil dabei bildet die Java-Bibliothek Jsoup. Sie wurde für das Arbeiten mit HTML entwickelt und ermöglicht daher sowohl das Parsen, wie auch Extrahieren, Manipulieren oder Korrigieren von HTML-Dokumenten **(QUELLE !!!)**. Für die dieser Arbeit zugrunde liegenden Testungen werden vor allem die erstgenannten Funktionen – also das Parsen und Extrahieren von HTML bzw. HTML-Schnipseln – benötigt.
Wie Jsoups `parse`-Funktion in diesem Projekt verwendet wurde, wurde bereits zuvor erklärt, das Extrahieren von HTML-Schnipseln lässt sich wie folgt realisieren: Mithilfe der `select`-Funktion wird aus einem HTML-Dokument oder einem HTML-Element ein HTML-Schnipsel herausgelöst, das den Anforderungen einer `CSS-Query` entspricht, die der Funktion als Parameter übergeben worden ist.
Über die Komplexität der `CSS-Query` kann gesteuert werden, wie detailliert die Struktur der HTML-Seite getestet werden soll. So kann mit `‘‘h2’’` zum Beispiel einfach eine Überschrift zweiter Ebene herausgefiltert werden, mit `‘‘.card .card-body h2.card-title’’` hingegen präziser nach einer `H2` gesucht werden, die die Klasse `card-title` besitzt und sich innerhalb des Bodies einer `Card` befindet.
Das Testing jeder einzelnen Webseite erfolgt im Wesentlichen nach ein und demselben Konzept, das im Folgenden geschildert werden soll. Zunächst werden die sogenannten Essentials – oder auch Basics – getestet, es wird also überprüft, ob die Seite die korrekte Überschrift besitzt sowie ob grundlegende Elemente – wie die Navigationsleiste und der Footer – angezeigt werden. Im Anschluss erfolgt eine detaillierte Überprüfung der einzelnen Bereiche der jeweiligen Seite und ihrer Elemente, also beispielsweise, ob der Bereich Teamname auf der Teamseite korrekt angezeigt wird oder ob das Formular zum Ändern des Teamnamens korrekt angezeigt wird (siehe `TeamPageTest.java`).
Abschließend wird getestet, ob entsprechende Daten, die durch den Controller bzw. das Model bereitgestellt werden, wie beabsichtigt mithilfe von Thymeleaf in die Seite gerendert werden. Durch diesen strukturierten Ansatz wird gewährleistet, dass alle wesentlichen Elemente und Funktionen der Webseite umfassend getestet werden.

Mit Beginn der Arbeiten an diesem Projekt wurden zunächst sämtliche HTML-Prototypen bzw. die ihnen zugrunde liegenden HTML-Templates testgetrieben entwickelt. Ein Beispiel dafür, auf das sich im Folgenden bezogen werden soll, ist die sogenannte `PlayerPage`, der das HTML-Template `player.html` als Basis dient, und die zugehörige Testklasse `PlayerPageTest.java`.
Ein einfacher Test, durch den die Anwesenheit der korrekten H1-Überschrift erzwungen werden kann, ist `test_01`:

```
@Test
```

```
...
void test_01() {
    String expectedTitle = "Spieler bearbeiten/hinzufügen";
    String pageTitle = RequestHelper
              .extractTextFrom(playerPage, "h1");
    assertThat(pageTitle).isEqualTo(expectedTitle);
}
```

Hier wird mittels der Utilities-Klasse `RequestHelper.java` der Text des H1-Tags der `PlayerPage` extrahiert. Dann wird überprüft, ob dieser mit dem erwarteten Text übereinstimmt. Ist dies nicht der Fall, schlägt der Test fehl. Wenn auf der Seite überhaupt kein solches Element vorhanden ist, schlägt der Test ebenfalls fehl, denn dann ist das Ergebnis des Parsens durch Jsoup schlichtweg ein leerer String. Auf diese Weise kann also gewährleistet werden, dass auf einer Seite die korrekte Überschrift angezeigt wird.
Für die Navigationsleiste wurde auf der `PlayerPage` – wie auch auf allen anderen Seiten – getestet, ob diese vorhanden ist:

```
@Test
...
void test_02() {
    Elements navbar = RequestHelper.extractFrom(playerPage, "nav");
    assertThat(navbar).isNotEmpty();
}
```

Dafür wird das `nav`-Element der Seite extrahiert und durch eine geeignete Assertion sichergestellt, dass diese nicht leer ist. Analog zur Navigationsleiste kann ebenfalls mit dem Footer verfahren werden. Die eigentliche Überprüfung, ob die `Essentials` den Anforderungen entsprechen, ist ausgelagert und in der `FragmentsTest.java` unter `src/test/java/de/bathesis/spielzeitenplaner/templates/fragments` zu finden.
Dort wird im Detail getestet, ob die einzelnen Elemente korrekt strukturiert sind, sie den gewünschten Text enthalten und die erwarteten Funktionen unterstützen – im Falle der Navigationsleiste beispielsweise, ob die Links zu den Unterseiten `Team`, `Recap`, `Spielzeiten planen` sowie `Einstellungen` funktionieren. Das entsprechende Fragment – also der wiederkehrende HTML-Schnipsel einer Webseite, der ausgelagert worden ist – ist in der `basics.html` unter `src/main/resources/templates/fragments` gespeichert. Dort wird es mittels des `th:fragment`-Tags als Solches gekennzeichnet und benannt.
Ist dies geschehen, so kann es im Folgenden dann wiederverwendet werden, indem es mithilfe der Nutzung des `th-replace`-Tags im HTML-Template und Thymeleaf in die entsprechende Seite hineingerendert wird. Ein konkretes Beispiel für die Verwendung eines `Thymeleaf-Fragments` in diesem Projekt ist der Footer:

```
<div th:fragment="footer">
```

```
        <footer class="footer fixed-bottom">
            <p>
                &copy; 2024 SpielzeitenPlaner. Alle Rechte vorbehalten.
            </p>
        </footer>
    </div>
```

Er ist mit dem `th:fragment`-Tag versehen und als `‘‘footer’’`, benannt. Innerhalb der `welcome.html` kann er dann einfach mithilfe des einzeiligen Codeschnipsels `<div th:replace=‘‘~{fragments/basics :: footer}’’></div>` eingefügt werden.
Durch diese Herangehensweise müssen die Funktionalitäten der `Essentials` nicht auf jeder Seite explizit getestet werden – bei zehn verschiedenen Seiten wären das immerhin 30 Tests, die aber stets nur das Gleiche testen würden – sondern lediglich das Vorhandensein gewisser Elemente. Wenn sich nun eine Beschriftung ändert, die Struktur angepasst werden soll oder ein neuer Bereich – zum Beispiel `Statistik` – hinzukommt, müssen die entsprechenden Tests nur an einer Stelle angepasst werden, die Testklassen für die einzelnen Webseiten bleiben unberührt, da hier – wie zuvor bereits beschrieben – nur das Vorhandensein geprüft wird.
Navigationsleiste und Footer sind im Rahmen des Spielzeitenplaners als feste Bestandteile jeder Seite eingeplant, um Letzteren eine Rahmung zu geben und Nutzenden die Navigation durch die einzelnen Bereiche der Anwendung zu erleichtern. Für den Fall, dass Navigationsleiste oder Footer jedoch komplett entfernt werden sollen, kann über die Verwendung des `Thymeleaf Layout Dialect` nachgedacht werden. Dieser ermöglicht es, Layout-Templates zu erstellen und zu definieren, die dann wiederum von anderen Templates verwendet werden können **(QUELLE !!!)**.
So kann die grundsätzliche Struktur einer Seite von ihrem konkreten Inhalt getrennt werden, was die Modularisierbarkeit fördert und die Wiederverwendbarkeit erhöht. Für das konkrete Testing bedeutet dies, dass ein Layout-Template ein Mal gesondert getestet wird, das Testing der einzelnen Seiten sich vollkommen auf den Inhalt konzentrieren kann.
Doch wie bereits zuvor beschrieben ist nicht nur das Testen der `Essentials` ein wichtiger Bestandteil der `WebPages`-Tests, sondern auch die Überprüfung der einzelnen Bereiche einer Seite und ihrer Elemente. Den Kern der Spieler-Seite bilden die Bereiche `Spieler-Daten` und `Spieler-Scores`. Durch das Entwickeln dreier Tests, die stets einen anderen Aspekt überprüfen, kann ein solcher Bereich im Produktivcode erzwungen und Schritt für Schritt geformt werden, bis er schließlich seine gegenwärtige Form erreicht hat.
In einem ersten Schritt kann ein Test geschrieben werden, der die grundsätzliche Struktur des Bereichs festlegt:

```
@Test
...
void test_04() {
    String expectedCardTitle = "Spieler-Daten";
    List<String> expectedAttributes = new ArrayList<>(List.of(
```

```
            "Vorname", "Nachname", "Trikotnummer", "Position"
        ));

        String cardTitle = RequestHelper.extractTextFrom(
            playerPage, ".card.player-data .card-body .card-title"
        );
        String playerInfo = RequestHelper.extractTextFrom(
            playerPage, ".card.player-data .card-body .player-info"
        );

        assertThat(cardTitle).isEqualTo(expectedCardTitle);
        assertThat(playerInfo).contains(expectedAttributes);
    }
```

Um diesen Test bestehen zu lassen, ist die Etablierung der Grundstruktur in der `player.html` erforderlich, wie dem Commit `5e52549` im Detail entnommen werden kann.
In einem zweiten Schritt kann dann die Anwesenheit des Spieler-Formulars erzwungen werden, das zum einen dazu dient, die Daten eines Spielers anzuzeigen, zum anderen aber auch das Ändern bereits gespeicherter Informationen unterstützt. Wie `test_05` der `PlayerPageTest.java` zu entnehmen ist, wird dort geprüft, ob es ein Formular mit der ID `playerForm` gibt sowie ob dieses die für die Verarbeitung der Daten notwendigen Elemente – wie Input-, Label-Felder und einen Button – besitzt. Außerdem wird an dieser Stelle ebenfalls gefordert, dass das Formular die `Post`-Methode verwenden und die Anfrage über `/team/savePlayer` verschickt werden soll sowie einen Button vom Typen `submit` enthalten muss, wie den folgenden Zeilen zu entnehmen ist:

```
Elements playerForm = RequestHelper.extractFrom(playerPage,
    "form#playerForm[method=\"post\"][action=\"/team/savePlayer\"]"
);
String buttonLabel = RequestHelper.extractTextFrom(playerForm,
    "button[type=\"submit\"]"
);
```

Durch solche spezifischen `CSS-Queries` können präzise Anforderungen an das Formular gestellt werden und sichergestellt werden, dass die notwendigen Funktionalitäten korrekt implementiert werden. Im hier vorliegenden Fall der Änderung von Spielerdaten kann bzw. muss parallel im Controller-Testing eine entsprechende Route und Methodenunterstützung implementiert werden (siehe Kapitel 3.2).
Nachdem nun die grundlegende Struktur des Spielerdaten-Bereichs etabliert und das Spieler-Formular vorhanden ist, kann abschließend in einem dritten Schritt die korrekte Anzeige der konkreten Spieler-Daten überprüft werden:

```
@Test
```

```
...
void test_07() {
    List<String> expectedValues = new ArrayList<>(List.of(
        Integer.toString(player.getId()),
        player.getFirstName(), player.getLastName(),
        player.getPosition(),
        Integer.toString(player.getJerseyNumber())
    ));

    List<String> values = RequestHelper.extractFrom(
        playerPage, "form#playerForm input"
    ).eachAttr("value");

    assertThat(values)
        .containsExactlyInAnyOrderElementsOf(expectedValues);
}
```

Die `expectedValues` entsprechen den Attributen des `player`, der als Instanzvariable in der Testklasse definiert ist. Mithilfe des `RequestHelpers` und Jsoups `eachAttr`-Methode können die tatsächlich angezeigten Werte der Input-Felder in einer Liste gespeichert und anschließend überprüft werden. Damit der Test jedoch ordnungsgemäß funktioniert, muss der mit `@MockBean` annotierte `playerService` noch konfiguriert werden. Mithilfe von `when(playerService.loadPlayer(player.getId())).thenReturn(player)` geschieht dies innerhalb der `setUp`-Methode entsprechend.
Um den Test nun bestehen zu lassen und die gewünschte Funktionalität zu implementieren, muss Folgendes geschehen: Die Spieler-Daten, die vom entsprechenden Service an den Controller weitergegeben und durch Letzteren im Model bereitgestellt werden, müssen mit dem jeweiligen Input-Feld verknüpft werden. Die Existenz jener Input-Felder wurde bereits im vorherigen Test gefordert, nicht aber ihr spezifischer Inhalt. Mithilfe von Thymeleaf lässt sich die entsprechenden Daten komfortabel in das jeweilige Template bzw. die jeweilige Seite hineinrendern:

```
<form id="playerForm" ... th:object="${playerForm}">
    ...
    <input type="text" ... th:field="*{firstName}" ...>
    ...
    <input type="text" ... th:field="*{lastName}" ...>
    ...
</form>
```

Hier wurde `th:object` verwendet, um das Formular mit dem Formular-Objekt `playerForm` aus dem Model zu paaren. Thymeleafs `th:field` bindet außerdem jedes einzelne Input-Feld

an das entsprechende Attribut, also `firstName`, `lastName`, `position` und `jerseyNumber`. Dies ist nicht nur für eine spätere Validierung und der damit verbundenen Ausgabe der Formular-Fehler von Vorteil und notwendig, sondern sorgt darüber hinaus dafür, dass die Daten des aktuellen Spielers auf der `PlayerPage` angezeigt werden.
Zusammengefasst lässt sich noch einmal sagen, dass sich durch gezieltes, umfangreiches Testing unterschiedlicher Aspekte die wesentlichen Funktionalitäten und Bausteine einer Webseite testgetrieben entwickeln lassen: Von der grundlegenden Strukur einer Seite bis hin zum Aufbau ihrer spezifischen Bereiche, die ihnen innewohnenden Elemente zur Datenerfassung, Interaktion und Kommunikation – also zum Beispiel Formulare – sowie die Anzeige konkreter und gespeicherter Daten und Inhalte.
Diese grundlegenden Testprinzipien lassen sich auf die Entwicklung jeder einzelnen Webseite übertragen und an ihre spezifischen Anforderungen anpassen, beispielsweise bei der Spielerbewertung auf der `RecapPage` (siehe `RecapPageTest.java`) oder der Anzeige und Bestätigung des Kaders in der Spielzeitenplanung (siehe `KaderPageTest.java`).

3.3 Testing der Controller

Eng verbunden mit mit dem Testing der `WebPages` ist auch die Überprüfung der Web-Steuereinheiten, also der entsprechenden Controller. Wie bereits im vorangegangenen Kapitel erwähnt, können konkrete Inhalte nur in die Seite eingefügt werden, wenn diese im Model vorhanden sind. Dies stellt unter anderem eine Aufgabe des Controllers dar. Doch neben der Datenaufbereitung und -bereitstellung ist er außerdem auch für die Verarbeitung von Benutzeranfragen, das Verwalten der Anwendungslogik, die Fehlerbehandlung und das grundlegende Routing zuständig. Alle soeben genannten Aufgaben sollen in diesem Kapitel unter dem Aspekt der testgetriebenen Entwicklung des Spielzeitenplaners eingehend beleuchtet werden.
Begonnen werden soll mit dem letzten Punkt – dem grundlegenden Routing, das den Startpunkt sämtlicher Controller-Tests darstellt. Denn wie bereits in Kapitel 3.1 festgehalten, sind sämtliche Testungen der Elemente und Strukturen einer ausgelieferten Webseite unter Gebrauch eines `MockMvc`-Objektes nur möglich, wenn zuvor ein entsprechendes Routing etabliert und ein geeignetes Request-Mapping stattgefunden hat.
Der wohl simpelste Test zur Überprüfung einer Route kann im Commit 3aec5fe betrachtet werden. Alles, was zum Bestehen des Tests zur Erreichbarkeit der Team-Seite benötigt wird, ist eine mit `@Controller` annotierte Klasse und eine mit `@GetMapping(“/team”)` beschriftete Handler-Methode, die wiederum den Namen eines Templates zurückgibt – in diesem Fall die `team.html`, die im Verzeichnis `src/main/ressources/templates` existieren muss.
Im weiteren Verlauf der Entwicklung des Projektes ist die `team()`-Handler-Methode dann in einen eigens für diesen Bereich angelegten `TeamController` ausgelagert worden. Des Weiteren ist mit der Einführung des `TeamService` das gewünschte Verhalten – hier die Rückgabe des Team- Objektes, das den Teamnamen enthält – gemockt und eine Überprüfung des `view`-Namen ergänzt worden, sodass sich final der folgende Testablauf ergibt:

```
@Test
...
void test_01() throws Exception {
    when(teamService.load())
      .thenReturn(new Team(142, "Holstein Kiel"));

    RequestHelper.performGet(mvc, "/team")
                 .andExpect(status().isOk())
                 .andExpect(view().name("team/team"));
}
```

Sobald die entsprechende Seite erreichbar ist, kann mit der testgetriebenen Entwicklung ihrer Struktur – wie in Kapitel 3.1 verdeutlicht – begonnen werden. Neben dem grundlegenden Routing ist der Controller aber ebenfalls für die Aufbereitung und Bereitstellung der durch die Service-Schicht zur Verfügung gestellten Daten verantwortlich. Eine Überprüfung dieser Verantwortlichkeit lässt sich wie folgt realisieren:

```
@Test
@DisplayName("Das Model für die Team-Seite ist korrekt befüllt.")
void test_02() throws Exception {
    // Erstellen eines Team-Objektes zu Testzwecken
    // Mocking des Team-Services

    // Erstellen einiger Test-Spieler
    // Mocking des Player-Services

    // Erstellen der Total-Scores & Mocking

    RequestHelper.performGet(mvc, "/team")
                 .andExpect(model().attribute("teamForm", teamForm))
                 .andExpect(model().attribute("players", players))
                 .andExpect(model().attribute(
                     "totalScores", totalScores
                 ));
}
```

Aus Gründen der Übersichtlichkeit ist hier auf eine vollständige Darstellung des Tests verzichtet worden, der genaue Wortlaut bzw. der genaue Code ist der `TeamControllerTest.java` zu entnehmen. Erklärung der Funktionsweise des Tests folgt...

3.4 Testing der Service-Schicht

Hier kommt das Testing der Service-Schicht hin.

3.5 Datenbank-Tests

Hier kommt das Testen der Datenbank-Schicht hin.

4 Fazit

Hier kommt das Fazit hin.