Heinrich Heine
Universität
Düsseldorf

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

Die testgetriebene Entwicklung eines Spielzeiten-Planers für den Jugendfußball

Bachelorarbeit

vorgelegt von

Florian Ohmes

28. November 2024

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Erstgutachter: Dr. Jens Bendisposto
Zweitgutachter: Dr. Markus Brenneis

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 28. November 2024

Florian Ohmes

Zusammenfassung

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

Die Zusammenfassung sollte den Umfang einer Seite nicht überschreiten.

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. Den Kern dieses Tests bilden der mithilfe des `MockMvcs` simulierte Get-Request und die spezifische Überprüfung der HTTP-Antwort durch `andExpect`. Durch das zuvor konfigurierte Mocking der Services erhalten Entwickelnde die Kontrolle über die zur Verfügung gestellten Daten. Unter Verwendung von `model().attribute(‘‘attributeName’’,`

`‘‘expectedValue’’)` kann dann gezielt gesteuert werden, welche Werte `teamForm`, `players` und `totalScores` annehmen sollen, denn nur wenn sie dem `expectedValue` entsprechen, wird der gegebene Test bestehen. Für `players` beispielsweise bedeutet das, dass das Attribut genau diejenige Liste von Spielern als Wert annehmen muss, die durch die `loadPlayers`-Methode des `PlayerService` zurückgegeben wird.
Neben dem Überprüfen der Model-Attribute hat `test_02` aber auch eine sinnvolle Nebenwirkung: Durch die Art und Weise, wie er geschrieben ist, werden die Existenz eines `Team`-Objektes und einer `TeamForm` sowie ein `TeamMapper` gefordert, der für die Übersetzung zwischen Domänen- und Formular-Objekt zuständig ist. Diese Aufteilung fördert die Trennung der Verantwortlichkeiten – Web-UI von Geschäftslogik – und erhöht damit auch die Wartbarkeit des Codes, da zukünftige Änderungen am Team-Formular von der Geschäftslogik losgelöst durchgeführt werden können.
Wie bisher gezeigt fokussieren sich die beiden vorangegangenen Tests auf das Anfordern von Ressourcen auf dem Server mittels eines GET-Requests und die damit verbundene Datenaufbereitung und -bereitstellung. Im Gegensatz dazu steht der POST-Request, der für das Erstellen oder Verändern einer Ressource auf dem Server verantwortlich ist.
Konkret für die `TeamPage` bedeutet dies, dass nicht nur das Aufrufen der Team-Seite sowie die Anzeige des Teamnamens und der Spieler im Team eine wichtige Funktion darstellt, die der Spielzeitenplaner gewährleisten sollte, sondern auch die Möglichkeit, den Teamnamen zu bearbeiten und zu ändern, Spielerinformationen zu aktualisieren oder sich nicht mehr im Team befindliche Akteure zu löschen. Für die Unterstützung manipulierender Benutzeranfragen ist auch hier zunächst einmal die Etablierung einer grundlegenden Route vonnöten:

```
@Test
@DisplayName("Es werden Post-Requests über /team/teamname akzeptiert.")
void test_05() throws Exception {
    mvc.perform(post("/team/teamname").param("name", "Spring Boot FC"))
       .andExpect(status().is3xxRedirection())
       .andExpect(view().name("redirect:/team"));
}
```

Das grundsätzliche Prinzip solcher Anfragen in diesem Projekt ist es, für jeden solcher Requests eine individuelle Route anzulegen, die dann von einer speziellen Handler-Methode eines zuständigen Controllers verarbeitet wird. Im Anschluss an eine erfolgreiche Anfrage wird dann auf eine Get-Route weitergeleitet. Im Falle der `TeamPage` bedeutet dies Folgendes: Analog zum `GetMapping` wird hier parallel zum Testcode eine mit `@PostMapping(‘‘/teamname’’)` annotierte Handler-Methode geschrieben, die `‘‘redirect:/team’’` retourniert. Letzteres stellt sicher, dass nach abgeschlossener Verarbeitung zur Team-Seite umgeleitet wird.
In einem zweiten Schritt muss dann sichergestellt werden, dass die zuständige Service-Methode korrekt aufgerufen wird:

```
@Test
```

```
void test_06() throws Exception {
    Team team = new Team(null, "Spring Boot FC");
    mvc.perform(post("/team/teamname").param("name", team.name()));
    verify(teamService).save(team);
}
```

Eine wie zuvor beschriebene Überprüfung kann mithilfe von `Mockitos` `verify`-Methode realisiert werden. Durch die letzte Zeile des `test_06` kann zum einen sichergestellt werden, dass die `save`-Methode des `TeamService` – also die für das Speichern des Teamnamens zuständige Service-Methode – durch den `TeamController` aufgerufen wird, zum anderen aber auch überprüft werden, ob diese mit dem richtigen Parameter aufgerufen wird. Letzteres ist besonders wichtig bei schichtbasierten Softwarearchitekturen – wie der Onion-Architektur, um zu gewährleisten, dass Objekte zwischen den Schichten korrekt übergeben werden.
Für die `changeTeamName`-Methode des `TeamControllers` bedeutet dies konkret, dass gewährleistet wird, dass der Teamname korrekt in das `TeamForm`-Objekt integriert wird, dieses wiederum korrekt in ein `Team`-Objekt übersetzt wird und schließlich als Parameter an die `save`-Methode übergeben wird.
Doch neben Routing und Verwalten der Anwendungslogik ist ein Controller – insbesondere bei POST-Requests – auch noch für die Validierung der Benutzereingaben zuständig. Beim Teamnamen muss daher überprüft werden, dass das entsprechende Input-Feld nicht leer bzw. blank ist und eine Länge von 100 Zeichen nicht überschreitet, wie der `TeamForm.java` zu entnehmen ist. Für ein weniger triviales und von daher interessanteres Beispiel kann abermals die `PlayerPage` herangezogen werden – genauer gesagt das Formular zur Verwaltung der Spieler-Daten: Hier müssen Vorname, Nachname, Position und Trikotnummer geeignet validiert werden. Folgender Test der `TeamControllerTest.java` soll veranschaulichen, wie eine solche Validierung erzwungen bzw. kontrolliert werden kann:

```
@Test
...
void test_12() throws Exception {
    String html = mvc.perform(post("/team/savePlayer")
            // Parameter hinzufügen
            ...
        )
        .andExpect(model().attributeErrorCount("playerForm", 5))
        .andReturn().getResponse().getContentAsString();
    String html2 = mvc.perform(post("/team/savePlayer")
            // Parameter hinzufügen
            ...
        )
        .andExpect(model().attributeErrorCount("playerForm", 1))
        .andReturn().getResponse().getContentAsString();
```

```
        Elements errors = Jsoup.parse(html).select(".error");
        Elements errors2 = Jsoup.parse(html2).select(".error");
        assertThat(errors).hasSize(4);
        assertThat(errors2).hasSize(1);
    }
```

Der hier gezeigte Test ist nach folgendem Prinzip aufgebaut: Zunächst wird wieder ein entsprechender POST-Request simuliert, ehe der `AttributeErrorCount` des zu betrachtenden Objektes – in diesem Fall die `PlayerForm` – überprüft wird. Im Anschluss wird dann noch kontrolliert, ob potenzielle Fehlermeldungen auch tatsächlich auf der Seite angezeigt werden. Dazu wird die durch die Anfrage zurückgegebene Antwort mit `Jsoup` geparst sowie alle `div`-Container mit der Klasse `error` extrahiert und im Bezug auf ihre Größe inspiziert. Für das hier vorgestellte Beispiel werden zwei verschiedene Anfragen simuliert, die jeweils so konzipiert sind, dass sie möglichst viele Fehler abdecken sollen.
Der ersten Anfrage, die final im String mit dem Namen `html` gespeichert wird, werden somit jeweils ein leerer String als Parameter für den Vornamen, Nachnamen und die Position hinzugefügt. Damit der `AttributeErrorCount` für die `PlayerForm` nun die erwartete Größe besitzt, müssen gleich mehrere Anpassungen im Produktivcode vorgenommen werden: In der entsprechenden Handler-Methode des Controllers muss die `PlayerForm` mit `@Valid` annotiert werden, damit die Benutzereingabe auch wirklich validiert wird. Des Weiteren muss direkt nach der `PlayerForm` ein weiterer Parameter vom Typ `BindingResult` eingefügt werden, in dem das Ergebnis der auf die Eingabe angewendeten Validierung gespeichert und an das Formular-Objekt gebunden wird.
Diese Schritte alleine reichen jedoch nicht aus, um den Test bestehen zu lassen. Damit das `BindingResult` nicht einfach leer bleibt, muss die eigentliche Validierung noch konfiguriert werden. Dies geschieht innerhalb der Formular-Klasse `PlayerForm.java` mithilfe geeigneter Validierungs-Annotationen. Dort wird festgelegt, dass Attribute wie der Vor- und Nachname oder die Position nicht blank sein dürfen sowie Letztere zwischen einem und fünf Zeichen lang sein und die Trikotnummer zwischen eins und 99 liegen muss.
Im Falle der ersten Anfrage – die oben bereits beschrieben wurde – werden also sämtliche `@NotBlank`-Annotation geprüft sowie die `@Size`-Annotation der Position und die `@NotNull`-Annotation der Trikotnummer. Schließlich wird dann noch in der Variable `error` gespeichert, ob für jedes Eingabefeld ein entsprechender Fehler-Container auf der Seite vorhanden ist und ein Fehler angezeigt wird – die beiden Fehler bezüglich der Position werden dabei in ein und demselben Container angezeigt, da sie sich jeweils auf dasselbe Attribut beziehen.
Im Falle der zweiten Anfrage wird dann noch eine vollständige Benutzereingabe simuliert, jedoch wird hierbei eine Trikotnummer über 100 gewählt, um die `Max`-Validierung der `jerseyNumber` zu begutachten. Dementsprechend wird also ein `AttributeErrorCount` von eins erwartet sowie ein Container mit einer Fehlermeldung auf der `PlayerPage`.
Auf die hier gezeigte Weise kann also die Validierung von Benutzereingaben und die Fehlerausgabe testgetrieben entwickelt werden. Das grundlegende Prinzip lässt sich prinzipiell auf andere Formulare übertragen, sogar verschachtelte Formulare sind umsetzbar, wie in der `RecapForm.java` und der `FormAssessment.java` gezeigt. Dort wird zum einen das

Recap-Formular an sich validiert, das bedeutet das Datum des Recaps darf weder leer sein noch in der Zukunft liegen, aber auch die Liste der Bewertungen soll validiert werden. Für jedes `FormAssessment` ist daher zusätzlich noch festgelegt, dass der Wert nicht `null` sowie zwischen null und fünf liegen muss.

3.4 Testing der Service-Schicht

Nachdem nun ausgiebig über das Testing des Web-Interfaces gesprochen wurde, soll sich das kommende Kapitel nun einer weiteren wichtigen Komponente, der Service- Schicht, widmen. Diese kann unabhängig von den anderen Komponenten entwickelt werden. Im Falle des hier vorliegenden Projektes – der Entwicklung des Spielzeitenplaners – ist die Service-Schicht schrittweise aufgebaut worden.
Dabei wurde sich an verschiedenen `use cases` orientiert, die die funktionalen Anforderungen der Anwendung hervorheben und die wiederum eine Interaktion der Nutzenden mit der Web-Oberfläche als Startpunkt besitzen. Ein Beispiel für einen in diesem Projekt vorliegenden `use case` wäre beispielsweise das Löschen eines Spielers durch die Nutzenden. Durch einen Klick auf den `Löschen`-Button auf der `TeamPage` wird das entsprechende Formular an den Server geschickt. Dort wird es von einer entsprechend konfigurierten Handler-Methode eines Controllers in Empfang genommen und verarbeitet. Im Zuge dessen wird die entsprechende Service-Methode – in diesem Fall die `deletePlayer`-Methode des `PlayerService` – aufgerufen, die den Fall dann bearbeitet. Schließlich ruft diese dann eine entsprechende Repository-Methode auf, die wiederum für die Löschung der Spieler-Daten in der Datenbank zuständig ist.
Innerhalb dieser Aufruf-Kette fordert also gewissermaßen eine Komponente die Existenz einer anderen. Bezogen auf die Onion-Architektur bedeutet dies: die äußere Web-Schicht erwartet das Vorhandensein der inneren Schichten – also der Service-Schicht und des Domain-Models – und diese wiederum die Verfügbarkeit der (äußeren) Persistenz-Schicht. Auf diese Weise können sich Entwickelnde vom einen Ende der Software-Zwiebel hindurch zum anderen arbeiten.
Ein konkretes Beispiel aus der Entwicklung des Spielzeitenplaners ist dem Commit `0a64ca3` zu entnehmen. Durch die Implementierung der Funktionalitäten in der Web-Schicht – siehe dazu die Änderungen an der `TeamControllerTest.java` – wird unter anderem das Vorhandensein der `deletePlayer`-Methode und damit verbunden auch die Existenz des `PlayerService` an sich gefordert. Um den Test also bestehen zu lassen und schließlich committen zu können ist die Erstellung einer `PlayerService.java` und einer darin enthaltenen `deletePlayer`-Methode zwingend erforderlich.
Sobald die Service-Klasse vorhanden ist, kann sie auch testgetrieben entwickelt werden. Im Folgenden soll nun zunächst das Vorgehen bei den sogenannten Basisoperationen geschildert werden, ehe dann auf komplexere Testfälle eingegangen werden soll. Das zuvor genannte Beispiel für die `deletePlayer`-Methode lässt sich einfach und unkompliziert wie folgt testgetrieben entwickeln:

```
@Test
...
void test_01() {
    Integer playerId = 17;
    playerService.deletePlayer(playerId);
    verify(playerRepository).deleteById(playerId);
}
```

Hier wird also überprüft, ob der `PlayerService` die Löschanfrage korrekt weiterverarbeitet. In diesem Fall bedeutet dies, dass diese an das zuständige Repository delegiert wird. Dabei ist besonders wichtig, dass das richtige Repository – hier also das `PlayerRepository` – verwendet wird und die entsprechende Methode mit dem aus der ursprünglichen Anfrage gesendeten Objekt aufgerufen wird.
Um diesen Test so schreiben zu können, muss das `PlayerRepository` zunächst mithilfe von `Mockito` gemockt werden, um die Kontrolle über sämtliche Aktionen des Repository zu erhalten, und der `PlayerService` mit dem soeben gemockten Repository ordnungsgemäß initialisiert werden, was wiederum einen geeigneten Konstruktor in der `PlayerService.java` voraussetzt. In diesem Zuge wird dann auch das Interface `PlayerRepository.java` innerhalb der Service-Schicht etabliert und seine Implementierung, die mit `@Repository` annotierte `PlayerRepositoryImpl.java`, in der Persistenzschicht. Ihre testgetriebene Entwicklung ist in Kapitel 3.5 beschrieben. Sind die zuvor genannten Voraussetzungen implementiert, so kann der `deletePlayer`-Methode der Aufruf `playerRepository.deleteById(id)` hinzugefügt werden, wobei `id` als Argument entgegen genommen wird (siehe dazu `PlayerService.java nach Commit e4bc878`).
Die Methode zum Löschen eines Spielers ist ein Beispiel für eine Basisoperation, bei der eine Anfrage an die entsprechende Repository-Methode weitergeleitet wird. Hier steht also die Delegation der Aufgabe im Vordergrund, während das Repository sich um die Löschlogik kümmert. Für die Verifizierung der Übergabe des korrekten Parameters wurde `deleteById(playerId)` auf `verify(playerRepository)` aufgerufen. Dies ist in diesem Fall auch völlig ausreichend, sollten jedoch mehrere Methodenaufrufe mit unterschiedlichen Argumenten auftreten, komplexe Objekte – wie beispielsweise ein Spieler mit seinen verschieden Attributen – übergeben oder aber Eigenschaften innerhalb des Methodenaufrufs verändert werden, so ist der Gebrauch eines `ArgumentCaptors` durchaus sinnvoll, wie `test_01` der `RecapServiceTest.java` veranschaulicht:

```
@Test
@DisplayName("Die Bewertungen werden gespeichert.")
void test_01() {
    List<Assessment> assessments = new ArrayList<>(List.of(
            // Hier manuell erstellte Bewertungen hinzufügen
    ));
    ArgumentCaptor<Assessment> assessmentCaptor =
        ArgumentCaptor.forClass(Assessment.class);
```

```
        recapService.submitAssessments(assessments);

        verify(assessmentRepository, times(4))
            .save(assessmentCaptor.capture());

        List<Assessment> savedAssessments =
            assessmentCaptor.getAllValues();

        assertThat(savedAssessments).isEqualTo(assessments);
    }
```

In dem oben gezeigten Test wird der `ArgumentCaptor` dazu benutzt, die verschiedenen Bewertungen, mit der die `save`-Methode aufgerufen wird, einzufangen. Bevor dies mithilfe `assessmentCaptor.capture()` geschehen kann, muss die spezifische Klasse, für die der `Captor` programmiert ist, zunächst bei der Initialisierung des `assessmentCaptors` konfiguriert werden. Nach dem Erfassungsvorgang können alle gespeicherten `Assessments` dann abgerufen und in einer Liste gespeichert, um schließlich weiteren Überprüfungen mithilfe von `asserThat` unterzogen werden zu können.
Während bisher Gesagtes vor allen Dingen das Delegieren eines Aufrufs an die Persistenzschicht und die Überprüfung der Übergabe von Parametern in den Vordergrund stellt, soll nun im Folgenden auf die Überprüfung von Rückgabewerten eingegangen werden. Eine in diesem Projekt häufig getestete Methode ist die `load`-Methode, die Daten aus der Datenbank anfragt und diese dem Web-UI zur Verfügung stellt. Beispielsweise liefert die `loadCriteria`-Methode des `SettingsService` die aktuell gespeicherten Kriterien, `loadFormation` stellt die aktuell genutzte Formation bereit und `loadPlayers` gewährleistet die Versorgung der Webschicht mit den aktuellen Spielern im Team. Letztere wird wie folgt getestet:

```
    @Test
    ...
    void test_02() {
        List<Player> players = TestObjectGenerator.generatePlayers();
        when(playerRepository.findAll()).thenReturn(players);

        List<Player> loadedPlayers = playerService.loadPlayers();

        verify(playerRepository).findAll();
        assertThat(loadedPlayers)
            .containsExactlyInAnyOrderElementsOf(players);
    }
```

Nachdem der `TestObjectGenerator` eine Liste von (Test-)Spielern bereitgestellt hat, kann

das `PlayerRepository` so konfiguriert werden, dass es diese Liste von Spielern zurückgibt. Sämtliche Repositories werden für die Service-Tests gemockt, um die Service-Schicht von der Datenbank losgelöst testen zu können. In dem folgenden `Act`-Schritt des Tests wird die `loadPlayers`-Methode dann ausgeführt und die Rückgabe in einer lokalen Variable gespeichert. So kann dann abschließend getestet werden, ob die `findAll`-Methode des `PlayerRepository` aufgerufen wurde – die Überprüfung der Delegation eines Requests ist ja bereits aus den vorangegangenen Tests bekannt – und ob die geladenen Spieler auch wirklich der gewünschten Liste entsprechen.
Dabei wird die `containsExactlyInAnyOrderElementsOf`-Methode verwendet, um sicherzustellen, dass sich auch wirklich nur die erwarteten Spieler in der Liste befinden und keine weiteren. Außerdem spielt die Reihenfolge keine Rolle, damit der Test auch weiterhin besteht, sollten die Spieler nach Namen, Score, Trikotnummer oder Position geordnet werden. Zusammenfassend kann gesagt werden, dass beim Delegieren und Koordinieren von Anfragen ein besonderes Augenmerk zum einen auf den korrekten Methodenaufruf, zum anderen aber auch die Überprüfung der Übergabe der korrekten Parameter sowie die Richtigkeit des Rückgabewertes gelegt werden sollte, denn diese sind für eine ordnungsgemäße Funktionsweise der Anwendung unerlässlich.
Darüber hinaus ist es wichtig Methoden der Service-Schicht zu testen, die sich nicht eindeutig einer Klasse aus dem Domain-Model zuordnen lassen oder aber eine Verknüpfung zwischen mehreren Entitäten herstellen. Im Bezug auf den Spielzeitenplaner ist dies zum Beispiel der Fall in der `SpielzeitenService.java`, in der aus allen verfügbaren Spielern ein Kader oder aus dem Kader wiederum eine Startelf ermittelt werden soll, und in der `PlayerService.java`, in der für einen Spieler für jedes Kriterium anhand mehrerer Bewertungen ein Score berechnet wird.
Mit Letzterem ist die `calculateScore`-Methode gemeint, die mithilfe mehrerer Tests in ihre gegenwärtige Form gebracht worden ist. Der Kern der Methode besteht aus einer `if`-Abfrage, die, wenn es sich um das Kriterium `Trainingsbeteiligung` handelt, eine private Methode speziell für diesen Fall aufruft, für alle anderen Kriterien den Score mithilfe der `calculateScoreOther`-Methode auf die gewöhnliche Weise berechnet. Die einzelnen Abzweigungen der Methode lassen sich gut in getrennten Tests entwickeln. Im Folgenden ist der Test gegeben, der überprüft, ob die Score-Berechnung im Falle eines gewöhnlichen Kriteriums – also einem anderen als der `Trainingsbeteiligung` – erfolgreich funktioniert:

```
@Test
...
void test_06() {
    Integer playerId = 1;
    Criterion criterion =
        new Criterion(2, "Sozialverhalten", "S", 0.25);
    Setting weeksGeneral = new Setting(1195, "weeksGeneral", 4.0);
    List<Assessment> assessments = new ArrayList<>(List.of(
        // Erstellen mehrerer Assessments (Bewertungen)
        ...
```

```
        ));
        Double expectedScore = assessments.stream()
            .mapToDouble(Assessment::getValue).average().orElseThrow();

        when(criterionRepository.findById(criterion.getId()))
            .thenReturn(Optional.of(criterion));
        when(settingRepository.findById(weeksGeneral.getId()))
            .thenReturn(Optional.of(weeksGeneral));
        when(assessmentRepository.findByPlayerIdAndCriterionIdAndDateAfter(
                playerId, criterion.getId(), LocalDate.now().minusWeeks(
                    weeksGeneral.getValue().intValue()).minusDays(1))
            ).thenReturn(assessments);

        Double score =
            playerService.calculateScore(criterion.getId(), playerId);

        assertThat(score).isEqualTo(expectedScore);
    }
```

Zunächst einmal sind hier einige `Arrange`-Schritte notwendig: Da die Methode eine `playerId` und eine `criterionId` als Eingabeparameter bekommt, müssen diese erst einmal definiert werden. Außerdem ist bei der Score-Berechnung der Wert der Einstellung `weeksGeneral` vonnöten, da dieser den Zeitraum festlegt, innerhalb dessen Bewertungen berücksichtigt werden. Des Weiteren wird dann noch eine Liste mit verschiedenen Bewertungen erstellt, die in die Berechnung einfließen sollen. Schließlich ergibt sich der `expectedScore`, der den Durchschnitt der Werte aller gültigen Bewertungen des Spielers für ein bestimmtes Kriterium bildet.
Nachdem die nötigen Werte und (Test-)Objekte generiert worden sind, müssen nun die durch die `calculateScore`-Methode direkt oder indirekt genutzten Repositories und ihre Methoden gemockt werden, um so die Kontrolle über die durch die Persistenzschicht bereitgestellten Daten zu bekommen, denn nur so kann es zu einer Überprüfung der Funktionsweise der zu testenden Service-Methode kommen. So müssen zum Beispiel die `findById`-Methoden des `criterionRepository` und des `settingRepository` so konfiguriert werden, dass sie bei einem Aufruf die zuvor generierten Objekte – also `criterion` bzw. `weeksGeneral` – zurückgeben.
Darüber hinaus muss dann noch diejenige Methode des `assessmentRepository`, die die Bewertungen nach Spieler, Kriterium und Datum filtert und die entsprechenden Daten lädt, so justiert werden, dass sie die zuvor erstellten Test-Bewertungen zur Verfügung stellt. Das eigentliche Testing des Ladens von Bewertungen aus der Datenbank ist in Kapitel 3.5 beschrieben, diese Funktionalität wird losgelöst von den anderen Schichten getestet, denn hier geht es lediglich darum, ob das Berechnen des Scores korrekt funktioniert.
Da nun alle `Arrangements` – also alle Vorbereitungen – getroffen sind, kann der `Act`-Schritt des Tests vollzogen werden: der Aufruf der `calculateScore`-Methode auf dem

`PlayerService` mit den notwendigen Parametern. Da es sich in diesem Fall nicht um das Kriterium `Trainingsbeteiligung` handelt, wird der Score als Durchschnitt aller gefilterter Bewertungen berechnet. Ob dieser dem erwarteten Wert entspricht, wird abschließend mittels eines geeigneten `assertThat`-Statements geprüft.
Für den Fall des Kriteriums `Trainingsbeteiligung` gibt es darüber hinaus noch einen weiteren Testfall, der durch `test_07` der `PlayerServiceTest.java` abgedeckt ist. Auch hier werden wieder sämtliche Abhängigkeiten zu anderen Schichten durch gezieltes Mocking aufgelöst, sodass der Code zur Berechnung des Trainings-Scores im Fokus steht und in Isolation getestet werden kann, um so eine ordnungsgemäße Funktionsweise zu gewährleisten.

3.5 Datenbank-Tests

In den vorangegangenen Kapiteln wurde gezeigt, wie innerhalb der Web- und Service-Schicht testgetrieben entwickelt werden kann. Dabei wurde zunächst die Benutzeroberfläche entwickelt und damit verbunden auch die Web-Steuereinheiten – die Controller. Diese rufen unter anderem die für die Benutzeranfragen zuständigen Service-Methoden auf. Deren testgetriebene Entwicklung erforderte wiederum die Existenz einiger Repositories und ihrer Methoden, die für das Laden der entsprechenden Daten aus der Datenbank verantwortlich sind.
Das folgende Kapitel soll sich demnach der testgetriebenen Entwicklung dieser Repositories – und damit verbunden der Persistenzschicht im Allgemeinen – widmen. Im Allgemeinen ist zu sagen, dass für sämtliche Datenbank-Tests die `Testcontainers`-Bibliothek in Kombination mit `Docker` verwendet wird. Eine solche Konfiguration bietet gleich mehrere Vorteile: Erstens werden so realistische Tests mit einer echten Instanz der für das Projekt ausgewählten Datenbank ermöglicht. Die Test-Datenbank kann somit exakt so wie die Produktiv-Datenbank konfiguriert werden, wodurch möglichst realistische Bedingungen simuliert werden.
Zweitens sorgt die Verwendung von `Testcontainers` dafür, dass jeder Test in einer bereinigten Umgebung stattfindet, um zu verhindern, dass sie einander beeinflussen und das Ergebnis verfälschen. Drittens wird das Hoch- und Runterfahren sowie das Setup des `Docker-Containers` automatisiert und von `Testcontainers` übernommen.
Grundsätzlich sind alle Datenbank-Testklassen, die im Verzeichnis `src/test/java/de/bathesis/spielzeitenplaner/database` zu finden sind, auf die gleiche Art und Weise aufgebaut. Zunächst einmal muss die jeweilige Testklasse mit drei verschiedenen Annotationen versehen werden: (1.) `@DataJdbcTest`, durch die Spring eine spezielle Testumgebung für die Persistenzschicht konfiguriert, der Web-Layer zum Beispiel wird komplett deaktiviert, um ressourcenschonender arbeiten zu können, (2.) `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)`, die dafür sorgt, dass Spring die Produktiv-Datenbank nicht automatisch durch eine `In-Memory`-Datenbank ersetzt, sondern die durch `Testcontainers` konfigurierte Test-Datenbank benutzt, und (3.) `@Testcontainers`, die für das Verwaltung der innerhalb der Klasse definierten Container zuständig ist (`Start`, `Stop` und `CleanUp`).

4 Fazit

Hier kommt das Fazit hin.