Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Till Theis

Das Play-Framework und dessen Einsatz zur Entwicklung von Real-Time-Web-Anwendungen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Till Theis

Das Play-Framework und dessen Einsatz zur Entwicklung von Real-Time-Web-Anwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Friedrich Esser
Zweitgutachter: Prof. Dr. Julia Padberg

Eingereicht am: 21. November 2013

Till Theis

Thema der Arbeit
Das Play-Framework und dessen Einsatz zur Entwicklung von Real-Time-Web-Anwendungen

Stichworte
Play, Scala, Real-Time-Web, Reaktiv, Funktionale Programmierung, Iteratees

Kurzzusammenfassung
Diese Arbeit stellt das Play Framework vor und legt ihren Schwerpunkt dabei auf die Entwicklung von Real-Time-Web-Anwendungen. Dies sind interaktive Anwendungen, die dem Nutzer Daten anzeigen, sobald sie dem Server bekannt werden und nicht erst, wenn der Nutzer die Seite neu lädt. Um dies zu erreichen, stellt Play eine Iteratee-Bibliothek zur Verfügung, die es ermöglicht Datenströme reaktiv und inkrementell zu verarbeiten. Diese Iteratee-Streams werden mitsamt ihren Designentscheidungen und Anwendungsmöglichkeiten erläutert, um anschließend eine Anwendung auf Basis von ihnen zu entwickeln.
Till Theis

Title of the paper
The Play Framework and its Use for Developing Real-Time Web Applications

Keywords
Play, Scala, Real-Time Web, Reactive, Functional Programming, Iteratees

Abstract
This thesis is an introduction to the Play framework and focuses on its features for developing real-time web applications. Real-time web applications are applications that push information to the user as soon as the server receives them instead of requiring the user to manually reload the page. In order to achieve that Play ships with an Iteratee library that makes it possible to reactively and incrementally process data streams. These Iteratee streams as well as their design and application possibilities will be explained to finally develop an application that builds on them.

Inhaltsverzeichnis

Listings

1. Einleitung

In diesem Kapitel wird erläutert, was das Real-Time-Web ausmacht und was mit dem Play-Framework und dieser Arbeit erreicht werden soll. Es wird ein Überblick über die bisherige Entwicklung des Webs gegeben und differenziert, womit sich diese Arbeit genau beschäftigt und womit nicht. Anschließend wird beschrieben, wie die weitere Struktur dieser Arbeit aussieht.

1.1. Entwicklung des Webs

Nach Taivalsaari und Mikkonen 2011 (S. 1–3) war das World Wide Web der 1990er Jahre ein Medium, bei dem einzelne Dokumente im Vordergrund standen. Das klassische Web der ersten Hälfte des Jahrzehnts bestand aus Dokumenten mit Text, Bildern und Links. Die zweite Hälfte machte das Hybrid Web aus. Browsertechnologien, wie Javascript und CSS, sowie Plugins wie Flash und QuickTime wurden entwickelt und erweiterten die statischen Dokumente um interaktive Elemente.

In den 2000ern verbreitete sich das dynamische Senden und Anfordern von Daten unter dem Namen Ajax (Asynchronous JavaScript and XML). Gegen Ende des Jahrzehnts wurden die vorhandenen Technologien genutzt, um ganze Anwendungen damit zu schreiben, wie z. B. Facebook oder Twitter.

1.2. Real-Time-Web

Eine Real-Time-Web-Anwendung ist eine Web-Anwendung, die automatisch neue Daten an den Client sendet, sobald diese dem Server bekannt werden. Dies unterscheidet sich vom klassischen Pull-Prinzip, in dem nur der Client per Anfrage eine neue Seite beim Server anfordern kann. Bei Real-Time-Events ist es stattdessen notwendig, dass der Server mittels Push-Prinzip neue Informationen an den Client sendet (vgl. Bozdag u. a. 2007, S. 1). Facebook beispielsweise zeigt neue Statusmeldungen von Freunden an, ohne dass manuell die Seite neu geladen werden muss.

1.3. Motivation und Ziel

Wenn man die in Abschnitt 1.1 beschriebene Entwicklung des Webs betrachtet, knüpft das Real-Time-Web genau daran an. Der Nachrichtenverkehr verläuft nicht mehr nur vom Client zum Server, sondern auch in umgekehrter Richtung vom Server zum Client. Das Play-Framework wird von Typesafe, der Firma von Scalas Begründer Martin Odersky, unterstützt und wirbt damit, die passenden Werkzeuge zur Entwicklung moderner Web-Anwendungen zu besitzen (vgl. Zenexity und Typesafe Inc. 2013k). Kernpunkte, die Play auszeichnen, sind starke Typsicherheit durch die Nutzung von Scala, sowie eine Architektur, die moderne clientseitige Technologien unterstützt (vgl. Hilton u. a. 2013, S. 4). Es gibt zwar weitere Web-Frameworks für Scala, diese sind allerdings weniger breit aufgebaut, weshalb Play das Potenzial besitzt, das Framework für Scala zu werden (vgl. Hilton u. a. 2013, S. 8). Ziel dieser Arbeit ist es, herauszufinden, ob Play diesen Anforderungen gerecht wird und wie damit Real-Time-Web-Anwendungen entwickelt werden können. Dazu sollen die dafür benötigten Kernkomponenten vorgestellt und analysiert werden, um damit schließlich eine eigene Real-Time-Web-Anwendung zu entwickeln.

1.4. Themenabgrenzung

Diese Arbeit beschäftigt sich in erster Linie mit den Teilen des Play-Frameworks, die benötigt werden, um Real-Time-Web-Anwendungen damit zu entwickeln. Zusätzlich wird benötigtes Hintergrundwissen, wie Architektur und Grundlagen des Frameworks zur Verfügung gestellt. Einige Themenbereiche, die dafür nicht elementar sind, wie z. B. Datenbanken, werden nicht behandelt. Des Weiteren wird vorausgesetzt, dass der/die LeserIn bereits mit den Grundtechnologien der Web-Entwicklung, wie HTML, CSS und JavaScript vertraut ist. Ein JavaScript-Framework wird aber nicht verwendet, um kein weiteres Vorwissen vorauszusetzen.

1.5. Struktur

Im zweiten Kapitel wird gezeigt, welche grundlegenden Werkzeuge das Framework für die Entwicklung von Web-Anwendungen besitzt und wie diese zu verwenden sind. Dazu wird zunächst erklärt, wie eine statische Website erstellt wird, um diese später um Real-Time-Komponenten zu erweitern. Nachdem die Grundtechniken zur Handhabung des Frameworks aufgezeigt worden sind, wird im dritten Kapitel in die reaktive Programmierung eingeführt. Diese ist grundlegend, um Plays Stream-Bibliothek zu verstehen, die im Mittelpunkt der Entwicklung von Real-Time-Web-Anwendungen steht. Das vierte Kapitel erklärt, wie auf der Client-Seite mittels JavaScript mit dem Server kommuniziert wird. Mit diesem Wissen wird

im fünften Kapitel schließlich eine Anwendung erstellt, die all die behandelten Punkte in praktischer Art und Weise zusammenführt.

2. Grundlagen

In diesem Kapitel werden die grundlegenden Techniken zur Webseiten-Entwicklung mit dem Play-Framework vorgestellt. Es werden hierbei in erster Linie die Komponenten vorgestellt, die für die Entwicklung von Real-Time-Web-Anwendungen unbedingt notwendig sind. Um die Techniken zur Entwicklung von statischen Web-Anwendungen zu demonstrieren, soll im Folgenden eine kleine Anwendung entwickelt werden. Diese Anwendung fragt ihre Benutzer, wie alt sie sind und bildet die gesammelten Information als Diagramm ab. Anhand dieser Anwendung soll die Entwicklung mit Models, Views und Controllern erklärt werden. Bevor mit der Implementierung begonnen werden kann, muss allerdings die die Grundarchitektur des Frameworks erklärt werden.

2.1. Architektur

Auf der untersten Ebene existiert ein Web-Server, der mit dem Framework ausgeliefert wird. Die Anfragen, die der Web-Server empfängt werden an Play weitergeleitet und schließlich von der Anwendung verarbeitet. Nach der Verarbeitung wird eine Antwort generiert und schließlich als HTTP-Antwort versendet. Der Anwendungscode einer Play-Applikation ist nach der Model-View-Controller-Architektur (MVC) aufgebaut.

Models beinhalten die Anwendungslogik der Applikation. Views stellen Informationen der Models i. d. R. als HTML-Seiten dar. Controller bilden eingehende HTTP-Anfragen mit Hilfe von Models und Views auf HTTP-Antworten ab. Controller empfangen HTTP-Anfragen, berechnen mit Hilfe der Models eine Antwort, lassen diese von einer View übersetzen, um das Ergebnis der View schließlich als HTTP-Antwort zurückzusenden (vgl. Hilton u. a. 2013, S. 45–48).

Ruft ein(e) WebseitenbesucherIn beispielsweise ein Nutzerprofil auf, so generiert der Web-Browser eine HTTP-Anfrage, die an den von Play verwendeten Web-Server gesendet wird. Das Play-Framework ermittelt den zuständigen Controller für die Anfrage und übergibt ihm diese. Der Controller sucht mit Hilfe der Anwendungslogik den passenden Nutzer (z. B. aus der Datenbank) heraus und übergibt dieses Nutzer-Model an die passende View. Die View bereitet die im Nutzer-Model gekapselten Informationen als HTML-Seite auf. Anschließend

gibt der Controller diese HTML-Darstellung des Nutzerprofils zurück, was vom Web-Server dann an den Web-Browser gesendet und schließlich dem/der Webseitenbesucher(in) angezeigt wird.

2.2. Erstellen einer Anwendung

Um eine neue Anwendung mit Play zu erstellen, muss auf der Kommandozeile in den Ordner navigiert werden, in dem das Projekt erstellt werden soll. Anschließend kann mit `play new <project name>` ein neues Projekt angelegt werden. In diesem Fall ist der Projektname `age_statistics_http`. Im startenden Assistenten wählt man die Hauptprogrammiersprache aus (Java oder Scala), in diesem Fall Scala. Daraufhin wird im aktuellen Verzeichnis ein neuer Ordner mit dem vorher angegebenen Namen `age_statistics` angelegt (vgl. Hilton u. a. 2013, S. 10). Die darin vorzufindende Verzeichnisstruktur wird im nächsten Abschnitt beschrieben.

2.3. Verzeichnisstruktur

Die Verzeichnisstruktur einer Play-Anwendung ist immer gleich. An dieser Stelle werden nur die Ordner vorgestellt, die für das Verständnis dieser Arbeit wichtig sind. Diese Ordner sind folgende (vgl. Zenexity und Typesafe Inc. 2013b):

app/	ausführbare Komponenten
app/controllers/	Controller-Komponenten
app/models/	Model-Komponenten
app/views/	View-Komponenten
conf/	Konfigurationsdateien
public/	öffentliche statische Dateien (JS, CSS, Bilder)

2.4. Starten einer Anwendung

Um die neu erstellte Anwendung zu starten, muss sie erst kompiliert werden. Das geschieht ebenfalls mit Hilfe des `play`-Befehls. Mit dem Aufruf von `play` gelangt man in die Play-Konsole. Von dort aus kann mit `compile` der Source-Code kompiliert und anschließend mit `run` ausgeführt werden. Der Aufruf von `compile` ist allerdings optional. Falls noch nicht

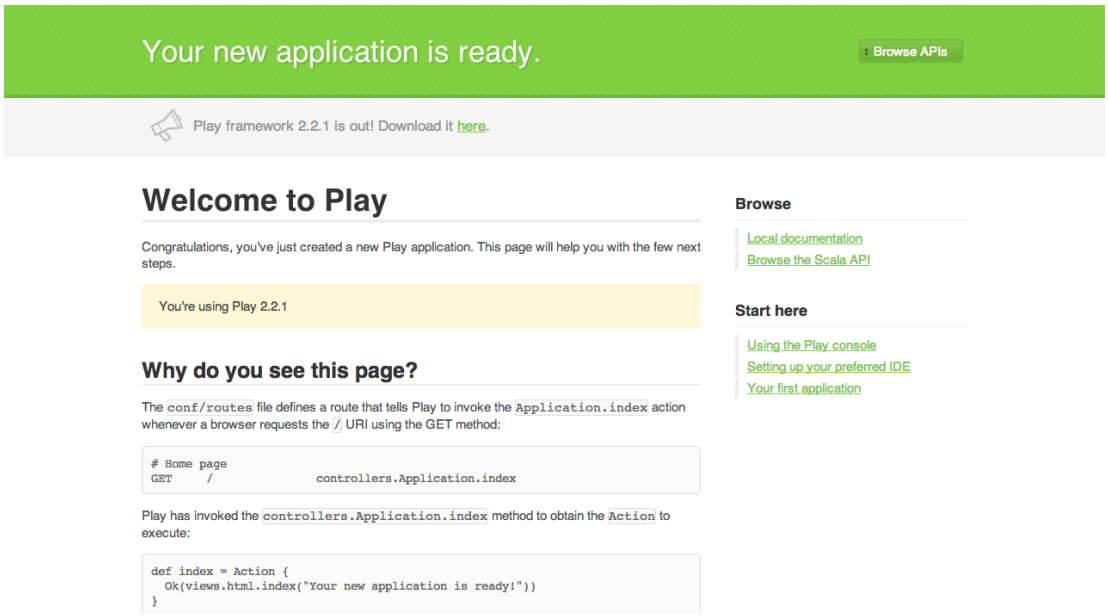

Abbildung 2.1.: Eine neu erstellte Play-Anwendung

kompilierte Änderungen existieren, werden diese beim Aufruf von `run` automatisch kompiliert, sobald die Website aufgerufen wird (vgl. Zenexity und Typesafe Inc. 2013q).

Jede neue Play-Anwendung ist bereits eine lauffähige Website. Diese kann nach dem Starten unter der URL `http://localhost:9000/` abgerufen werden (siehe Abb. 2.1). Um die Anwendung bei Dateiänderungen automatisch neu kompilieren zu lassen, ohne sie erst im Browser aufrufen zu müssen, kann in der Play-Konsole statt `run` `~run` ausgeführt werden (vgl. Hilton u. a. 2013, S. 12).

2.5. Routing

Wenn die Website von jemandem aufgerufen wird, soll eine bestimmte Controller-Aktion ausgeführt werden. Damit Play weiß, welche Controller-Aktion für die aufgerufene URL ausgeführt werden soll, muss dies in der `routes`-Datei definiert werden. Diese Datei befindet sich unter `conf/routes` und besitzt nach der Erstellung einer Anwendung bereits zwei Einträge (vgl. Listing 2.1). An dieser Stelle ist allerdings nur der erste Eintrag interessant.

```
1  GET     /     controllers.Application.index
```

Listing 2.1: Die routes-Datei

Dieser Eintrag bedeutet, dass HTTP-Anfragen der Methode `GET` der URL `/` auf die Controller-Aktion `index` des Controllers `controllers.Application` abgebildet werden. Neben `GET` gibt es noch `POST`, `PUT`, `DELETE` und `HEAD`. Die am Häufigsten verwendeten HTTP-Methoden sind `GET`, `POST`, `PUT` und `DELETE`, um Daten abzufragen, zu bearbeiten, zu erstellen und zu löschen (vgl. Hilton u. a. 2013, S. 7). Der Pfad `/` steht für die URL `http://localhost:9000/`. Es sind auch längere Pfade, wie z. B. `/animals/cat` möglich, davon wird in diesem Beispiel allerdings nicht Gebrauch gemacht. Die Controller-Aktion `controllers.Application.index`, auf die der Routing-Eintrag zeigt, ist bereits implementiert. Diese wird in Abschnitt 2.7 allerdings durch eine eigene Implementierung ersetzt.

2.6. Model

Bevor die Anwendung Daten anzeigen kann, muss eine Datenstruktur für die Altersstatistiken entworfen werden. Für diese Statistik sollen nur die Alterszahlen und die Personenzahl des jeweiligen Alters gesammelt werden. Dafür eignet sich eine `Map[Int, Int]`, wobei die Schlüssel das Alter sind und die Werte dazu die Anzahl an Personen, die dieses Alter haben. Um den Code verständlicher zu machen, wird ein Typ-Alias mit dem Namen `AgeStatistics` eingeführt. Dies wird in der Datei `app/models/package.scala` durchgeführt und hat den in Listing 2.2 gezeigten Inhalt. Die Verwendung eines `package object`s ist nötig, um den Typ-Alias paket-weit einzurichten (vgl. Odersky und Spoon 2010).

```
1 package object models {
2   type AgeStatistics = Map[Int, Int]
3 }
```

Listing 2.2: Der AgeStatistics-Typ-Alias

Um die Verwendung noch komfortabler zu machen, wird in `app/models/AgeStatistics.scala` ein Objekt mit Konstruktormethoden erstellt, wie in Listing 2.3 zu sehen. Es wird eine `apply`-Methode zur Verfügung gestellt, sodass die Werte der Statistik direkt übergeben werden können, wie z. B. `AgeStatistics(6 -> 1, 10 -> 2)`. Dadurch, dass `AgeStatistics.empty` für undefinierte Schlüssel einen Standardwert von null (0) zurückgibt, können auch unbekannte Alterseinträge abgefragt werden, ohne dass ein Fehler auftritt.

```
1 object AgeStatistics {
2   def apply(statistics: (Int, Int)*): AgeStatistics =
3     empty ++ Map(statistics: _*)
4   val empty: AgeStatistics = Map.empty.withDefaultValue(0)
5   val sample: AgeStatistics = apply( 6 -> 1 /* , ... */ )
```

```
}
```

Listing 2.3: Das AgeStatistics-Hilfsobjekt

2.7. Controller

Nachdem im Router definiert wurde, welche URLs auf welche Controller-Aktionen abgebildet werden sollen, werden die betroffenen Controller-Aktionen nun implementiert. Die zuvor genannte Aktion `controllers.Application.index` befindet sich in der Datei `app/controllers/Application.scala`. Die Standardimplementierung wird wie in Listing 2.4 zu sehen, umdefiniert.

```
object Application extends Controller {
  def index = Action {
    Ok(views.html.index(AgeStatistics.sample))
  }
}
```

Listing 2.4: Der Application-Controller mit index-Aktion

Controller sind Objekte, die von `Controller` erben und Aktionen definieren. Aktionen, die beim Aufruf einer URL ausgeführt werden sollen, sind Controller-Objekt-Methoden mit dem Rückgabetyp `Action`. Eine Aktion, bzw. `Action` ist eine Funktion von einer HTTP-Anfrage nach einer HTTP-Antwort. In der obigen Form wird die HTTP-Anfrage ignoriert, es können aber auch `Actions` mit explizitem oder implizitem Request-Parameter erstellt werden (vgl. Zenexity und Typesafe Inc. 2013a). Dies ist z. B. dann notwendig, wenn Formulare verarbeitet werden sollen, was in Unterabschnitt 2.9.3 (Dynamische Statistiken und Formularverarbeitung im Controller) zu sehen ist.

Um eine HTTP-Antwort zu generieren, stellt das Objekt `play.api.mvc.Results`, von dem `play.api.mvc.Controller` erbt, mehrere Konstruktoren zur Verfügung. Der in Listing 2.4 verwendete Konstruktor `Ok` erstellt eine HTTP-Antwort mit dem Status-Code `200 OK`. Der Inhalt dieser Antwort ist das Ergebnis der View `views.html.index`, worauf im folgenden Abschnitt näher eingegangen wird. Neben `Ok` gibt es u. a. noch `BadRequest` für fehlerhafte Anfragen (z. B. unvollständiges Formular) und `Redirect`, um eine Seitenweiterleitung auf eine angegebene URL durchzuführen (vgl. Zenexity und Typesafe Inc. 2013a).

2.8. View

Im vorigen Abschnitt wurde gezeigt, wie mittels `Ok(views.html.index(ageStatistics))` eine View gerendert und als HTTP-Antwort verschickt werden kann. Das dazugehörige View-Template befindet sich unter `app/views/index.scala.html`. Bevor die Implementierung dieses Templates gezeigt wird, soll an einem einfacheren Beispiel verdeutlicht werden, wie View-Templates geschrieben werden.

2.8.1. Views in Play

```
1 @(title: String)
2
3 @import scala.math.pow
4
5 <!doctype html>
6 <meta charset="utf-8">
7 <title>@title</title>
8 <p>4^3 = @{pow(4, 3)}</p>
9 @Html("<p>This␣is␣a␣simple␣view␣template.</p>")
```

Listing 2.5: Ein einfaches View-Template

Anhand des in Listing 2.5 gezeigten Codes sollen Struktur und Verwendung von View-Templates in Play veranschaulicht werden. View-Templates bestehen aus Scala- und HTML-Code und verhalten sich wie Funktionen, die HTML-Code generieren. Am Anfang eines Templates steht die Parameterliste, worüber die anzuzeigenden Daten übergeben werden. Nach der Parameterliste können Pakete importiert werden. Das `@`-Symbol führt einen Scala-Ausdruck an, der in geschweifte Klammern (`{}`) eingeschlossen werden kann (vgl. Zenexity und Typesafe Inc. 2013p).

Neben den Template-Parametern und den importierten Paketen können sog. Helper verwendet werden. Helper sind in Funktionen ausgelagerter Template-Code (vgl. Hilton u. a. 2013, S. 179). Play kodiert aus Sicherheitsgründen Scala-Strings automatisch so, dass dadurch kein HTML-Code generiert werden kann. Um dies zu verhindern, kann der `Html`-Helper verwendet werden, wie in der letzten Zeile von Listing 2.5 zu erkennen. Dieser Helper nimmt als Argu-

ment einen String und fügt diesen unverändert an der Stelle des Aufrufs im Template ein (vgl. Zenexity und Typesafe Inc. 2013p).

2.8.2. Altersstatistiken-View

Das eigentliche View-Template unter `app/views/index.scala.html` ist etwas umfangreicher. Das Zeichnen des Diagramms erfolgt cientseitig via JS und ist in `public/javascripts/main.js` ausgelagert. Dieses Script wird wie auch alle anderen Dateien im `public/`-Ordner via `@routes.Assets.at("javascripts/main.js")` adressiert (vgl. Hilton u. a. 2013, S. 111). Das gesamte Template hat den in Listing 2.6 vereinfacht dargestellten Inhalt.

```
@(statistics: AgeStatistics)
<script src="@routes.Assets.at("javascripts/main.js")"></script>
<div id="ageChart"><svg></svg></div>
<script>
  var statistics = @Html(Json.toJson(statistics.map {
    case (k, v) => (k.toString, v)
  }).toString);
  makeAgeStatisticsChart("#ageChart␣svg", statistics);
</script>
```

Listing 2.6: Das View-Template

`makeAgeStatisticsChart` erwartet neben dem CSS-Selector, der das Diagramm-Element identifiziert, die Altersstatistiken, die auf Server-Seite als `Map` vorliegen. Diese werden auf der JS-Seite als Objekt erwartet, weil JS keine `Maps` kennt, aber Objekte eine ähnliche Funktionalität bieten. Weil JS-Objekte die Schlüsselwerte als `String` erwarten, müssen die Schlüssel serverseitig erst in `Strings` konvertiert werden. Anschließend kann mittels `Json.toJson` die `Map` in ein JS-Objekt konvertiert und mit `toString` und `Html` in das Template eingefügt werden. Das `play.api.libs.json`-Paket enthält verschiedene Werkzeuge, um Scala-Werte in eine JS-kompatible Darstellung zu konvertieren und umgekehrt. `Json.toJson` ist ein besonders einfacher Weg diese Konvertierung durchzuführen und funktioniert ohne weiteres Zu-Tun für unterschiedliche Scala-Typen, darunter auch `Map[String, Int]` (vgl. Hilton u. a. 2013, S. 214–215). Wenn die `statistics`-Variable serverseitig `Map(51 -> 3, 16 -> 5, 10 -> 2)` enthält, so wird dies für die Client-Seite in `{"51":3,"16":5,"10":2}` konvertiert. Aus diesen konvertierten Daten wird dann das Diagramm erstellt, das in Abb. 2.2 zu sehen ist.

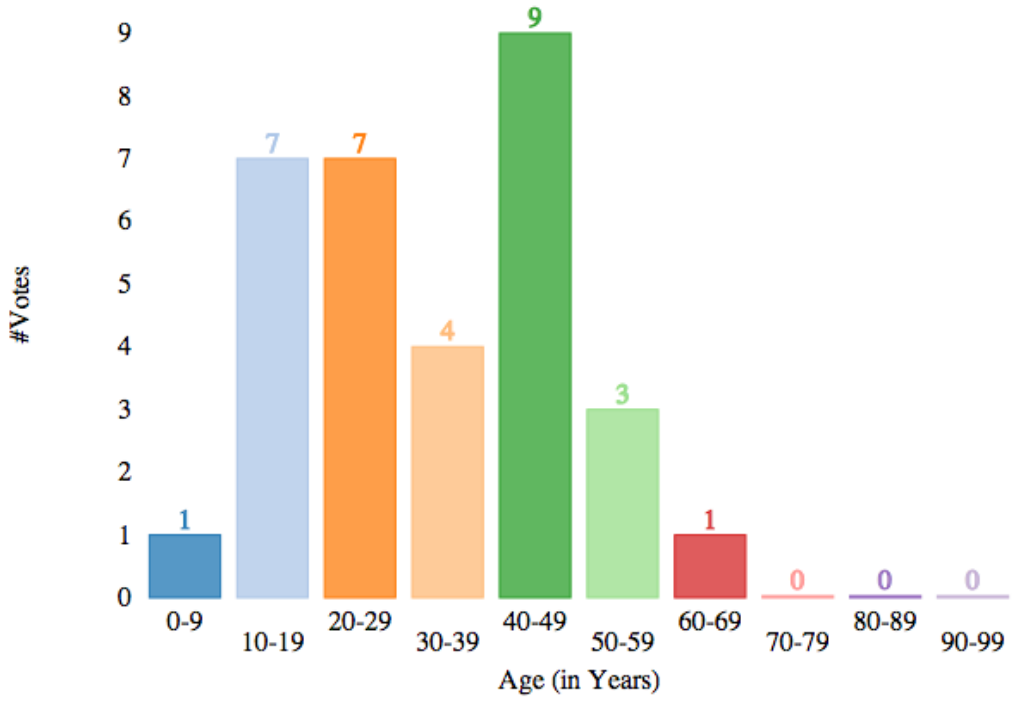

Abbildung 2.2.: Die Altersstatistiken-View

2.9. Sammeln der Statistiken

Die Nutzer sollen die Möglichkeit erhalten, der Website mitzuteilen, wie alt sie sind, um in die Statistik einzugehen. Um das Beispiel einfach zu halten, soll nicht geprüft werden, ob jemand bereits sein Alter angegeben hat. Um die Anwendung um Nutzer-Interaktion zu erweitern, müssen View, Routen-Datei und Controller erweitert werden.

2.9.1. Formular in der View

Um die Nutzer nach ihrem Alter zu fragen, benötig es erst einmal ein Formular, über das sie ihre Daten angeben können. Dieses Formular ist der View schnell mit einigen Zeilen HTML hinzugefügt, wie in Listing 2.7 zu sehen. Der Einfachheit halber werden nur Altersangaben zwischen einem und 99 Jahren akzeptiert.

```
1 <form method="post" action="@routes.Application.input">
2   <fieldset>
3     <legend>Your Age</legend>
```

```
4       <div>
5         <label for="ageInput">How Old Are You?</label>
6         <input type="number" min="1" max="99" id="ageInput"
      name="age" placeholder="Enter␣Age">
7       </div>
8       <button type="submit">Submit</button>
9     </fieldset>
10  </form>
```

Listing 2.7: Das View-Template mit Formular

Der im obigen Listing zu sehende Code `@routes.Application.input` ist ein Beispiel von sog. Reverse Routing. Dabei wird die URL für einen Routen-Eintrag in der `routes`-Datei berechnet und muss deshalb bei Änderungen nicht auch in der View geändert werden (vgl. Hilton u. a. 2013, S. 98–100). In diesem Fall wird die URL für die Controller-`Action`, die für die Formularverarbeitung zuständig ist von einem Helper generiert. Der hier verwendete Routing-Eintrag wird im folgenden Unterabschnitt angelegt.

2.9.2. Eintrag in der Routen-Datei

Wenn dieses Formular abgesendet wird, wird nicht wie zuvor ein `GET`-Request an den Server gesendet, sondern ein `POST`-Request, wie im `method`-Attribut des `form`-Tags angegeben. In der `conf/routes`-Datei muss deshalb ein neuer Eintrag hinzugefügt werden, der `POST`-Requests abdeckt und an den Controller weiterleitet, wie in Listing 2.8 zu sehen.

```
1 POST    /     controllers.Application.input
```

Listing 2.8: Die Routen-Eintrag für Formulareingaben

2.9.3. Dynamische Statistiken und Formularverarbeitung im Controller

Damit die Statistiken aktualisiert werden können, wird der Einfachheit halber direkt im `Application`-Controller mit `var ageStatistics = AgeStatistics.empty` eine Variable eingeführt, die den aktuellen Wert der Statistiken enthält. Es wäre auch möglich, ein Model anzulegen, welches die aktuelle Statistik hält, doch weil in diesem Beispiel nur der `Application`-Controller auf die Statistik zugreift, ist es auch möglich, die aktuelle Statistik direkt im Controller zu hinterlegen. Die Controller-`Actions` arbeiten nun nur noch mit diesem Wert, anstatt mit `AgeStatistics.sample`, wie es vorher der Fall war.

Der Routen-Eintrag für das Formular definiert, dass `POST`-Requests auf den Pfad `/` an die `input`-Action des `controllers.Application`-Controllers weitergeleitet werden sollen. Diese Action wird definiert, wie in Listing 2.9 zu sehen.

```
1  val ageForm = Form("age" -> number(1, 99))
2
3  def input = Action { implicit request =>
4    ageForm.bindFromRequest.fold(
5      invalidForm => BadRequest(invalidForm.errorsAsJson.toString),
6      { case (age) =>
7        ageStatistics =
8          ageStatistics.updated(age, ageStatistics(age) + 1)
9        Redirect(routes.Application.index)
10     }
11   )
12 }
```

Listing 2.9: Formularverarbeitung im Controller

Das obige Listing beginnt mit der Definition der serverseitigen Formulardarstellung. `ageForm` enthält die Formulardefinition für die Altersangabe. `play.api.data.Form.apply` erstellt aus einem `play.api.data.Mapping` ein `play.api.data.Form`-Objekt. Ein `Mapping` bildet ein oder mehrere Formularfelder auf einen Wert mit assoziiertem Datentyp ab. Das `play.api.data.Forms`-Hilfsobjekt enthält u. a. die Methode `tuple`, die aus Paaren von Formularfeldname und `Mapping` ein einziges `Mapping`-Objekt erstellt. `tuple("name" -> text, "age" -> number)` würde ein `Mapping` für zwei Formularfelder erstellen, das beide Werte auf ein Tupel abbildet. `text` und `number` sind vordefinierte `Mappings` des `Forms`-Hilfsobjekts, die Zeichenketten, bzw. Zahlen erwarten (vgl. Hilton u. a. 2013, S 174–175). Für den Fall dass nur ein Formularfeld erwartet wird, gibt es eine weitere Variante von `Form.apply`, die einen Formularfeldnamen und ein `Mapping` erwartet, welche in Listing 2.9 verwendet wird. Dem verwendeten `number`-Mapping werden außerdem noch Mindest- und Höchstwert mitgeteilt, damit nur glaubwürdige Werte in die Statistik gelangen.

Die `input`-Action benötigt einen `Request`-Parameter, um die Formulardaten lesen zu können. `ageForm.bindFromRequest` füllt die serverseitige Darstellung des Formulars über den impliziten `Request`-Parameter mit Daten, womit mittels der `fold`-Methode dann weitergearbeitet werden kann (vgl. Hilton u. a. 2013, S. 179). `fold` nimmt als Argumente zwei Funktionen. Die erste Funktion wird ausgeführt, wenn die Formulardaten nicht auf die interne Formulardarstellung abgebildet werden konnten. Die zweite Funktion wird ausgeführt, wenn es keine Fehler

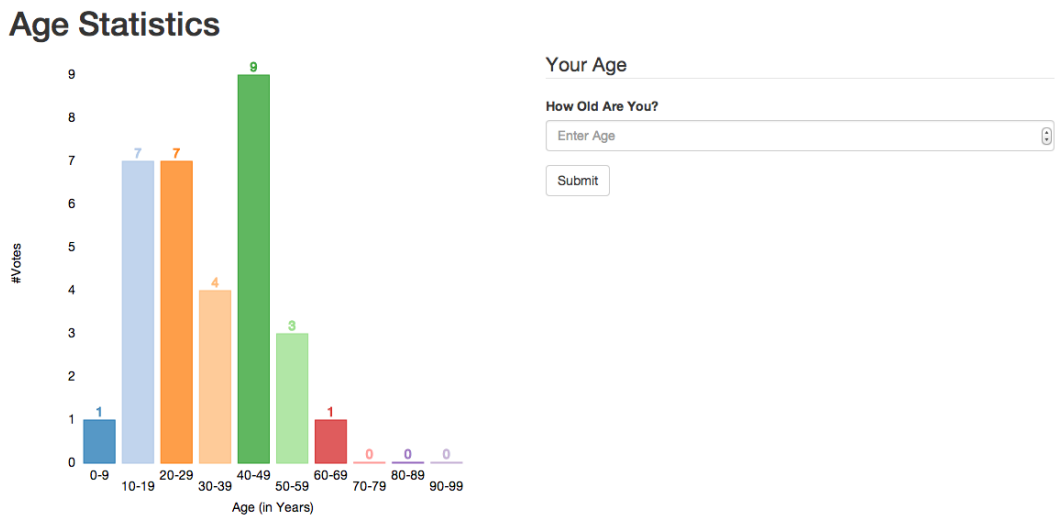

Abbildung 2.3.: Die Altersstatistiken-View mit Formular

gab, dann kann mit den übergebenen Formulardaten weitergearbeitet werden (vgl. Hilton u. a. 2013, S. 176).

In der `input`-Action wird im Fehlerfall eine Fehlermeldung angezeigt, die zum Zwecke dieses Beispiels nur ein einfacher JSON (JavaScript Object Notation)-String ist. Der Fehlerfall kann eintreten, obwohl in der View Minimum und Maximum des Altersfeldes angegeben sind, wenn z. B. der Nutzer die Prüfung aus dem ausgelieferten HTML-Code entfernt. Tritt kein Fehler auf, so werden die Altersstatistiken aktualisiert und eine Weiterleitung auf die `index`-Action ausgelöst. Dadurch landet der Nutzer/die Nutzerin nach Versendung des Formulars wieder auf der Hauptseite und sieht die aktualisierte Statistik. Die finale Version der Website ist in Abb. 2.3 zu sehen.

3. Reaktive Programmierung

Reaktive Programmierung ist die Programmierung von reaktiven Systemen. Ein reaktives System ist ein System, das von seiner Umgebung kontinuierlich Daten empfängt und darauf reagiert. In nebenläufigen Systemen können gleich mehrere solcher Datenströme unabhängig voneinander existieren (vgl. Pucella 1998, S. 1). Reaktive Prozesse haben die Eigenschaft, schrittweise, immer wenn sie neue Daten empfangen, ein Ergebnis aufzubauen. Deshalb sind sie bei Berechnungen essentiell, bei denen nicht nur das Endergebnis wichtig ist, sondern auch die Schritte, die dazu geführt haben (vgl. Chen 2000, S. 2).

In diesem Kapitel werden mit Monaden und speziell der `Future`-Monade die Grundbausteine der reaktiven Programmierung mit Play vorgestellt. Anschließend daran wird Plays Iteratee-Bibliothek vorgestellt, die die zuvor vorgestellten Konzepte nutzt, um eine Stream-Implementierung anzubieten, die zugleich effizient und flexibel ist.

3.1. Monaden

Monaden sind Container-artige Strukturen, die einige Operationen definieren, um auf den gekapselten Daten zu arbeiten. Das Konzept der Monaden stammt ursprünglich aus der Mathematik, genauer der Kategorientheorie. Moggi 1989 nutzte Monaden für Beweise über Programmen, worauf Wadler 1990 aufbaute, um Monaden zur Programmierung zu verwenden. In diesem Abschnitt sollen die mathematischen Hintergründe von Monaden kurz angerissen werden, um diese anschließend im Zusammenhang mit Scala näher zu erläutern.

3.1.1. Monaden in der Kategorientheorie

Eine Monade in einer Kategorie C ist ein Tripel $T = (T, \eta, \mu)$, wobei T ein Funktor $T : C \to C$ ist und η und μ natürliche Transformationen mit $\eta : Id_C \to T$ und $\mu : T^2 \to T$ sind (vgl. Mac Lane 1998, S. 137, Moggi 1989, S. 2).

Um diese Definition in den Kontext der Programmierung zu übersetzen, soll folgende vereinfachte Erklärung dienen (vgl. Wadler 1990, S. 2–3): Eine Kategorie in der Mathematik ist, sehr vereinfacht dargestellt, eine Menge von Objekten und den Operationen auf diesen

Objekten (vgl. Mac Lane 1998, S. 7). Als die Kategorie C kann die Programmiersprache selber, mitsamt ihrer Typen und den Operationen auf den Werten dieser Typen gesehen werden(vgl. Barr und Wells 1999, S. 6–8). Ein Funktor T ist eine Struktur mit einem Operator, der Funktionen $A \to B$ zu Funktionen $TA \to TB$ transformiert. Natürliche Transformationen können als Funktionen aufgefasst werden. η hebt einen Wert in die Monade und μ macht aus einer verschachtelten Monade ein nicht-verschachtelte Monade.

Um diese Definition verständlicher zu machen, soll sie in Scala-Code übersetzt werden. Wird T nach `map`, η nach `unit` und μ nach `join` umbenannt, so ergeben sich die in Listing 3.1 gezeigten Typen für eine eine Monade `M`.

```
1 def unit[A](a: A): M[A]
2 def map[A, B](f: A => B): M[A] => M[B]
3 def join[A](mm: M[M[A]]): M[A]
```

Listing 3.1: Die Monaden-Operationen der Kategorientheorie

Eine Monade ist eine Art Container mit drei assoziierten Operationen. `unit` hebt einen einen Wert in die Monade. `map` transformiert einen Wert innerhalb der Monade. `join` verringert die Tiefe von verschachtelten Monaden um eins.

3.1.2. Kleisli-Tripel in der Kategorientheorie

Ein Kleisli-Tripel über einer Kategorie C ist $T = (T, \eta, \_^*)$, wobei $T : Obj(C) \to Obj(C)$, $\eta_A : A \to TA$ und * ein Operator ist, der Morphismen der Art $f : A \to TB$ nach $f^* : TA \to TB$ umformt (vgl. Moggi 1989, S. 2).

Eine vereinfachte Erklärung für den Kontext der Programmierung ist folgende: T ist eine Abbildung von einem Objekt der Kategorie C auf ein anderes. η und * können wieder als Funktionen aufgefasst werden. η hebt einen Wert in den Tripel und * lässt eine Funktion, die auf Werten arbeitet, die keine Tripel sind, auf Tripeln arbeiten.

Kleisli-Tripel entsprechen Monaden. Aus jedem Kleisli-Tripel $(T, \eta, \_^*)$ lässt sich eine Monade (T, η, μ) ableiten und umgekehrt, wobei $T(f : A \to B) = (\eta_B \circ f)^*$ und $\mu_A = id^*_{TA}$ (vgl. Moggi 1989, S. 2).

In Scala-Code könnten die Operationen aussehen, wie in Listing 3.2 gezeigt. Dabei sind wieder T nach `map`, η nach `unit` und * nach `bind` umbenannt worden.

```
1 def unit[A](a: A): M[A]
2 def bind[A, B](f: A => M[B]): M[A] => M[B]
```

Listing 3.2: Die Kleisli-Tripel-Operationen der Kategorientheorie

3.1.3. Monaden in Scala

Wie oben beschrieben, lassen sich die monadischen Semantiken sowohl mit Hilfe von Monaden, als auch mit Hilfe von Kleisli-Tripeln erreichen. Scala hat Monaden via `for`-Comprehensions fest in die Sprache integriert und nutzt dafür sowohl Operationen des Kleisli-Tripels, als auch der klassischen Monade. In der Scala-Standardbibliothek existiert allerdings kein allgemeines Trait für Monaden. Um einen Datentyp innerhalb von `for`-Comprehensions nutzen zu können, muss dieser die zwei Methoden `map` und `flatMap` mit den Typen aus Listing 3.3 implementieren.

```
class M[A] {
  def map[B](f: A => B): M[B] = ???
  def flatMap[B](f: A => M[B]): M[B] = ???
}
```

Listing 3.3: Die Monaden-Operationen in Scala

Die `map`-Methode entspricht hierbei der `map`-Funktion des vorletzten Unterabschnitts und die `flatMap`-Methode entspricht der `bind`-Funktion des letzten Unterabschnitts. Für die `unit`-Funktion gibt es in Scala kein standardisiertes Pendant. Stattdessen wird der jeweilige Konstruktor der Monade oder die Konstruktormethode des Companion-Objekts verwendet, sofern diese existieren. Auch wenn Scalas `for`-Comprehensions sie nicht verlangen, so haben die Implementationen der Monaden in der Standardbibliothek i. d. R. auch die Methode `flatten`, die der als `join` vorgestellten Funktion entspricht. Bevor weiter auf `for`-Comprehensions eingegangen wird, sollen einige Beispiele gezeigt werden, um das Konzept zu verdeutlichen.

3.1.4. Beispiele für Monaden

Nachdem bisher nur theoretische Konzepte vorgestellt wurden, sollen an dieser Stelle einige Beispiele für Monaden aufgezeigt werden. Monaden finden sich in vielen bekannten Datentypen wieder, wie z. B. `Option` oder `List`.

Die Option-Monade

`Option` ist ein Datentyp, um optionale Daten darzustellen. Optionale Daten sind Daten, die vielleicht nicht vorhanden sind und würden in Sprachen, wie z. B. Java als `null` dargestellt werden (vgl. Odersky u. a. 2010, S. 289). `Some` entspricht dabei `unit` und hebt einen Wert in die Monade, wohingegen `None` ein Objekt ohne Wert erstellt. Mit `map` und `flatMap` kann ein Wert, der mit `Some` in die Monade gehoben wurde, verändert werden.

Wenn im Kontext der Altersstatistiken-Anwendung zwei Altersgruppen zusammengezählt werden sollen, könnte dies aussehen, wie in Listing 3.4. Dabei soll für dieses Beispiel ignoriert

werden, dass alle Statistiken einen Standardwert von null (0) für unbekannte Altersangaben zurückgeben. Dazu werden beide Altersgruppen mit `get` ausgelesen, wobei `get` das `get` von `scala.collection.Map` ist und das Ergebnis als `Option` zurückliefert. Anschließend werden beide Ergebnisse mit Hilfe der Monaden-Operationen kombiniert.

```
1 val statistics: AgeStatistics = AgeStatistics(20 -> 4, 30 -> 17)
2
3 val twentyYearOldCount: Option[Int] = statistics.get(20)
4 val thirtyYearOldCount: Option[Int] = statistics.get(30)
5
6 val twentyAndThirtyYearOldCount: Option[Int] =
7   twentyYearOldCount.flatMap(twenties =>
8     thirtyYearOldCount.map(thirties => twenties + thirties))
9 // twentyAndThirtyYearOldCount hat den Wert Some(21)
```

Listing 3.4: Die Option-Monade

Die List-Monade

Eine weitere verbreitete Monade ist `List`. Im Gegensatz zur oben vorgestellten `Option`-Monade ist `List` ein Container, der mehrere Elemente beinhalten kann. Mit `List.apply` lässt sich eine Liste mit einem Element erstellen und entspricht damit der `unit`-Funktion. `apply` nimmt zwar beliebig viele Elemente, doch diese Eigenschaft ist hierbei zu vernachlässigen.

Die Methoden `map` und `flatMap` operieren im Falle von `List` i. d. R. auf mehreren Elementen. Nur im Falle, dass die Liste leer ist, haben `map` und `flatMap` keinen Effekt. Ist die Liste nicht leer, so wendet `map` die übergebene Funktion auf alle Elemente der Liste an. Bei `flatMap` wird die Funktion ebenfalls auf alle Elemente angewendet. Weil die übergebene Funktion allerdings immer eine neue Liste zurückgibt, werden diese anschließend wieder in eine Liste zusammengefügt.

3.1.5. for-Comprehensions

`for`-Comprehensions bieten eine alternative Schreibweise für die zuvor vorgestellten Monaden-Operationen. Durch die Notation mit `for`-Comprehensions lassen sich einige längere Ausdrücke, die diese Operationen verwenden, verständlicher ausdrücken, indem die Notation einer imperativen Kontrollstruktur ähnelt. `for`-Comprehensions haben die in Listing 3.5 gezeigte Form. In diesem Listing, werden, wie schon im Beispiel der `Option`-Monade, Werte aus den Monaden `twentyYearOldCount` und `thirtyYearOldCount` entnommen und anschließend neu zusammengesetzt.

```
for {
  twenties <- twentyYearOldCount
  thirties <- thirtyYearOldCount
} yield twenties + thirties
```

Listing 3.5: Monaden-Operationen mit for-Comprehension

Ausdrücke der Form `x <- m` werden Generatoren genannt. Generatoren werden in `map`- und `flatMap`-Operationen übersetzt. Die ersten Generatoren innerhalb einer `for`-Comprehension werden zu `flatMap`s und der letzte Generator wird zu `map` (vgl. Odersky u. a. 2010, S. 490). Wird der Code aus Listing 3.5 nach diesen Regeln umgeformt, so ergibt sich wieder der ursprüngliche Code aus Listing 3.4.

3.1.6. Gesetze

Die hier vorgestellten Monaden-Operationen müssen einige Gesetze befolgen, um der mathematischen Definition gerecht zu werden. Diese Gesetze sind Moggi 1989 (S. 2) und O'Sullivan u. a. 2008 (S. 356–257) entnommen. Sie sollen für die Operationen des Kleisli-Tripels gegeben und anhand von Scala-Code verdeutlicht werden. Im Scala-Code ist η durch die Funktion `unit`, die ein Objekt der gewünschten Monade zurückliefert, und * durch die Methode `flatMap` ersetzt worden. In den folgenden Definition werden die Funktionen f und g verwendet, wobei $f : A \to TB$, $g : B \to TC$ und T das Kleisli-Tripel ist. Die jeweiligen beiden Seiten von $=$, bzw. = sind gleich.

1. linkes neutrales Element

   $f^* \circ \eta_A = f$

   `unit(x).flatMap(f) = f(x)`

   `unit` ist das linke neutrale Element von `flatMap`. Anstatt einen Wert mit `unit` in eine Monade zu heben, um anschließend darauf `flatMap` mit einer Funktion `f` aufzurufen, kann direkt `f` auf `x` angewendet werden.

2. rechtes neutrales Element

   $\eta^* = id_{TA}$

   `m.flatMap(unit) = m`

   `unit` ist das rechte neutrale Element von `flatMap`. `unit` darf den übergebenen Wert nicht verändern, weshalb es keine Wirkung hat, wenn es auf der rechten Seite von `flatMap` steht.

3. Assoziativität

   $g^* \circ f^* = (g^* \circ f)^*$

   ```
   m.flatMap(f).flatMap(g) = m.flatMap(x => f(x).flatMap(g))
   ```

   `flatMap` ist assoziativ. Es macht keinen Unterschied, ob das zweite `flatMap` auf dem Ergebnis des ersten aufgerufen wird, oder ob es innerhalb des ersten aufgerufen wird. Nur die Reihenfolge der Funktionsaufrufe von `f` und `g` muss übereinstimmen.

3.2. Futures und Promises

Ein `scala.concurrent.Future` ist ein Platzhalter für das Ergebnis einer nebenläufigen Berechnung und hält nach Beendigung dieser Berechnung dessen Ergebnis. Ein `Future` ist an ein `scala.concurrent.Promise` gekoppelt, welches einmalig den Wert des `Futures` setzen kann. `Futures` sind nur lesbar, wohingegen `Promises` einmalig schreibbar sind. Dieser Abschnitt behandelt diese beiden Datentypen und orientiert sich dabei an den Quellen Haller u. a. 2013 und Typesafe Inc. 2013a.

3.2.1. Futures

`Future` ist eine Monade, die das Ergebnis einer nebenläufigen Berechnung kapselt. Ein `Future` ein Datentyp, der entweder für ein erfolgreich berechnetes Ergebnis, für einen aufgetretenen Fehler, oder für eine noch andauernde Berechnung steht. Solange kein Fehler aufgetreten und auch die Berechnung noch nicht vollendet ist, ist auch das `Future` noch nicht vollendet und hält kein Ergebnis. Über den `Future`-Datentyp kann in nicht-blockierender Art und Weise auf dem berechneten Ergebnis operiert werden.

Futures ohne Promise erstellen

Um ohne `Promise` einen `Future`-Wert zu erstellen, stellt das `Future`-Companion-Objekt mehrere Konstruktormethoden bereit. `Future.successful` erstellt aus einem konkreten Wert einen bereits vollendeten `Future`. `Future.failed` erstellt aus einem `Throwable` einen `Future` im Fehlerzustand. `Future.apply` erstellt aus einer Berechnung (`=> A`) einen `Future` im erfolgreich vollendeten Zustand, wenn die Berechnung zu einem Ergebnis kommt oder einen `Future` im Fehlerzustand, wenn die Berechnung eine `Exception` wirft. Fatale `Exceptions` (`VirtualMachineError`, `NotImplementedError` u. a., vgl. École Polytechnique Fédérale de Lausanne 2013) werden im Thread, in dem `Future.apply` aufgerufen wurde allerdings wieder geworfen. Im Falle von `Future.apply` wird die Berechnung im Hintergrund ausgeführt, sodass

die Berechnung nicht die aktuellen Thread blockiert. Wodurch dieses Verhalten realisiert wird, wird in Unterabschnitt 3.2.3 beschrieben.

Futures verwenden

Wie alle Monaden in Scala, haben `Futures` die Methoden `map` und `flatMap` womit auf das jeweils berechnete Ergebnis zugegriffen werden kann. `def map[S](f: T => S): Future[S]` führt die übergebene Funktion aus, sobald die Berechnung des `Futures` zu einem Ergebnis gekommen ist. Führt die Berechnung zu keinem Ergebnis, weil ein Fehler aufgetreten ist, so wird auch die übergebene Funktion nicht ausgeführt. Mit `def flatMap[S](f: T => Future[S]): Future[S]` verhält es sich genauso. Keine dieser Methoden blockiert den aktuellen Thread, weil die übergebenen Funktionen im Hintergrund ausgeführt werden.

Um mehrere `Futures` zu kombinieren, bedeutet zu warten, bis alle `Futures` ihre Berechneten Werte halten, erst dann kann mit diesen Werten weitergearbeitet werden. Listing 3.6 zeigt, wie dies für zwei `Futures` umgesetzt werden könnte. Die Werte `f1` und `f2` könnten für die nächsten beiden eingehenden Altersangaben der Statistikenanwendung stehen.

```
1 val f1: Future[Int] = Future(35)
2 val f2: Future[Int] = Future(40)
3
4 val f1f2: Future[(Int, Int)] = for {
5   age1 <- f1
6   age2 <- f2
7 } yield (age1, age2)
8 // f1f2 hat den Wert Future((35, 40))
```

Listing 3.6: Futures kombinieren

Das `Future`-Trait und das dazugehörige Companion-Objekt bieten noch weitere Möglichkeiten, mit `Futures` zu arbeiten, sie zu kombinieren und v. a. Fehler zu behandeln. Dieser Unterabschnitt beschränkt sich allerdings auf die monadischen Operationen auf `Futures`.

3.2.2. Promises

`Promises` sind neben `Futures` der zweite Teil der `Futures`-API. Mit Hilfe von `Promises` können `Futures` einmalig Werte zugewiesen werden, wonach diese nicht mehr verändert werden können. Um `Promises` zu erstellen können im Companion-Objekt die drei Methoden `Promise.success`, `Promise.failure` und `Promise.apply` verwendet werden. `success` und `failure` nehmen einen konkreten Wert, bzw. einen `Throwable`-Wert, mit dem das assoziierte `Future`-Objekt vollendet wird. `apply` erstellt ein `Promise`, dessen `Future` noch keinen Wert besitzt.

Um dem `Future` eines `Promises`, das mit `Promise.apply` erstellt wurde, einen Wert zuzuweisen, können die Methoden `success`, bzw. `failure` auf dem `Promise` aufgerufen werden, die wieder einen konkreten Wert, bzw. ein `Throwable`-Objekt als Argument nehmen. Der Rückgabewert dieser beiden Methoden ist das `Promise`-Objekt, auf dem sie aufgerufen wurden. Wird ein zweites Mal versucht, dem assoziierten `Future` einen Wert zuzuweisen, wird eine `IllegalStateException` geworfen. Mit der Methode `future` kann auf das assoziierte `Future`-Objekt eines `Promises` zugegriffen werden. Listing 3.7 zeigt, wie ein Future erstellt wird, das nach 2 Sekunden mit einem `Int` erfüllt wird.

```
1  val promise: Promise[Int] = Promise()
2  val future: Future[Int] = promise.future
3
4  future.foreach(x => println(s"Die␣Zahl␣ist␣$x."))
5  // gibt nach 2 Sekunden "Die Zahl ist 42." aus
6
7  Future {
8    Thread.sleep(2000)
9    promise.success(42)
10 }
```

Listing 3.7: Future-Erstellung mit einem Promise

3.2.3. Execution Contexts

Die nebenläufigen Berechnungen, die von `Futures` ausgeführt werden, werden innerhalb von `scala.concurrent.ExecutionContexts` ausgeführt. Ein `ExecutionContext` ist vergleichbar mit einem Thread-Pool und ist dafür zuständig, dass an ihn delegierte Aufgaben im Hintergrund ausgeführt werden. Die zuvor vorgestellte Methode `Future.apply` nimmt neben der Berechnungsanweisung auch einen impliziten Parameter vom Typ `ExecutionContext`. Dieser `ExecutionContext` muss nicht per Hand erstellt werden. Mittels `import scala.concurrent.ExecutionContext.Implicits.global` kann ein vordefinierter `ExecutionContext` importiert werden.

3.3. Streams

Streams sind inkrementelle Datenströme, die nicht-blockierendes Lesen erlauben und kombiniert und transformiert werden können. Sie erlauben inkrementelle und funktionale Datenverarbeitung mit voller Kontrolle über den Ressourcenverbrauch. Anwendungsbeispiele für

Streams sind beispielsweise die Implementierung von Web-Servern oder Datenkompression. Die in diesem Kapitel vorgestellten Streams sind auch bekannt als Iteratee I/O. Sie entstanden im Umfeld der Programmiersprache Haskell und wurden von Oleg Kiselyov vorgestellt (vgl. Lato 2010, S. 19). Als Quellen hierfür dienen Kiselyov 2012b, Kiselyov 2012a, Lato 2010 und die Seiten der offiziellen Play-Dokumentation (Zenexity und Typesafe Inc. 2013g, Zenexity und Typesafe Inc. 2013f„ Zenexity und Typesafe Inc. 2013h, Zenexity und Typesafe Inc. 2013l).

3.3.1. Design von Streams

Iteratee IO-Streams bestehen aus den vier Komponenten `Input`, `Iteratee`, `Enumerator` und `Enumeratee`. `Inputs` beinhalten die gestreamten Elemente. `Iteratees` konsumieren `Input`-Elemente und dienen somit als Datensenken. `Enumerators` generieren `Input`-Elemente und dienen somit als Datenquellen. `Enumeratees` transformieren `Iteratees` und `Enumerators`. Diese Komponenten und ihr jeweiliger Verwendungszweck werden in den folgenden Unterabschnitten näher vorgestellt.

3.3.2. Design von Inputs

Die von der Datenquelle generierten Elemente werden in `Inputs` verpackt. Der Datentyp `Input` ist definiert, wie in Listing 3.8 gezeigt (vgl. Zenexity und Typesafe Inc. 2013m, Z. 239).

```
1 sealed trait Input[+E]
2 object Input {
3   case class El[+E](e: E) extends Input[E]
4   case object Empty extends Input[Nothing]
5   case object EOF extends Input[Nothing]
6 }
```

Listing 3.8: Der Input-Datentyp

Der Datentyp ist über den kovarianten Typ `E` der gehaltenen Elemente parametrisiert. Kovarianz eines Typs `E` ist in Scala am +-Prefix zu erkennen und bedeutet, dass wenn `F` ein spezialisierter Typ von `E` ist, `Input[F]` auch ein spezialisierter Typ von `Input[E]` ist (vgl. Odersky u. a. 2010, S. 393). Dadurch wären beispielsweise sowohl `Input[Cat]`, als auch `Input[Dog]` Spezialisierungen von `Input[Animal]`. Wäre `Input` nicht kovariant über den Typ der der gehaltenen Elemente, wären `Input[Cat]`, `Input[Dog]` und `Input[Animal]` völlig unterschiedliche Typen.

Des Weiteren ist anzumerken, dass `Input` ein algebraischer Datentyp ist. Ein algebraischer Datentyp ist ein Datentyp mit mehreren, alternativen Konstruktoren, die jeweils eigene Felder

besitzen. Mittels Pattern Matching kann i. d. R. von einem Wert eines solchen Datentyps auf den verwendeten Konstruktor geschlossen werden (vgl. Hudak u. a. 2007, S. 14–15).

In Scala werden algebraische Datentypen häufig mit Hilfe von einem Marker-Trait und `case`-Klassen und -Objekten implementiert. Das Marker-Trait ist der algebraische Datentyp und die `case`-Klassen und -Objekte sind die Konstruktoren. Die Nutzung von `case`-Klassen und -Objekten hat den Vorteil, dass diese mittels Pattern Matching zerlegt werden können. Dadurch, dass das Marker-Trait `sealed` ist, wird verhindert, dass von außerhalb der Quellcode-Datei weitere Konstruktoren hinzugefügt werden können (vgl. Eriksen 2012). In diesem Fall besitzt der algebraische Datentyp `Input` die Konstruktoren `El`, `Empty` und `EOF`. Der Konstruktor `El` besitzt ein Feld, `Empty` und `EOF` besitzen keine Felder.

Für den Fall, dass die Datenquelle nicht erschöpft ist, existiert im Companion-Objekt die Unterklasse `Input.El` mit dazugehöriger Konstruktormethode. Diese Implementierung hält genau ein Element aus der Datenquelle. Über die Entsprechung zu `Input.Empty` in Kiselyovs Implementierung schreibt er „[it] signifies a stream with no immediately available data but which is still continuing“ (vgl. Kiselyov 2012a). Außerdem findet es im Kontext von `Iteratees` Verwendung, um zu signalisieren, dass kein Teil der Eingabe übriggebliebenen ist. Wenn die Datenquelle erschöpft ist, wird dies mit dem Objekt `Input.EOF` mitgeteilt.

Sowohl `Input.Empty`, als auch `Input.EOF` sind vom Typ `Nothing`. `Nothing` ist ein besonderer Typ in Scala, ein so genannter Bottom-Typ. Das bedeutet, dass er ganz unten in der Klassenhierarchie steht und eine Spezialisierung jedes anderen Typs ist. Es gibt keine Instanz vom Typ `Nothing`, aufgrund seiner Position in der Klassenhierarchie kann er jedoch jeden anderen Typ annehmen (vgl. Odersky u. a. 2010, S. 216). Weil der Typparameter des `Input`-Datentyps kovariant ist, lassen sich `Input.Empty` und `Input.EOF` mit jedem anderen `Input` kombinieren.

Ein Wert vom Typ `Input` hat also drei mögliche Zustände:

1. Es gibt ein neues Element (`Input.El`).
2. Es gibt noch kein neues Element, doch die Datenquelle ist noch aktiv (`Input.Empty`)
3. Die Datenquelle ist erschöpft (`Input.EOF`).

3.3.3. Design von Iteratees

Die von der Datenquelle in `Inputs` verpackten Elemente werden von `Iteratees` konsumiert, um daraus ein Ergebnis aufzubauen. `Iteratees` kapseln ihren Zustand `Step`, der den Verarbeitungszustand widerspiegelt. Der Typ des `Step`-Zustands ist wie in Listing 3.9 gezeigt (vgl. Zenexity und Typesafe Inc. 2013m, Z. 271).

```
sealed trait Step[E, +A]
object Step {
  case class Done[+A, E](a: A, remaining: Input[E])
    extends Step[E, A]
  case class Cont[E, +A](k: Input[E] => Iteratee[E, A])
    extends Step[E, A]
  case class Error[E](msg: String, input: Input[E])
    extends Step[E, Nothing]
}
```

Listing 3.9: Der Step-Datentyp

Der algebraische Datentyp ist über die zwei Typen `E` und `A` parametrisiert. `E` ist der Typ der zu konsumierenden Elemente. `A` ist der Typ des zu berechnenden Ergebnisses. `A` ist kovariant, sodass auch Spezialisierungen von `A` als Werte verwendet werden können. Die Kovarianz über den Typ `A` ermöglicht es auch hier wieder `Nothing` in den Konstruktoren zu verwenden, wie im vorigen Unterabschnitt beschrieben.

Ein `Iteratee`, der mit seiner Berechnung fertig ist, ist im Zustand `Step.Done`. Ein solcher `Iteratee` hält sein berechnetes Ergebnis und den Teil der letzten Eingabe, der nicht mehr verarbeitet wurde. Ein `Iteratee` in diesem Zustand nimmt keine weiteren Elemente an. Häufig ist der Eingaberest `Input.Empty`, weil die gesamte Eingabe konsumiert wurde. Ein Beispiel für einen nicht leeren Eingaberest ist, wenn ein `Iteratee` zeilenweise Dateiinhalte empfängt und darin nach einem Wort sucht. Wenn das gesuchte Wort gefunden wurde, wird als Restwert der bisher nicht angesehene Rest der Zeile verwendet.

Ein `Iteratee`, der noch kein Endergebnis berechnet hat, ist im Zustand `Step.Cont`. Das Argument von `Step.Cont` ist die Schritt-Funktion `Input[E] => Iteratee[E, A]`. Die Eingabe der Schritt-Funktion ist ein `Input` mit einem neuen zu verarbeitenden Element. Die Ausgabe ist ein neuer `Iteratee`, der den neuen Berechnungsstand nach dem Verarbeiten des Eingabeelements hält. Diese Funktion wird als Continuation bezeichnet. Eine Delimited Continuation repräsentiert den Rest einer Berechnung bis zu einem bestimmten Punkt (vgl. Rompf u. a. 2009, S. 1). In diesem Fall berechnet die Continuation immer die Verarbeitung genau eines weiteren Elements.

Falls bei der Verarbeitung eines Elements ein Fehler auftritt, wird dies durch den Zustand `Step.Error` signalisiert. Ein solcher Fehler wird durch eine Fehlermeldung und durch das verursachende Eingabeelement dargestellt. Beispielsweise kann ein `Iteratee` nach dem Empfang eines `Input.EOF`-Elements signalisieren, dass es noch mehr Daten benötigt.

Ein `Iteratee` hat also einen Verarbeitungszustand vom Typ `Step` mit drei möglichen Zuständen:

1. Es wurde erfolgreich ein Ergebnis berechnet (`Step.Done`).
2. Es werden weitere Daten zur Berechnung benötigt (`Step.Cont`).
3. Es ist ein Fehler aufgetreten (`Step.Error`).

Außerdem ist anzumerken, dass der `Iteratee`-Typ eine Monade ist. Dadurch ist es möglich, einen beliebigen Wert in die Monade zu heben, die dann im `Step.Done`-Zustand ist. Auch kann ein erfolgreich berechneter Wert innerhalb der Monade transformiert werden. Mehr dazu in Unterabschnitt 3.3.8 (Iteratees sind Monaden).

3.3.4. Design von Enumerators

Ein `Enumerator` ist die Datenquelle, die ihre Daten, in `Inputs` verpackt, bereitstellt. Der Datentyp `Enumerator` ist definiert, wie in Listing 3.10 vereinfacht dargestellt (vgl. Zenexity und Typesafe Inc. 2013j).

```
1 trait Enumerator[E] {
2   def apply[A](i: Iteratee[E, A]): Future[Iteratee[E, A]]
3 }
```

Listing 3.10: Der Enumerator-Datentyp

Ein `Enumerator` ist eine Funktion von `Iteratee` nach `Iteratee` und ist über die Typen des übergebenen `Iteratees` parametrisiert. Der Eingabewert ist der `Iteratee`, an den die generierten Elemente gesendet werden. Der Rückgabewert ist der übergebene `Iteratee` nach Fertigstellung oder Abbruch seiner Berechnung. Der zurückgegebene `Iteratee` befindet sich in der `Future`-Monade.

Bei der Anwendung dieser Funktion werden dem `Iteratee` so lange Daten übergeben, bis die Datenquelle erschöpft ist oder bis der `Iteratee` keine Daten mehr annimmt. Die Herkunft der Daten, die der `Enumerator` an den `Iteratee` sendet, ist beliebig. Weil der Rückgabetyp sich in der `Future`-Monade befindet, können auch zeitintensive Berechnungen durchgeführt werden, ohne den Programmfluss zu unterbrechen. Dadurch ist es beispielsweise möglich, die Daten aus Dateien oder dem Netzwerk zu empfangen ohne zu blockieren.

3.3.5. Design von Enumeratees

Ein `Enumeratee` ist ein Stream-Transformator, der `Iteratees` eines bestimmten Element-Typs zu `Iteratees` eines anderen Element-Typs konvertiert. Der Datentyp `Enumeratee` ist definiert, wie in Listing 3.11 vereinfacht dargestellt (vgl. Zenexity und Typesafe Inc. 2013i).

```
trait Enumeratee[From, To] {
  def apply[A](inner: Iteratee[To, A]):
    Iteratee[From, Iteratee[To, A]]
}
```

Listing 3.11: Der Enumeratee-Datentyp

Ein `Enumeratee` ist eine Funktion von `Iteratee` nach `Iteratee`. Der Eingabewert ist der zu transformierende `Iteratee` vom inneren Element-Typ `To`. Der Ausgabewert ist ein `Iteratee` vom äußeren Element-Typ `From`. Der neue `Iteratee` nimmt Elemente vom Typ `From` an und transformiert sie nach `To`. Diese transformierten Werte werden dann an den ursprünglichen `Iteratee` weitergegeben.

Elemente vom Typ `From` heißen äußere Elemente, weil sie vom resultierenden `Iteratee` zuerst empfangen werden. Elemente vom Typ `To` heißen innere Elemente, weil sie zum `Iteratee` gehören, das sich im Rückgabetyp des resultierenden `Iteratees` befindet. Die Elemente des äußeren `Iteratees` werden nach der Transformation an den inneren `Iteratee` weitergereicht.

Die Transformation muss dabei aber nicht ein Element nach genau einem anderen Element abbilden. Ein äußeres Element `From` kann zu einem, keinem oder auch mehreren inneren Elementen `To` abgebildet werden. Genauso ist es möglich, dass mehrere äußere Elemente zu einem inneren Element zusammengefasst werden.

Ein `Enumeratee` kann auch als `Enumerator` betrachtet werden. Dies kommt daher, dass `Iteratees` Monaden sind. Ein `Enumeratee` ist ein `Enumerator`, dessen Ergebnis nicht in der `Future`-Monade, sondern in der `Iteratee`-Monade liegt. Jeder `Enumeratee` ist auch ein `Iteratee`, wenn man seinen Rückgabewert betrachtet, der vom Typ `Iteratee` ist. Ein `Enumeratee` ist also sowohl in der Rolle eines `Iteratees`, als auch in der Rolle eines `Enumerators`. Er ist ein `Iteratee` des äußeren Typs `From` und ein `Enumerator` des inneren Typs `To`, weil er Elemente des äußeren Typs konsumiert und Elemente des inneren Typs generiert.

3.3.6. Design von Komposition

Bei der Komposition werden zwei oder mehr `Iteratees` oder `Enumerators` zu einem neuen `Iteratee` oder `Enumerator` kombiniert. Es gibt hierbei zwei prinzipielle Arten der Komposition. Diese Arten sind die sequentielle und die parallele Komposition.

Komposition von Iteratees

Sequenzielle Komposition zweier `Iteratees` funktioniert nach folgendem Prinzip: Zuerst wird der erste `Iteratee` angewendet, bis er keine Elemente mehr annimmt oder die Datenquelle erschöpft ist. Dann wird der zweite `Iteratee` auf die noch übrigen Elemente der Datenquelle angewendet, bis auch dieser `Iteratee` keine Elemente mehr annimmt, oder die Datenquelle erschöpft ist. Anschließend werden die Ergebnisse beider `Iteratees` kombiniert, z. B. als Paar.

Bei der parallelen Komposition werden die Elemente der Datenquelle an beide `Iteratees` weitergegeben. Dies geschieht im Gegensatz zur sequentiellen Komposition allerdings ohne, dass zuerst ein `Iteratee` vollständig beendet sein muss. Es ist möglich `Iteratees` so zu kombinieren, dass der resultierende `Iteratee` eine Datenquelle auf mehrere Datensenken abbildet. Die Datensenken sind hierbei die kombinierten `Iteratees`. Beispielsweise können die Elemente der Datenquelle immer an beide `Iteratees` weitergegeben werden, um die Ergebnisse beider `Iteratees` anschließend als Paar zu zusammenzufassen.

Eine weitere Art der parallelen Komposition ist es, mehrere Datenquellen auf eine Datensenke abzubilden. Es wird hierbei ein `Iteratee` erstellt, der von mehreren Datenquellen liest und daraus ein Ergebnis berechnet. Dieser `Iteratee` ist ein verschachtelter `Iteratee`, d. h. das Ergebnis des äußeren `Iteratees` ist der innere `Iteratee`, der schließlich das zusammengesetzte Ergebnis zurückgibt (mehr dazu in Unterabschnitt 3.3.11). In Kiselyovs Implementierung ist es möglich, in beliebiger Reihenfolge aus den unterschiedlichen Datenquellen zu lesen. Nach dem Kenntnisstand des Autors ist dies in Plays Implementierung nicht möglich, weil diese nicht auf einer so hohen Abstraktionsebene arbeitet, wie Kiselyovs. Der Grund hierfür könnte zum einen sein, dass durch die Einführung einer weiteren Abstraktionsschicht das Verständnis der Iteratee-Implementierung erschwert würde. Zum anderen könnte es sein, dass dieses Feature im Kontext von Web-Anwendungen zu wenig Verwendung findet und der Implementierungsaufwand deshalb nicht gerechtfertigt ist.

Komposition von Enumerators

Die sequentielle Komposition zweier `Enumerators` erfolgt ähnlich, wie die sequentielle Kompositionen von `Iteratees`. Der resultierende `Enumerator` generiert erst alle Elemente des ersten `Enumerators` und dann alle Elemente des zweiten `Enumerators`. Die sequentielle Komposition von `Enumerators` entspricht also der Verkettung ihrer Ausgaben.

Bei der parallelen Komposition von `Enumerators` werden die Elemente der `Enumerators` nicht nacheinander, sondern durcheinander generiert. Es muss also nicht erst ein `Enumerator` erschöpft sein, bevor Daten generiert werden, die aus einem anderen `Enumerator` entstammen.

Eine Möglichkeit ist, dass die Reihenfolge der Elemente dadurch bestimmt wird, welcher `Enumerator` zuerst ein neues Element zur Verfügung stellt.

3.3.7. Anwendung von Streams

In den vorigen Unterabschnitten zum Design der `Iteratee`-Streams wurden die Grundkomponenten des Moduls vorgestellt. In den folgenden Unterabschnitt soll ihr Einsatz mit Hilfe von konkreten Code-Beispielen gezeigt werden. Es werden häufig mehrere Möglichkeiten für die Verwendung der einzelnen Komponenten erläutert, um die unterschiedlichen Abstraktionsschichten zu verdeutlichen. Die Codebeispiele stammen, sofern nicht anders vermerkt, vom Autor.

3.3.8. Anwendung von Iteratees

Plays `Iteratees` kapseln ihren `Step`-Zustand. Die elementare Methode dieses Traits ist `fold`. Die Signatur dazu ist in Listing 3.12 zu lesen (vgl. Zenexity und Typesafe Inc. 2013m, Z. 400).

```
1 def fold[B](folder: Step[E, A] => Future[B])
2   (implicit ec: ExecutionContext): Future[B]
```

Listing 3.12: Die Signatur von fold

Mit Hilfe von `fold` lässt sich der Zustand des `Iteratees` transformieren. Deshalb sind sehr viele Methoden des `Iteratee`-Traits auf Basis von `fold` definiert. Durch Übergabe eines `Step`-Werts an die `folder`-Funktion bestimmt die konkrete `Iteratee`-Implementierung, wie mit übergebenen Elementen umzugehen ist und ob überhaupt weitere Elemente akzeptiert werden. Nur durch die im `Step`-Wert befindliche Eingabeverarbeitung kann ein `Iteratee` neue Elemente empfangen. Dadurch wird nach und nach der zu berechnende Wert aufgebaut. Der implizite Parameter `ec` definiert den `ExecutionContext`, in dem die internen `Future`-Berechnungen ausgeführt werden sollen. `ExecutionContexts` wurden in Unterabschnitt 3.2.3 im Kontext von Futures und Promises behandelt.

Iteratees erstellen

Es gibt drei Möglichkeiten, einen neuen `Iteratee` zu erstellen.

1. Durch Erstellung einer neuen Klasse, die das `Iteratee`-Trait implementiert.
2. Durch Benutzung einer Konstruktormethode analog zu den `Step`-Zuständen (`Done`, `Cont` und `Error`).

3. Durch Benutzung einer Konstruktormethode des Companion-Objekts (`fold`, `foreach`, u. a.).

Zu Demonstrationszwecken soll im Folgenden ein `Iteratee` erstellt werden, der alle empfangenen Elemente aufsummiert. Es wird dabei nach jeder der drei möglichen Varianten implementiert.

Erstellung durch Vererbung

Die erste Variante ist, von `Iteratee` zu erben und die `fold`-Methode zu implementieren. Dies erfordert viel Schreibarbeit, weil nur die sehr generische `fold`-Methode verwendet werden kann. In Listing 3.13 ist die Implementierung eines Aufsummierungs-`Iteratees` zu sehen, der verwendet werden könnte, um das Gesamtalter der Altersstatistik aus Kap. 2 zu berechnen.

```
case class SumIteratee(sum: Int = 0) extends Iteratee[Int, Int] {
  def fold[B](folder: Step[Int, Int] => Future[B])
      (implicit ec: ExecutionContext): Future[B] = {
    folder(Step.Cont {
      case Input.El(i) => SumIteratee(sum + i)
      case Input.Empty => this
      case Input.EOF   => new Iteratee[Int, Int] {
        def fold[B](folder: Step[Int, Int] => Future[B])
            (implicit ec: ExecutionContext) = {
          folder(Step.Done(sum, Input.EOF))
        }
      }
    })
  }
}

val sumIterateeFromInheritance: Iteratee[Int, Int] =
   SumIteratee()
```

Listing 3.13: Erstellung eines Iteratees durch Vererbung

Der `folder`-Funktion muss der aktuelle `Step`-Zustand übergeben werden. Diese Funktion soll den `Iteratee` dann mit Elementen versorgen. Anschließend soll der `folder` dann aus dem finalen Zustand ein Ergebnis berechnen. Solange noch weitere Elemente kommen können, also solange kein `Input.EOF`-Element empfangen wurde, geht der `Iteratee` in einen neuen `Cont`-Zustand über. Bei diesem Zustandsübergang wird auch die interne Berechnung fortgesetzt. Es wird also in einen `Cont`-Zustand übergegangen, der die aktualisierte Summe hält. Wenn ein

leeres Element (`Input.Empty`) verarbeitet werden soll, wird der `Iteratee` unverändert zurückgegeben, weil sich an der Gesamtsumme nichts geändert hat. Sobald keine weiteren Elemente mehr kommen können, wird die Berechnung beendet und ein `Iteratee` an den `folder` übergeben, der sich immer im `Done`-Zustand befindet. Mit dem `ExecutionContext` muss nicht direkt gearbeitet werden, es muss nur dafür gesorgt werden, dass ein passender implizieter Wert erstellt oder importiert wurde (z. B. mittels `import scala.concurrent.ExecutionContext.Implicits.global`).

Erstellung durch Konstruktormethode

Die zweite Möglichkeit, einen `Iteratee` zu erstellen, ist mit Hilfe einer der Konstruktormethoden. Es gibt die Konstruktormethoden `Done`, `Cont` und `Error`, die jeweils einen `Iteratee` im gleichnamigen `Step`-Zustand erstellen. Die Methoden befinden sich direkt im `play.api.libs.iteratee`-Paket und sind eigentlich einfache Objekte mit einer `apply`-Methode, können aber wie Methoden verwendet werden. Jede diese Methoden nimmt auch die gleichen Argumente, wie ihr Pendant. Der Code, der einen Aufsummierungs-`Iteratee` mit Hilfe dieser Methoden erstellt, ist in Listing 3.14 zu sehen.

```
1 def sumIteratee(sum: Int = 0): Iteratee[Int, Int] = Cont {
2   case Input.El(i) => sumIteratee(sum + i)
3   case Input.Empty => sumIteratee(sum)
4   case Input.EOF   => Done(sum, Input.EOF)
5 }
6
7 val sumIterateeFromConstructor: Iteratee[Int, Int] =
    sumIteratee()
```

Listing 3.14: Erstellung eines Iteratees durch eine Konstruktormethode

Der Code ist dem in Erstellung eines Iteratees durch Vererbung vorgestellten Code sehr ähnlich. Im Gegensatz dazu muss hier jedoch nicht mehr explizit von `Iteratee` geerbt und die `fold`-Methode implementiert werden. Dies macht den Code schon wesentlich kürzer und einfacher. Der Code ist auf die elementaren Aufgaben reduziert und es ist nun klar zu erkennen, was bei welcher Art von `Input` geschieht.

Erstellung durch Konstruktormethode im Companion-Objekt

Die dritte und letzte Möglichkeit nutzt eine problemspezifische Konstruktormethode im Companion-Objekt. Das Aufsummieren einer Menge von Zahlen lässt sich sehr einfach mit Hilfe eines Folds implementieren. Das `Iteratee`-Companion-Objekt beinhaltet u. a. die `fold`-Konstruktormethode, die für solche Operationen gedacht ist. Durch Einsatz dieser Methode

wird die Lösung, wie in Listing 3.15 zu erkennen, zum Einzeiler. Es muss nicht mehr zwischen unterschliedlichen `Input`-Arten unterschieden werden, sondern es muss sich nur noch um tatsächliche Eingabeelemente gekümmert werden.

```
val sumIterateeFromHelper: Iteratee[Int, Int] =
  Iteratee.fold(0)(_ + _)
```

Listing 3.15: Erstellung eines Iteratees durch Konstruktormethode im Companion-Objekt

Eine weitere hilfreiche Konstruktormethode ist `getChunks`. Diese Methode gibt einen `Iteratee` zurück, der alle empfangenen Elemente in einer Liste sammelt. Der Typ dieses `Iteratees` ist entsprechend `Iteratee[E, List[E]]`. Dieser `Iteratee` wird im Folgenden zu Veranschaulichungszwecken an einigen Stellen dieser Arbeit verwendet.

Iteratees ausführen

Um die `Iteratees` mit Daten zu versorgen, existiert, wie in Iteratees erstellen beschrieben, die Methode `fold`. `fold` nimmt als Argument eine Funktion `folder` vom Typ `Step[E, A] => Future[B]`. Diese Funktion übergibt ein Element an den `Iteratee` und muss daraufhin den Zustand des neuen `Iteratees` überprüfen, ob weitere Elemente akzeptiert werden. Erst nach dieser Analyse kann entschieden werden, ob ein weiteres Element übergeben wird oder nicht. In Listing 3.16 wird gezeigt, wie dem zuvor erstellten `Iteratee` zur Berechnung des Gesamtalters, die einzelnen Altersangaben in Form von `Int`s zugeführt werden.

```
def folder(xs: Int*)(step: Step[Int, Int]): Future[Int] = {
  def folder_(xs: List[Int])(step: Step[Int, Int]): Future[Int] =
    xs match {
      case Nil => step match {
        case Step.Cont(k) => k(Input.EOF).fold {
          case Step.Done(sum, Input.EOF) => Future(sum)
          case _ => Future.failed(new Exception("invalid␣state"))
        }
        case _ => Future.failed(new Exception("invalid␣state"))
      }
      case x :: xs => step match {
        case Step.Cont(k) => k(Input.El(x)).fold(folder_(xs))
        case _ => Future.failed(new Exception("invalid␣state"))
      }
    }

  folder_(xs.toList)(step)
```

```
18 }
19
20 val sumResult: Future[Int] =
21   sumIterateeFromHelper.fold(folder(22, 25, 54))
22 // sumResult hat den Wert Future(101)
```

Listing 3.16: Ausführung eines Iteratees durch folder-Funktion

Das Übergeben von Daten mittels der `fold`-Methode ist, wie in Listing 3.16 zu erkennen, recht aufwändig. Dies kommt daher, dass zwei Zustände geprüft werden müssen. Zum Einen muss der Zustand der Eingabemenge geprüft werden, ob noch Elemente existieren, die an den `Iteratee` übergeben werden sollen. Zum Anderen muss sichergestellt werden, dass der `Iteratee` sich korrekt verhält. Ein `Iteratee` verhält sich korrekt, wenn er nach dem Empfang von `Input.EOF` keine weiteren Element annimmt, also in einen Endzustand übergeht. Nur wenn all diese Voraussetzungen erfüllt wurden, kann am Ende das Ergebnis dem `Iteratee` im `Step.Done`-Zustand entnommen werden. Durch Verwendung von `Enumerators` wird dies allerdings wesentlich einfacher.

Iteratees sind Monaden

`Iteratees` sind Monaden und haben deshalb die in Scala für Monaden üblichen Methoden `map` und `flatMap`. Um einen Wert in die Monade zu heben wird der Konstruktor `play.api.libs.iteratee.Done` verwendet, der dadurch erstellte `Iteratee` ist anschließend im `Step.Done`-Zustand. Mit `map` kann ein erfolgreich berechneter Wert, also ein Wert innerhalb eines `Iteratees` im `Step.Done`-Zustand verändert werden. Mit `flatMap` kann ebenfalls ein erfolgreich berechneter Wert verändert werden, doch mit dem Unterschied, dass die an `flatMap` übergebene Funktion einen neuen `Iteratee` zurückgibt, der aber nicht unbedingt auch im `Step.Done`-Zustand sein muss.

3.3.9. Anwendung von Enumerators

Mit Hilfe von `Enumerators` können auf einfache Weise Daten an `Iteratees` übergeben werden. In diesem Unterabschnitt werden Anwendung und Funktionsweise von `Enumerators` gezeigt. Es wird erklärt, auf welche Arten `Enumerators` erstellt werden können, und wie sie genutzt werden können, um Daten an `Iteratees` zu übergeben.

Enumerators erstellen

Es gibt drei unterschiedliche Techniken, einen `Enumerator` zu erstellen.

1. Durch Erstellung einer neuen Klasse, die das `Enumerator`-Trait implementiert.
2. Durch Benutzung einer Konstruktormethode im Companion-Objekt (`apply`, `generateM`, u. a.).
3. Durch Benutzung einer Konstruktormethode im `play.api.libs.iteratee.Concurrent`-Objekt (`broadcast`, `unicast`, u. a.).

Das o. g. `Concurrent`-Objekt stellt u. a. Konstruktoren für imperativ steuerbare `Enumerators` bereit. Im Folgenden werden alle Herangehensweisen vorgestellt, um Zahlen für den im vorigen Unterabschnitt vorgestellten Summierungs-`Iteratee` zu generieren.

Erstellung durch Vererbung

Bei der Erstellung eines `Enumerators` durch Vererbung, muss das `Enumerator`-Trait implementiert werden. Alles, was dafür nötig ist, ist die `apply`-Methode zu implementieren. Innerhalb dieser Methode wird der übergebene `Iteratee` mit Hilfe seiner `fold`-Methode befüllt, wie in Listing 3.17 zu sehen.

```
case class NumberEnumerator(xs: Int*) extends Enumerator[Int] {
  def apply[A](iteratee: Iteratee[Int, A]):
      Future[Iteratee[Int, A]] = {
    xs.foldLeft(Future(iteratee)) { (futureIteratee, x) =>
      futureIteratee.flatMap { iteratee =>
        iteratee.fold {
          case Step.Cont(k) => Future(k(Input.El(x)))
          case _ => Future(iteratee)
        }
      }
    }
  }
}

val numberEnumeratorFromInheritance: Enumerator[Int] =
  NumberEnumerator(22, 25, 54)
```

Listing 3.17: Erstellung eines Enumerators durch Vererbung

Alles was ein `Enumerator` zu tun hat, ist seine Daten an den übergebenen `Iteratee` zu übergeben. Dieses Vorgehen ist in diesem Beispiel als Fold über den `Iteratee` implementiert. Es wird immer ein Element an den `Iteratee` übergeben und der resultierende `Iteratee` als neues Zwischenergebnis verwendet. Falls der `Iteratee` keine weiteren Elemente akzeptiert,

obwohl noch Elemente übrig sind, wird dieser als Endergebnis verwendet. Das hier vorgestellte Verfahren ist in leicht abgewandelter Form der Implementierung von `Enumerator.apply` entnommen (vgl. Zenexity und Typesafe Inc. 2013j, Z. 611 und Z. 654).

Erstellung durch Konstruktormethode im Companion-Objekt

Die zweite Variante, einen `Enumerator` zu erstellen, ist mit Hilfe einer der Konstruktormethoden des Companion-Objekts. Hierfür gibt es mehrere hilfreiche Methoden, die an dieser Stelle aber nicht alle vorgestellt werden können. Stattdessen wird exemplarisch eine Methode vorgestellt, die auch in den folgenden Beispielen verwendet werden wird.

Die hier vorgestellte Methode ist die `apply`-Methode des Companion-Objekts mit der Signatur `def apply[E](in: E*): Enumerator[E]`. Diese Methode nimmt als Parameter beliebig viele Elemente eines Typs und wird diese bei Anwendung in `Input`-Elemente verpacken und an den `Iteratee` übergeben. Wie der `Enumerator` mit dieser Methode erstellt wird, ist in Listing 3.18 zu sehen.

```
1 val numberEnumeratorFromApply: Enumerator[Int] =
2   Enumerator(22, 25, 54)
```

Listing 3.18: Erstellung eines Enumerators durch die apply-Konstruktormethode

Erstellung durch Konstruktormethode im Concurrent-Objekt

Das Objekt `Concurrent` stellt weitere Konstruktoren für `Enumerators` bereit. Unter anderem beinhaltet dieses Objekt die Methode `broadcast`, die imperatives schreiben auf einen oder mehrere `Iteratees` erlaubt. `broadcast` nimmt keine Argumente und gibt `(Enumerator[E], Channel[E])` zurück. Mit Hilfe des `Channels` können imperativ Daten an den `Enumerator` übergeben werden. Ein `Channel` hat u. a. die Methoden `def push(item: E): Unit` und `def end(): Unit`. Diese Methoden erlauben es, ein Element an die verbundenen `Iteratees` zu übergeben, bzw. zu signalisieren, dass der `Enumerator` keine weiteren Elemente mehr hat.

Listing 3.19 zeigt, wie `Concurrent.broadcast` verwendet werden kann. Zu beachten ist hierbei, dass ein verbundener `Iteratee` nur die Elemente empfängt, die an den `Enumerator` übergeben wurden, nachdem der `Iteratee` mit dem `Enumerator` verbunden wurde. Alte Elemente werden vom `Enumerator` nicht für neue `Iteratees` gespeichert. Nach Übergabe aller Zahlen muss der `Channel` wieder geschlossen werden, andernfalls wird niemals signalisiert, dass keine weiteren Elemente mehr kommen und verbundene `Iteratees` werden nicht in den `Step.Done`-Zustand übergehen. In Unterabschnitt 4.2.3 im Real-Time-Web-Kapitel wird die hier beschriebene Methode verwendet, um mittels Server Sent Events die Altersangaben an alle verbundenen Clients zu verbreiten.

```
val (numberEnumeratorFromBroadcast, broadcastChannel) =
  Concurrent.broadcast[Int]

// Iteratee(s) mit numberEnumeratorFromBroadcast verbinden

broadcastChannel.push(22)
broadcastChannel.push(25)
broadcastChannel.push(54)
broadcastChannel.end()
```

Listing 3.19: Erstellung eines Enumerators durch die broadcast-Konstruktormethode

Anwendung auf Iteratees

`Enumerators` sind, wie in Unterabschnitt 3.3.4 erklärt, Funktionen. Das Ergebnis der Anwendung einer solchen `Enumerator`-Funktion ist der übergebene `Iteratee` nach Konsum der Elemente in der `Future`-Monade. Ein Beispiel für diese ersten Schritte ist in Listing 3.20 zu sehen.

```
val iteratee: Iteratee[Int, Int] = Iteratee.fold(0)(_ + _)
val enumerator: Enumerator[Int] = Enumerator(22, 25, 54)
val futureIterateeAfterApplication: Future[Iteratee[Int, Int]] =
  enumerator(iteratee)
```

Listing 3.20: Anwendung eines Enumerators auf einen Iteratee

Um mit dem neuen `Iteratee` weiterzuarbeiten wäre es möglich, alle weiteren Operationen innerhalb der `Future`-Monade durchzuführen. Weil es aber eine Indirektion bedeuten würde, `Iteratees` innerhalb von `Futures` zu bearbeiten, ist es möglich einen Wert vom Typ `Future[Iteratee[E, A]]` in einen Wert vom Typ `Iteratee[E, A]` zu transformieren. Beide Typen haben die gleiche Bedeutung, indem sie für ein Ergebnis stehen, das möglicherweise noch nicht vorhanden ist. Weil ein `Iteratee` innerhalb der `Future`-Monade aus diesem Betrachtungswinkel aber redundant ist, existiert die Methode `Iteratee.flatten`, um diese Transformation durchzuführen.

Nachdem ein `Enumerator` alle Elemente an einen `Iteratee` übergeben hat, sendet er allerdings kein `Input.EOF`-Element. Dadurch ist es möglich, mehrere `Enumerators` auf einen `Iteratee` anzuwenden, worauf in Unterabschnitt 3.3.11 näher eingegangen wird. Um ein `Input.EOF`-Element an einen `Iteratee` zu senden und anschließend das berechnete Ergebnis zu erhalten, wird die `run`-Methode aus dem `Iteratee`-Trait aufgerufen. Eine Weiterführung des zuvor begonnenen `Enumerator`-Beispiels ist in Listing 3.21 zu finden. In diesem Beispiel

werden die beiden oben beschriebenen Techniken angewendet, um die berechnete Summe aus dem `Iteratee` zu extrahieren.

```
val iterateeAfterApplication: Iteratee[Int, Int] =
  Iteratee.flatten(futureIterateeAfterApplication)

val futureResult: Future[Int] = iterateeAfterApplication.run
// futureResult hat den Wert Future(101)
```

Listing 3.21: Extrahierung des Ergebnisses aus einem Iteratee

Mit Hilfe der Methode `Enumerator.run` ist es außerdem möglich, `Enumerator`anwendung und Ergebnisextraktion zusammenzufassen. Diese Methode hat die Signatur `def run[A](i: Iteratee[E, A]): Future[A]`. Unter Einsatz dieser Methode wird die Verarbeitung im Aufsummierungsbeispiel, wie in Listing 3.22 zu sehen, auf einen Befehl verkürzt.

```
val futureResult2: Future[Int] = enumerator.run(iteratee)
// futureResult2 hat den Wert Future(101)
```

Listing 3.22: Anwendung eines Enumerators mit gleichzeitiger Ergebnisextrahierung

3.3.10. Anwendung von Enumeratees

`Enumeratees` sind Stream-Transformatoren, die `Iteratees` eines Typs zu `Iteratees` eines anderen Typs transformieren. Sie sind, wie in Unterabschnitt 3.3.5 erklärt, Funktionen von `Iteratee` nach `Iteratee`. Das Ergebnis des resultierenden `Iteratees` ist wiederum ein `Iteratee`. Und zwar ist es der ursprüngliche `Iteratee` nach Empfang der Elemente. Plays `Enumeratees` ermöglichen es allerdings auch `Enumerators` und andere `Enumeratees` zu transformieren. In diesem Unterabschnitt wird beschrieben, wie eigene `Enumeratees` erstellt und wie sie auf `Iteratees`, `Enumerators` und `Enumeratees` angewendet werden können.

Erstellung von Enumeratees

Es gibt drei Möglichkeiten, einen `Enumeratee` zu erstellen.

1. Durch Erstellung einer neuen Klasse, die das `Enumeratee`-Trait implementiert.
2. Durch Benutzung einer Konstruktormethode des Companion-Objekts (`map`, `filter`, u. a.)
3. Durch Benutzung einer Konstruktormethode des `Traversable`-Objekts aus dem `play.api.libs.iteratee`-Paket (`take`, `drop`, u. a.)

Im Folgenden werden die ersten beiden Möglichkeiten vorgestellt und mit Beispielen veranschaulicht. Die dritte Möglichkeit ist an dieser Stelle nur der Vollständigkeit halber aufgeführt, weil sie vom Autor als zu speziell erachtet wird. Es sei hier nur erwähnt, dass das `Traversable`-Objekt einige wenige Konstruktormethoden für `Enumeratees` bereitstellt, die auf Elementen vom Typ `TraversableLike` aus der Scala-Standardbibliothek arbeiten. Um die Altersstatistik zu schönen, sollen alle Altersangaben ab 50 um zehn Jahre verjüngt werden. In den Beispielen wird ein `Enumeratee` implementiert, der die eingehenden Zahlen auf die festgelegte Altersgrenze hin überprüft und gegebenenfalls angepasst an den inneren `Iteratee[Int, Int]` weiterleitet.

Erstellung durch Vererbung

Das `Enumeratee`-Trait verlangt, dass die Methode `applyOn` implementiert wird. Die Methode `apply` ist als Alias für `applyOn` definiert. `applyOn` hat die Signatur, die in Unterabschnitt 3.3.5 für `apply` vorgestellt wurde. Eine Mögliche Implementierung für einen `Enumeratee`, der die Zahlen eines `Iteratees` anpasst, ist in Listing 3.23 zu sehen.

```
case object RejuvinatingEnumeratee extends Enumeratee[Int, Int] {
  def applyOn[A](inner: Iteratee[Int, A]):
      Iteratee[Int, Iteratee[Int, A]] =
    Iteratee.flatten(inner.fold {
      case Step.Cont(k) => Future(Cont {
        case Input.El(number) if number >= 50 =>
          RejuvinatingEnumeratee(k(Input.El(number - 10)))
        case Input.El(number) =>
          RejuvinatingEnumeratee(k(Input.El(number)))
        case Input.Empty => RejuvinatingEnumeratee(k(Input.Empty))
        case Input.EOF => Done(Cont(k))
      })
      case _ => Future(Done(inner, Input.Empty))
    })
}

val enumerateeFromInheritance: Enumeratee[Int, Int] =
  RejuvinatingEnumeratee
```

Listing 3.23: Erstellung eines Enumeratees durch Vererbung

Die Implementierung dieses `Enumeratees` ist der von `Enumeratee.map` nachempfunden. Sie ist allerdings weniger abstrakt, als in der Referenzimplementierung (vgl. Zenexity und Typesafe Inc. 2013i, Z. 372, Z. 196 und Z. 85). Der `Enumeratee` prüft, ob der innere `Iteratee` weitere

Elemente annehmen kann. Kann er keine weiteren Elemente annehmen, wird er als Ergebnis zurückgegeben. Kann er Elemente annehmen, werden diese transformiert und anschließend weitergegeben. Sobald ein `Input.EOF` empfangen wird, wird in den Endzustand mit dem finalen inneren `Iteratee` als Ergebnis übergegangen.

Erstellung durch Konstruktormethode des Companion-Objekts

Die zweite hier vorgestellte Möglichkeit, einen `Enumeratee` zu erstellen ist mit Hilfe seines Companion-Objekts. Im Companion-Objekt finden sich mehrere Konstruktormethoden, die Listenoperationen sehr ähneln, wie z. B. `drop`, `filter`, `map` oder `zip`. Um alle Elemente des `Iteratees` zu transformieren bietet sich `map` an. Mit Hilfe dieser Methode werden alle Elemente durch eine angegebene Funktion transformiert, bevor sie an das innere `Iteratee` gegeben werden. `map` hat folgende Signatur (vgl. Zenexity und Typesafe Inc. 2013i, Z. 361):

```
1 trait Map[E] {
2   def apply[NE](f: E => NE)(implicit ec: ExecutionContext):
3     Enumeratee[E, NE]
4 }
5
6 def map[E]: Map[E]
```

Listing 3.24: Die Signatur von Enumeratee.map

Diese Signatur wirkt sperrig, lässt sich aber sinngemäß auf folgende Vereinfachung reduzieren:

```
1 def map[E, NE](f: E => NE): Enumeratee[E, NE]
```

Listing 3.25: Die vereinfachte Signatur von Enumeratee.map

Der wichtige Unterschied zwischen diesen beiden Signaturen ist, dass die erste nur über einen Typ `E` parametrisiert ist. Erst das zurückgegebene Objekt verlangt den zweiten Typ `NE`. Dadurch wird, so vermutet der Autor, Code, der diese Methode nutzt, kürzer, weil der Parameter `NE` in der Regel vom Compiler abgeleitet werden kann. So muss in aller Regel, sofern überhaupt nötig, nur noch der erste Typ-Parameter angegeben werden. Zu einem früheren Entwicklungsstand des Frameworks wurde die zweite Variante verwendet und wurde erst später zur aktuellen Signatur abgeändert (vgl. Zenexity und Typesafe Inc. 2011). Der implizite `ExecutionContext`-Parameter wird wieder für die interne Arbeit mit `Futures` benötigt.

Unter Einsatz dieser `map`-Methode ist die Transformation in einer einzigen Anweisung möglich. Der dazu nötige Code kann Listing 3.26 entnommen werden.

```
1 val enumerateeFromConstructor: Enumeratee[Int, Int] =
2   Enumeratee.map(x => if (x >= 50) x - 10 else x)
```

Listing 3.26: Erstellung eines Enumeratees durch die map-Konstruktormethode

Anwendung auf Iteratees

Für die Anwendung auf `Iteratees` eignet sich die `apply`-, bzw. die `applyOn`-Methode, die im Unterabschnitt Erstellung durch Vererbung implementiert wurde. Der Rückgabewert dieser Methode ist ein verschachtelter `Iteratee`, nämlich der äußere und der innere `Iteratee`. Dies ermöglicht es, nach Beendigung des äußeren `Iteratees` wieder den ursprünglichen `Iteratee` nach Empfang der transformierten Elemente zu erhalten. Für den Fall, dass der ursprüngliche `Iteratee` nicht wiederverwendet werden soll, gibt es die Methode `transform`. `transform` gibt nach Anwendung nur den tranfsormierten `Iteratee` zurück. Listing 3.27 zeigt, wie ein `Enumeratee` auf einen `Iteratee` angewendet werden und anschließend wieder mit dem ursprünglichen `Iteratee` weitergearbeitet werden kann.

```
1  val t: Enumeratee[Int, Int] =
2    Enumeratee.map(x => if (x >= 50) x - 10 else x)
3  val i: Iteratee[Int, List[Int]] = Iteratee.getChunks
4  val e: Enumerator[Int] = Enumerator(22, 54)
5
6  val transformedI: Iteratee[Int, Iteratee[Int, List[Int]]] = t(i)
7  val originalI: Iteratee[Int, List[Int]] =
8    Iteratee.flatten(e.run(transformedI))
9
10 val result: Future[List[Int]] = e.run(originalI)
11 // result hat den Wert Future(List(22, 44, 22, 54))
```

Listing 3.27: Enumerateeanwendung auf Iteratees

Der transformierende `Enumeratee` wird auf den konsumierenden `Iteratee` angewendet (Z. 6). Daraufhin werden alle generierten Elemente des `Enumerators` angepasst, bevor sie an den `Iteratee` gelangen. Der `Enumerator` sendet einmal die Elemente an den transformierten `Iteratee` (Z. 8) und einmal an den ursprünglichen `Iteratee` (Z. 10). Am Ergebnis ist zu erkennen, dass die Elemente im ersten Fall korrekt transformiert und im zweiten Fall unverändert übergeben wurden.

Anwendung auf Enumerators

Neben `Iteratees` können auch `Enumerators` transformiert werden. Dafür stellt das `Enumerator`-Trait die Methode `through` bereit. `through` nimmt als Argument einen `Enumeratee` und gibt als Rückgabewert einen neuen `Enumerator` zurück.

Anwendung auf Enumeratees

Enumeratees können auch auf andere Enumeratees angewendet werden, wodurch sich eine Kette von Transformationen abbilden lässt. Mit Hilfe der Methode compose oder dem Operator >><> lassen sich zwei Enumeratees verketten. Die Signatur von compose ist Listing 3.28 zu entnehmen.

```
def compose[To2](other: Enumeratee[To, To2]):
    Enumeratee[From,To2]
```

Listing 3.28: Die Signatur von compose

In Listing 3.29 ist ein Beispiel zu sehen, in dem zwei Enumeratees verkettet werden. Der erste Enumeratee sorgt dafür, dass Personen ab 70 gar nicht mehr in der Altersstatistik auftauchen, indem der Stream mit Hilfe von Enumeratee.filter auf die kleineren Zahlen reduziert wird. Der zweite Enumeratee transformiert die übriggebliebenen Elemente wie zuvor so, dass alle Personen ab 50 um 10 Jahre jünger gemacht werden. Anschließend werden beide Enumeratees kombiniert und auf einen Enumerator angewendet, sodass schon transformierte Elemente generiert werden. Im Ergebnis ist gut zu erkennen, dass der Iteratee nur die Altersangaben unter 50 unverändert empfängt.

```
val t1: Enumeratee[Int, Int] = Enumeratee.filter(_ < 70)
val t2: Enumeratee[Int, Int] =
  Enumeratee.map(x => if (x >= 50) x - 10 else x)
val t12: Enumeratee[Int, Int] = t1.compose(t2)

val e: Enumerator[Int] = Enumerator(22, 25, 54, 76)
val i: Iteratee[Int, List[Int]] = Iteratee.getChunks

val result: Future[List[Int]] = e.through(t12).run(i)
// result hat den Wert Future(List(22, 25, 44))
```

Listing 3.29: Enumerateeanwendung auf Enumeratees

3.3.11. Anwendung von Komposition

Die in Unterabschnitt 3.3.6 vorgestellten Arten von Komposition sollen an dieser Stelle durch Anwendungsbeispiele veranschaulicht werden. Dabei wird auf die sequentielle und parallele Komposition von Iteratees und Enumerators eingegangen.

Sequentielle Komposition von Iteratees

Zwei `Iteratees` sequentiell zu kombinieren, bedeutet zuerst den ersten `Iteratee` auszuführen, bis er keine Elemente mehr annimmt oder die Datenquelle erschöpft ist und dann den zweiten `Iteratee` nach dem gleichen Prinzip auszuführen. Dafür eignen sich die Methoden `map` und `flatMap` des `Iteratee`-Traits. Diese Methoden erlauben es, wie bei jeder Monade, den Wert innerhalb der Monade zu transformieren. Der Wert der `Iteratee`-Monade steht fest, sobald der `Iteratee` in den `Done`-Zustand übergegangen ist. Dies geschieht, nachdem der `Iteratee` die für ihn erforderlichen Elemente empfangen und daraus ein Ergebnis berechnet hat. Um zwei `Iteratees` nacheinander auszuführen, wird zuerst per `flatMap` auf den berechneten Wert des ersten `Iteratees` zugegriffen. Anschließend wird ein `map` über den zweiten `Iteratee` durchgeführt, um an das zweite Ergebnis zu gelangen und beide Ergebnisse zu kombinieren. Ein Code-Beispiel, das dieses Prinzip ausnutzt, ist in Listing 3.30 zu lesen.

```
1 val i1: Iteratee[Int, Option[Int]] = Iteratee.head
2 val i2: Iteratee[Int, Int] = Iteratee.fold(0)(_ + _)
3
4 val i12: Iteratee[Int, (Option[Int], Int)] =
5   i1.flatMap(res1 => i2.map(res2 => (res1, res2)))
6
7 val e: Enumerator[Int] = Enumerator(1, 4, -2)
8 val result: Future[(Option[Int], Int)] = e.run(i12)
9 // result hat den Wert Future((Some(1), 2))
```

Listing 3.30: Sequentielle Komposition von Iteratees

In diesem Fall gibt es zwei `Iteratees`. Der erste `Iteratee` konsumiert genau ein Element und gibt dieses, in einem `Option`-Wert verpackt, zurück. Der `Option`-Typ ist notwendig, damit auch ein Ergebnis existiert, wenn nur ein `Input.EOF`, aber kein `Input.El` empfangen wird. Der zweite `Iteratee` summiert alle empfangenen Elemente auf und gibt die Summe als Ergebnis zurück. Das Ergebnis der Komposition dieser beiden `Iteratees` ist ein Paar aus dem Ergebnis des ersten und des zweiten `Iteratees`. Das Kombinieren von `i1` und `i2` lässt sich durch das Verwenden einer `for`-Comprehension allerdings noch lesbarer gestalten, wie in Listing 3.31 zu sehen.

```
1 val i12: Iteratee[Int, (Option[Int], Int)] = for {
2   res1 <- i1
3   res2 <- i2
4 } yield (res1, res2)
```

Listing 3.31: Sequentielle Komposition von Iteratees mit for-Comprehension

Parallele Komposition von Iteratees: eine Quelle auf mehrere Senken

Bei der parallelen Komposition zweier `Iteratees` werden die Elemente der Datenquelle an beide `Iteratees` übergeben. Eine Datenquelle auf mehrere Datensenken abzubilden, lässt sich mit Hilfe von `Enumeratees` und den Methoden `zip`, bzw. `zipWith` bewerkstelligen. `zip` erstellt aus zwei `Iteratees` einen neuen `Iteratee`, der die empfangen Elemente an beide `Iteratees` weitergibt. Die genaue Signatur von `Enumeratee.zip` ist Listing 3.32 zu entnehmen.

```
def zip[E, A, B](inner1: Iteratee[E, A], inner2: Iteratee[E, B]):
  Iteratee[E, (A, B)]
```

Listing 3.32: Die Signatur von Enumeratee.zip

Sobald beide `Iteratees` im `Done`-Zustand sind, werden die Ergebnisse als Paar zurückgegeben. Falls ein `Iteratee` in den `Error`-Zustand übergeht, ist auch der resultierende `Iteratee` im `Error`-Zustand. Die Verwendung von `zipWith` ermöglicht es, das Ergebnis nicht nur als Paar zu erhalten, sondern es kann eine Funktion übergeben werden, die die Ergebnisse zu einem beliebigen Datentyp zusammenfasst. In Listing 3.33 ist ein Beispiel zu sehen, das die beiden `Iteratees` aus dem Beispiel für Sequentielle Komposition von Iteratees verwendet, sie aber parallel kombiniert.

```
val i1: Iteratee[Int, Option[Int]] = Iteratee.head
val i2: Iteratee[Int, Int] = Iteratee.fold(0)(_ + _)

val i12: Iteratee[Int, (Option[Int], Int)] =
  Enumeratee.zip(i1, i2)

val e: Enumerator[Int] = Enumerator(1, 4, -2)
val result: Future[(Option[Int], Int)] = e.run(i12)
// result hat den Wert Future((Some(1), 3))
```

Listing 3.33: Parallele Komposition von Iteratees mit einer Quelle und mehreren Senken

Parallele Komposition von Iteratees: mehrere Quellen auf eine Senke

Es können auch mehrere `Enumerators` auf einen `Iteratee` angewendet werden. Ein auf diese Weise zusammengesetzter `Iteratee` ist ein verschachtelter `Iteratee`, wobei der äußere `Iteratee` als Ergebnis einen inneren `Iteratee` zurückliefert, der von einem weiteren `Enumerator` Elemente benötigt. Wie ein solcher `Iteratee` aussieht, ist in Listing 3.34 zu sehen. `headHeadIteratee` ist ein `Iteratee`, der von zwei `Enumerators` liest und das jeweils erste Element zurückgibt. Um einen verschachtelten `Iteratee` auszuführen, müssen die

Iteratees der einzelnen Ebenen nacheinander ausgeführt werden, was an den Variablen intermediateResult und result zu erkennen ist.

```
def headHeadIteratee[A, B]:
  Iteratee[A, Iteratee[B, (Option[A], Option[B])]] =
    Iteratee.head.map(a => Iteratee.head.map(b => (a, b)))

val i:
  Iteratee[Char, Iteratee[Int, (Option[Char], Option[Int])]] =
    headHeadIteratee[Char, Int]
val e1: Enumerator[Char] = Enumerator('1', '2', '3')
val e2: Enumerator[Int] = Enumerator()

val intermediateResult:
  Iteratee[Int, (Option[Char], Option[Int])] =
    Iteratee.flatten(e1.run(i))

val result: Future[(Option[Char], Option[Int])] =
  e2.run(intermediateResult)
// result hat den Wert Future((Some('1'), None))
```

Listing 3.34: Parallele Komposition von Iteratees mit mehreren Quellen auf eine Senke

Sequentielle Komposition von Enumerators

Sequentielle Komposition zweier Enumerators heißt, dass erst die Elemente des ersten Enumerators generiert werden, bis dieser erschöpft ist und anschließend die Elemente des zweiten Enumerators generiert werden. Das Enumerator-Trait stellt hierfür die Methode andThen mit der Signatur Enumerator[E] => Enumerator[E] bereit. Diese Methode erstellt einen neuen Enumerator, der die Elemente des ursprünglichen Enumerators und dann die Elemente des übergebenen Enumerators enumeriert. Listing 3.35 zeigt, wie andThen verwendet werden kann, um zwei Enumerators sequentiell zu komponieren.

```
val i: Iteratee[Int, List[Int]] = Iteratee.getChunks
val e1 = Enumerator(1, 2)
val e2 = Enumerator(3)
val e12: Enumerator[Int] = e1.andThen(e2)

val result: Future[List[Int]] = e12.run(i)
// result hat den Wert Future(List(1, 2, 3))
```

Listing 3.35: Sequentielle Komposition von Enumerators

Parallele Komposition von Enumerators

Wenn immer das zuerst verfügbare Element zweier `Enumerators` generiert werden soll, wird dies als parallele Komposition bezeichnet. Das `Enumerator`-Companion-Objekt stellt hierfür die Konstruktormethode `interleave` bereit. `interleave` kommt in unterschiedlichen Varianten, darunter auch in einer Variante mit der in Listing 3.36 gezeigten Signatur.

```
def interleave[E](e1: Enumerator[E], es: Enumerator[E]*):
  Enumerator[E]
```

Listing 3.36: Die Signatur von interleave

`interleave` nimmt einen oder mehrere `Enumerators` als Argumente und kombiniert diese nach dem zuvor beschriebenen Prinzip. In Listing 3.37 wird mit Hilfe dieser Methode gezeigt, wie mehrere `Enumerators` parallel komponiert werden können.

```
def timeoutEnumerator[A](x: A, d: Duration): Enumerator[A] =
  Enumerator.flatten(Promise.timeout(Enumerator(x), d))

val i: Iteratee[Int, List[Int]] = Iteratee.getChunks
val e1 = timeoutEnumerator(1, 3 seconds)
val e2 = timeoutEnumerator(2, 1 second)
val e3 = timeoutEnumerator(3, 2 seconds)
val e123: Enumerator[Int] = Enumerator.interleave(e1, e2, e3)

val result: Future[List[Int]] = e123.run(i)
// result hat den Wert Future(List(2, 3, 1))
```

Listing 3.37: Parallele Komposition von Enumerators

Zu bemerken ist hierbei, dass Play das Objekt `play.api.libs.concurrent.Promise` mit mehreren Hilfsmethoden zur Verfügung stellt. Mit dessen `timeout`-Methode ist es möglich, ein `Future`-Objekt zu erstellen, dass nach Verlauf einer angegebenen Zeitspanne einen Wert liefert. `Enumerator.flatten` transformiert einen Wert vom Typ `Future[Enumerator[A]]` nach `Enumerator[A]` analog zur Methode `Iteratee.flatten`, die in Unterabschnitt 3.3.9 behandelt wurde. Gut zu erkennen ist, dass die Reihenfolge der Elemente im Ergebnis mit der jeweiligen Zeitspanne korreliert und nicht etwa mit der Reihenfolge, in der die `Enumerators` an `interleave` übergeben wurden.

3.3.12. Gesetze

Kiselyov identifiziert in Kiselyov 2012b (S. 12–13 u. S. 18–21) mehrere Gesetze für Iteratee-Streams, die Beweise über Code ermöglichen, der diese Streams verwendet. Zur besseren Lesbarkeit werden in Listing 3.38 einige Typ-Aliase eingeführt, die in den Gesetzesdefinitionen verwendet werden.

```
type E     = Char            // Der Eingabeelementtyp (Char)
type I[A]  = Iteratee[E, A] // Ein Iteratee von Char nach A
type M[A]  = Future[A]       // Die Monade (Future)
type IM[A] = M[I[A]]         // Ein Iteratee in der Future-Monade
```

Listing 3.38: Typ-Aliase der Gesetze

Außerdem wird die Funktion `en_str` mehrfach verwendet. Diese Funktion hat die Signatur `String => Enumerator[E]` und macht aus einem übergebenen String einen `Enumerator`, der als Elemente die einzelnen Zeichen des Strings generiert. Die in den Code-Ausschnitten gezeigten Variablen `left` und `right` haben immer die gleiche Bedeutung.

1. Komposition

   Die Komposition von `Enumerators` entspricht der Verkettung ihrer Eingaben. `s1` und `s2` in Listing 3.39 sind zwei beliebige `Strings`.

   ```
   val left: Enumerator[E] = en_str(s1 + s2)
   val right: Enumerator[E] = en_str(s1).andThen(en_str(s2))
   ```

   Listing 3.39: Kompositionsregel

2. Verkettung

   Ein `Iteratee i`, der einen `String s1` erkennt, erkennt auch `s1 + s2`, für jeden String `s2`. Erkennen bedeutet, dass ein `Iteratee` erfolgreich in den Endzustand `Done` übergeht.

   ```
   def flatMap[A, B](m: IM[A], f: A => IM[B]): IM[B] =
     m.map(_.flatMapM(f))

   val left: IM[String] = en_str(s1 + s2)(i.flatMap(f))
   val right: IM[String] =
     flatMap(en_str(s1)(i), (x: String) => en_str(s2)(f(x)))
   ```

   Listing 3.40: Verkettungsregel, wenn ein Iteratee die Eingabe erkennt

Die in Listing 3.40 verwendete Funktion `flatMap` verhält sich wie auch andere `flatMap`-Methoden. Diese Implementierung arbeitet allerdigns auf der Monade `IM`, die aus `Future` und `Iteratee` zusammengesetzt ist. Erkennt der `Iteratee` `s1` nicht, so gilt die in Listing 3.41 abgebildete Gleichheit.

```
val left: IM[String] = en_str(s)(i.flatMap(f))
val right: IM[String] = en_str(s)(i).map(_.flatMap(f))
```

Listing 3.41: Verkettungsregel, wenn ein Iteratee die Eingabe nicht erkennt

3. Nullelement

   Ein abweichender `Iteratee` ist ein absorbierendes Element bzgl. `flatMap`. Ein abweichender `Iteratee` ist ein `Iteratee`, der nach Empfang von `Input.EOF` nicht in einen Endzustand übergeht. `failure` in Listing 3.42 ist ein solcher `Iteratee`. Ein absorbierendes Element ist ein Element, das, mit jedem anderen Element verknüpft, wieder das absorbierende Element ergibt (vgl. Kilp u. a. 2000, S. 15). Die Null beispielsweise, ist das absorbierende Element der Multiplikation.

```
def failure[A]: I[A] = Cont(_ => failure)

val left: I[String] = failure.flatMap(f)
val right: I[String] = failure
```

Listing 3.42: Nullelementregel

4. Rechtsdistributivität

   `flatMap` ist rechtsdistributiv, sofern der `Iteratee` idempotent ist. Eine Funktion f ist idempotent, wenn $f(x) = f(f(x))$ (vgl. Aceto u. a. 2013, S. 1), mehr dazu im nächsten Absatz.

```
val left: I[String] =
  i.flatMap(x => alternative(k1(x), k2(x)))
val right: I[String] =
  alternative(i.flatMap(k1), i.flatMap(k2))
```

Listing 3.43: Rechtsdistributivitätsregel

`alternative` komponiert die Ergebnis-`Iteratees` der beiden Funktionen `k1` und `k2`. Der resultierende `Iteratee` erkennt, was der am schnellsten erkennende Eingabe-`Iteratee` erkennt. Idempotenz wird von Kiselyov, wie in Listing 3.44 zu sehen, definiert.

```
val left: M[(I[A], I[A])] =
  en_str(s)(i).flatMap(x => Future((x, x)))
val right: M[(I[A], I[A])] =
  en_str(s)(i).flatMap(x =>
    en_str(s)(i).flatMap(y => Future((x, y))))
```

Listing 3.44: Idempotenz eines Iteratees

4. Real-Time-Web

Nachdem im vorigen Kapitel Plays Streams vorgestellt wurden, wird in diesem Kapitel der letzte Teil vorgestellt, der für den Real-Time-Aspekt von Web-Anwendungen notwendig ist. In diesem Kapitel sollen die clientseitigen Werkzeuge vorgestellt werden, die Real-Time-Web-Anwendungen möglich machen. Dafür gibt es zwei Techniken. Diese sind Web-Sockets und Server Sent Events. Erstere bildeten das bekannte Socket-Konzept im Browser ab, wohingegen letztere eine Einweg-Kommunikation vom Server zum Client ermöglichen. Die Kommunikation in entgegengesetzter Richtung erfolgt dabei über herkömmliche HTTP-Anfragen.

4.1. Web-Sockets

Das Web-Socket-Protokoll (vgl. IETF 2013) ermöglicht es Client und Server sich gegenseitig über ein Web-Socket Nachrichten zu schicken. Auf Client-Seite gibt es hierfür den `WebSocket`-Typ. Auf Server-Seite (Play) gibt es ebenfalls eine Klasse namens `WebSocket`, die im Controller anstelle von `Action` als Request-Handler verwendet werden kann. Web-Sockets werden von Chrome 14, Firefox 11 und Internet Explorer 10 und in neueren Versionen unterstützt (vgl. Deveria 2013b).

4.1.1. Client-Seite

W3C 2012b definiert das Interface für Web-Sockets in JavaScript. Die für diese Arbeit relevanten Teile des Interfaces sind die folgenden: In Listing 4.1 ist eine vereinfachte Version des Interfaces abgebildet, die nur die für diese Arbeit relevanten Teile zeigt. Ein `EventHandler` kann jede beliebige Funktion sein, beim Aufruf wird ein `Event`-Objekt übergeben, das Informationen über das jeweilige Ereignis enthält (vgl. W3C 2013).

```
1 [Constructor(DOMString url)]
2 interface WebSocket {
3   attribute EventHandler onopen;
4   attribute EventHandler onclose;
5   attribute EventHandler onmessage;
```

```
6   void close();
7   void send(DOMString data);
8 };
```

Listing 4.1: Das WebSocket-Interface in JavaScript

Mittels `new WebSocket(url)` lässt sich ein WebSocket auf die angegebene URL öffnen. Auf `onopen`, `onerror`, `onclose` und `onmessage` können `EventHandler` registriert werden, die bei Auftreten des jeweils gleichnamigen Events aufgerufen werden. Das `Event`, das dem `onmessage`-Handler übergeben wird, enthält ein Attribut `data`, das den Inhalt der empfangenen Nachricht als `String` enthält. Mit `send` können Daten an den Server übertragen werden. Die `close`-Methode schließt das `WebSocket` wieder.

4.1.2. Server-Seite

Auf der Server-Seite wird ein `WebSocket` ähnlich wie eine `Action` dargestellt. Statt eines `Action { ... }`-Blocks wird im Controller ein `WebSocket.using { ... }`-Block verwendet. `WebSocket.using[A]` erwartet als Argument eine Funktion von `RequestHeader` nach `(Iteratee[A, _], Enumerator[A])`. `A` ist der Typ der Nachrichten, die mit dem Client ausgetauscht werden, dieser ist der gleiche für eingehende und ausgehende Nachrichten. Der `Request-Handler` muss in der Regel nicht direkt verwendet werden. Die Funktion muss ein Paar aus `Iteratee` und `Enumerator` zurückgeben. Der `Iteratee` empfängt die eingehenden Nachrichten des Clients und der `Enumerator` generiert Nachrichten, die an den Client verschickt werden.

Zusätzlich zur oben beschriebenen Methode im Controller, muss eine entsprechende Route in der `conf/routes`-Datei angelegt werden. Dieser Eintrag unterscheidet sich nicht von denen für reguläre `Actions` und wird für die HTTP-Methode `GET` definiert.

4.1.3. Altersstatistiken-Anwendung

Um die statische Anwendung zur Erfassung von Altersstatistiken soll dynamisch werden und neue Einträge in Echtzeit anzuzeigen. Dazu sollen die Altersangaben nach der Eingabe an alle aktiven Clients gesendet werden, damit diese daraufhin ihre Darstellung aktualisieren. Es hierfür View, Controller und Routen-Datei geändert werden. In der `conf/routes`-Datei wird der Eintrag, der auf die `input`-Action zeigt auf die in Listing 4.2 gezeigte Definition geändert.

```
1 GET     /input     controllers.Application.input
```

Listing 4.2: Web-Sockets in der routes-Datei der Altersstatistiken-Anwendung

Die `input-Action` wird durch einen im vorigen Unterabschnitt eingeführten `WebSocket.using`-Block ersetzt, der in Listing 4.3 zu sehen ist. Als Nachrichtentyp ist hierbei `String` gewählt, weil sowohl Client, als auch Server gut damit arbeiten können. Denkbar wären auch `Int` oder `JsValue` für JSON-Nachrichten, diese würden das Beispiel allerdings verkomplizieren. Um möglichst einfach alle aktiven Clients erreichen zu können, wird mit Hilfe der Konstruktormethode `Concurrent.broadcast`, die in Unterabschnitt 3.3.9 (Anwendung von Enumerators) vorgestellt wurde, ein `Enumerator` mit assoziiertem `Channel` erstellt. Über diesen `Channel` können imperativ Elemente an den `Enumerator` übergeben werden. Sobald eine neue Altersangabe über den `Iteratee` bekannt wird, wird diese über den `Enumerator` an alle Clients verbreitet, damit sie ihre Datendarstellung aktualisieren können.

```
val (outEnumerator, outChannel) = Concurrent.broadcast[String]

def input = WebSocket.using[String] { request =>
  val in = Iteratee.foreach[String] { ageString =>
    type NFE = NumberFormatException
    catching(classOf[NFE]).opt(ageString.toInt).foreach { age =>
      if (age > 0 && age < 100) {
        ageStatistics =
          ageStatistics.updated(age, ageStatistics(age) + 1)
        outChannel.push(age.toString)
      }
    }
  }

  (in, outEnumerator)
}
```

Listing 4.3: Web-Sockets im Controller der Altersstatistiken-Anwendung

In der View muss weiterer JS-Code eingefügt werden, das zuvor erstellte Formular und auch der Rest der View kann wiederverwendet werden Listing 4.4 zeigt, den neu hinzugekommenen Code. Der darin zu sehende Code `@routes.Application.input.webSocketURL()` führt neben dem in Unterabschnitt 2.9.1 eingeführten Reverse Routing auf der generierten URL einen Aufruf von `webSocketURL` durch. Der Aufruf von `webSocketURL` sorgt dafür, dass die Route das Web-Socket-Protokoll verwendet (vgl. Hilton u. a. 2013, S. 281). Obige Anweisung wird bei einer lokalen Installation zu `ws://localhost:9000/input`.

```
var ws =
  new WebSocket("@routes.Application.input.webSocketURL()");
```

```
3
4  ws.onopen = function() {
5    form.onsubmit = function() {
6      ws.send(input.value)
7      form.reset();
8      return false; // prevent submission
9    }
10 };
11
12 ws.onmessage = function(event) {
13   var age = parseInt(event.data, 10);
14   chart.increment(age);
15   chart.update();
16 };
```

Listing 4.4: Web-Sockets in der View der Altersstatistiken-Anwendung

Bevor das `WebSocket` geöffnet und nachdem es geschlossen wurde, muss verhindert werden, dass das Formular abgeschickt werden kann, weil serverseitig keine `Action` für das Formular existiert. Der Code dafür soll an dieser Stelle aber nicht gezeigt werden werden. Die verwendeten Variablen `form` und `input` enthalten Referenzen auf die HTML-Elemente für das Formular und das Eingabefeld. Sobald das `WebSocket` geöffnet ist, wird dafür gesorgt, dass bei Formularabsendung das `WebSocket` verwendet wird, statt einer regulären Formularübertragung. Sobald vom Server eine Nachricht eintrifft, wird das Diagramm beim Client über die bereits vorhandene `chart`-Variable aktualisiert. Diese Variable wurde zuvor mittels der `makeAgeStatisticsChart`-Funktion erstellt, die sich in der Datei `public/javascripts/main.js` befindet.

4.2. Server Sent Events

Server Sent Events ist eine Technologie, bei der die Server-Seite Nachrichten an die Client-Seite senden kann. Dabei wird im Gegensatz zu Web-Sockets kein eigenständiges Protokoll verwendet, sondern auf HTTP zurückgegriffen. Server Sent Events wird von Chrome 6 und Firefox 6 und in neueren Versionen, nicht aber von Internet Explorer unterstützt (vgl. Deveria 2013a).

4.2.1. Client-Seite

Auf der Client-Seite gibt es das `EventSource`-Interface, das, wie der Name schon sagt, die serverseitige Datenquelle repräsentiert. Eine vereinfachte Darstellung des Interfaces, die nur die für diese Arbeit relevanten Teile enthält, ist in Listing 4.5 zu sehen (vgl. W3C 2012a).

```
[Constructor(DOMString url)]
interface EventSource {
  attribute EventHandler onopen;
  attribute EventHandler onmessage;
  void close();
};
```

Listing 4.5: Das EventSource-Interface in JavaScript

Das `EventSource`-Interface hat große Ähnlichkeit mit dem `WebSocket`-Interface. Was im Vergleich dazu allerdings fehlt, ist die `send`-Methode, weil Server Sent Events nur Kommunikation in eine Richtung erlauben. Abgesehen davon erfüllen die Attribute und Methoden die gleichen Aufgaben, wie ihre Pendants im `WebSocket`-Interface. Die empfangenen Nachrichten sind `Strings` (vgl. W3C 2012a).

4.2.2. Server-Seite

Play unterstützt Server Sent Events mit Hilfe von Actions, die statt einer einfachen Antwort einen Datenstrom zurückliefern. Das HTTP-Protokoll unterstützt mittels Chunked Transfer Encoding das kontinuierliche Senden von Daten vom Server an den Client, anstelle von Antworten fester Größe. Um eine HTTP-Antwort dieser Form zu erstellen kann auf einem Wert des Typs `play.api.mvc.Results.Status` die `chunked`-Methode aufgerufen werden. Diese Methode lässt aus einem übergebenen `Enumerator` die Inhalte der Antwort generieren Zenexity und Typesafe Inc. (2013o).

Ein solcher Stream ist allerdings noch keine gültige Datenquelle für Server Sent Events. Um das zu ändern muss der übergebene `Enumerator` durch den `play.api.libs.EventSource-Enumeratee` transformiert werden, wodurch die Stream-Elemente korrekt kodiert werden. Anschließend muss der Dokumenttyp auf `text/event-stream` gesetzt werden, damit der Stream clientseitig korrekt erkannt wird (vgl. W3C 2012a). Eine gültige `EventSource` ist in Listing 4.6 zu sehen.

```
def eventSource = Action {
  Ok.chunked(Enumerator("44", "34", "50").through(EventSource()))
    .as("text/event-stream")
```

```
}
```

Listing 4.6: Server Sent Events auf Server-Seite

4.2.3. Altersstatistiken-Anwendung

Die Altersangaben werden wie bei der statischen Variante über separate HTTP-Nachrichten empfangen, wofür nach wie vor im Controller die `input`-Action zuständig ist. Um diese Antworten an alle verbundenen Clients weiterzuleiten wird, wie auch schon bei den Web Sockets mit einem `Enumerator` und `Concurrent.broadcast` gearbeitet. Um nach Empfang einer Altersangabe diese Information an den `Enumerator` zu übergeben, wird die `input`-Action, wie in Listing 4.7 zu sehen, um eine Zeile erweitert.

```
val (outEnumerator, outChannel) = Concurrent.broadcast[Int]

def input = Action { implicit request =>
  ageForm.bindFromRequest.fold(
    invalidForm => BadRequest(invalidForm.errorsAsJson.toString),
    { age =>
      ageStatistics =
        ageStatistics.updated(age, ageStatistics(age) + 1)
      outChannel.push(age)
      Ok
    }
  )
}
```

Listing 4.7: Server Sent Events in der input-Action der Altersstatistiken-Anwendung

Um die Datenquelle vom Server abrufen zu können, muss zunächst ein Eintrag in der `routes`-Datei hinzugefügt werden. Die Datenquelle soll über die URL `/dataSource` erreichbar sein, woraus sich der Eintrag aus Listing 4.8 ergibt.

```
GET     /eventSource     controllers.Application.eventSource
```

Listing 4.8: Server Sent Events in der routes-Datei der Altersstatistiken-Anwendung

Der im vorigen Unterabschnitt gezeigt `EventSource-Enumeratee` nutzt Teile, die auf eine andere Technologie namens Comet Sockets ausgelegt sind. Im Falle von Comet Sockets wird anstelle von einfachen Nachrichten, wie bei Server Sent Events, kompletter JavaScript-Code an den Client gesendet (vgl. Zenexity und Typesafe Inc. 2013c). Dies, so vermutet der Author, wird der Grund sein, weshalb o. g. `Enumeratee` übergebene `Strings` in Anführungszeichen

setzt (’...’), sobald eine String-Nachricht an den Client gesendet wird. Wenn vom Server aus die String-Nachricht mit dem Inhalt 35 an den Client gesendet wird, so kommt beim Client eine String-Nachricht mit dem Inhalt ’35’ an, was in diesem Fall nicht gewollt ist. Dieses beobachtete Verhalten ist ein Bug und wird in der folgenden Play-Version 2.3 korrigiert (vgl. Zenexity und Typesafe Inc. 2013e).

Der EventSource-Enumeratee nimmt allerdings u. a. einen impliziten Parameter vom Typ CometMessage[A]. Diese case class nimmt als Konstruktorargument eine Funktion vom Typ A => String. Mit einem passenden Wert lässt sich oben beschriebenes Phänomen also umgehen. Weil es in diesem Fall also nicht zu vermeiden ist, einen impliziten Parameter mitzugeben, kann auch gleich mit Ints gearbeitet werden, indem ein impliziter Wert vom Typ CometMessage[Int] bereitgestellt wird. Die in Listing 4.9 gezeigte Lösung nutzt daher den impliziten Wert intMessage, um die Zahlen des Enumerators in Strings umzuformen.

```
implicit val intMessage = Comet.CometMessage[Int](_.toString)
def eventSource = Action {
  Ok.chunked(outEnumerator.through(EventSource()))
    .as("text/event-stream")
}
```

Listing 4.9: Server Sent Events im Controller der Altersstatistiken-Anwendung

Auf der Client-Seite wird zunächst dafür gesorgt, dass beim Absenden des Formulars die Seite nicht neu geladen wird, sondern im Hintergrund ein HTTP-Request mit dem Formulardaten versendet wird. Dann wird der entsprechende EventHandler für onmessage gesetzt, um nach Empfang neuer Informationen das Diagramm zu aktualisieren. Dies ist in Listing 4.10 zu sehen.

```
var form = document.forms[0];
var input = document.getElementById("ageInput");

form.onsubmit = function() {
  var params = "age=" + input.value;
  var request = new XMLHttpRequest();
  request.open("POST", "@routes.Application.input");
  request.setRequestHeader("Content-type",
                           "application/x-www-form-urlencoded");
  try {
    request.send(params);
  } catch (e) {} // invalid input will result in Bad Request

```

```
14    form.reset();
15    return false; // prevent submission
16  }
17
18  var eventSource =
19    new EventSource("@routes.Application.eventSource");
20
21  eventSource.onmessage = function(event) {
22    var age = parseInt(event.data, 10);
23    chart.increment(age);
24    chart.update();
25  };
```

Listing 4.10: Server Sent Events in der View der Altersstatistiken-Anwendung

4.3. Web Sockets vs. Server Sent Events

Wann ist es angebracht Web Sockets zu verwenden und wann sollten besser Server Sent Events eingesetzt werden? Web Sockets sind relativ mächtig, indem sie bidirektionale Kommunikation über ein eigenes Protokoll ermöglichen, wohingegen Server Sent Events lediglich eine Kommunikationsrichtung unterstützen, dafür aber auf HTTP zurückgreifen. Nehlsen 2013b hat in einem Test herausgefunden, dass Server Sent Events in einer Anwendung ähnlich der hier entwickelten Altersstatistikenanwendung performanter als Web Sockets sind. Wird also nur Kommunikation in eine Richtung benötigt, sollten Server Sent Events eingesetzt werden.

Bei Anwendungen, bei denen viel Kommunikation zwischen Client und Server anfällt, wie es z. B. bei Spielen der Fall sein kann, bieten sich stattdessen Web Sockets an, weil diese Kommunikation über die gleiche Verbindung durchgeführt werden kann. Bei Anwendungen, die hauptsächlich Nachrichten vom Server zum Client versenden, aber nur wenige Nachrichten in die andere Richtung, können auch Server Sent Events verwendet werden. In diesem Fall würden die wenigen Nachrichten, die zum Server gesendet werden, mittels HTTP im Hintergrund versendet, wie es in Unterabschnitt 4.2.3 demonstriert wurde (vgl. Bidelman 2010).

5. Anwendung: Twitter News

In diesem Kapitel wird eine Real-Time-Web-Anwendung entwickelt, welche die in den vorigen Kapiteln vorgestellten Techniken anwendet. Twitter News analysiert Tweets von Nachrichtenseiten und extrahiert dabei die meistgetweeteten Wörter, die meistgeretweeteten und die meistdiskutierten Tweets. Hierbei wird in erster Linie demonstriert, wie Plays reaktive Komponenten eingesetzt werden, um eine Real-Time-Web-Anwendung zu realisieren.

5.1. Idee

Twitter News soll in anschaulicher Art und Weise die meistgetweeteten Wörter, die meistretweeteten und die meistdiskutierten Tweets von Nachrichtenseiten darstellen. Die dargestellten Daten sollen immer aktuell sein, weshalb sie auf einen bestimmten Zeitraum beschränkt werden, wie z. B. die letzten fünf Minuten. Zusätzlich sollen die im Browser dargestellten Daten aktualisiert werden, sobald der Server neue Daten empfängt, sodass zu jeder Zeit nur relevante Nachrichten angezeigt werden. Abb. 5.1 zeigt einen Screenshot der fertigen Anwendung, um die beschriebene Idee zu verdeutlichen. In den folgenden Abschnitten wird aber in erster Linie auf die Anwendungslogik eingegangen.

Es existiert bereits eine ähnliche Anwendung namens BirdWatch (vgl. Nehlsen 2013a), die allerdings umfangreicher ist und einen etwas anderen Ansatz verfolgt. BirdWatch filtert Tweets nach einem Text, der vom Nutzer eingegeben wurde, wohingegen Twitter News eine eigene Liste von Twitter-Accounts besitzt, deren Tweets dargestellt werden. Des Weiteren besitzt BirdWatch ein umfangreicheres User-Interface als Twitter News, weil Twitter News in erster Linie eine Demonstration der in dieser Arbeit vorgestellten Techniken ist und keine vollwertige Web-Anwendung sein soll. Der Autor hat keinen Code von BirdWatch verwendet, sondern die Twitter News-Anwendung eigenständig entwickelt.

5.2. Werkzeuge

In diesem Abschnitt sollen die Werkzeuge identifiziert werden, mit denen die zuvor definierte Idee umgesetzt werden kann. Um an die Daten von Twitter zu gelangen, stellt Twitter sog. Stre-

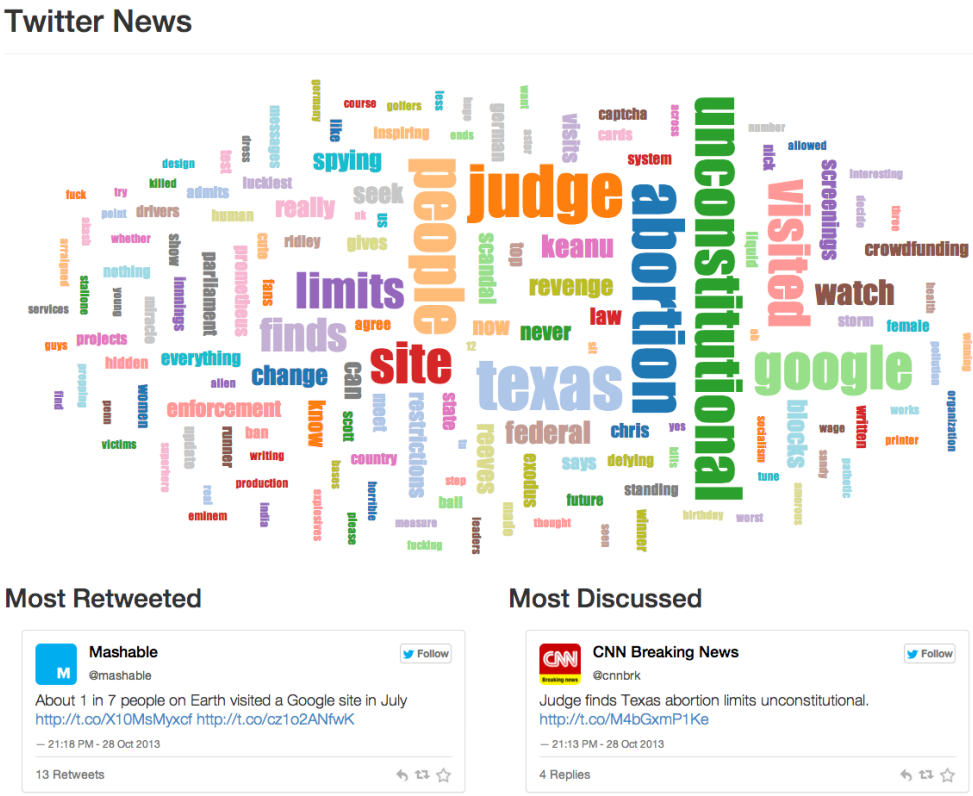

Abbildung 5.1.: Die TwitterNews-Anwendung

aming APIs zur Verfügung (vgl. Twitter 2012). Mit Hilfe der Streaming APIs kann der globale Twitter-Verkehr gefiltert werden, sodass z. B. nur Tweets von ausgewählten Nachrichtenseiten empfangen werden. Diese Streams werden über eine HTTP-Verbindung als JSON-Nachrichten übertragen.

Dieser eingehende Nachrichten-Stream kann auf Server-Seite mit Hilfe von `Iteratees` verarbeitet werden. Das Play-Framework enthält Werkzeuge, um HTTP-Anfragen zu versenden und die Antwort mit `Iteratees` nach und nach zu verarbeiten.

Nach Verarbeitung der eingehenden Tweet-Daten müssen die Nachrichten schließlich an die verbundenen Clients gesendet werden. Da hierbei nur Daten vom Server an die Clients versendet werden sollen, bieten sich Server Sent Events an. Diese Daten werden auf Client-Seite dann möglichst ansprechend dargestellt.

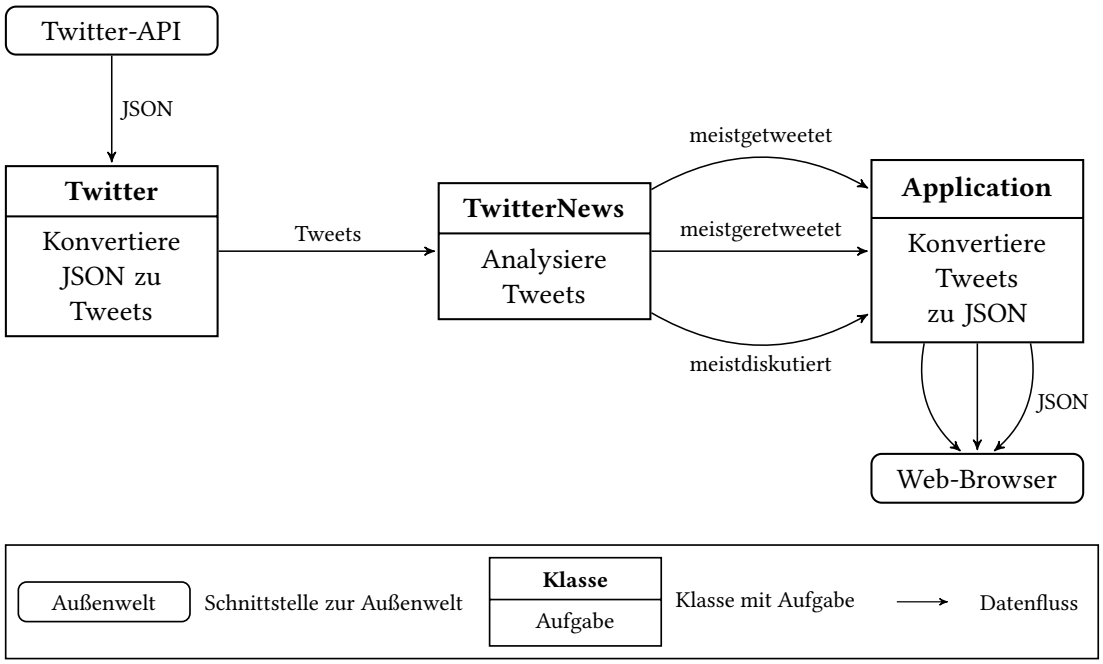

Abbildung 5.2.: Der Datenfluss der Twitter News-Anwendung

5.3. Umsetzung

In diesem Abschnitt wird beschrieben, wie die zuvor genannten Werkzeuge genutzt werden, um die Anwendung zu implementieren. Dabei wird nicht Schritt für Schritt durch den Entwicklungsprozess geführt, sondern die bereits geschriebene Anwendung vorgestellt. Es werden die Teile gezeigt, die von den im Hauptteil der Arbeit behandelten Komponenten Gebrauch machen. Es werden konkret verwendete Teile der Iteratee-Streams-Bibliothek gezeigt und es wird auf besondere Implementierungsentscheidungen eingegangen.

5.3.1. Architektur

Die Anwendungslogik besteht im Kern aus den zwei Klassen `Twitter` und `TwitterNews`. `Twitter` ist dafür zuständig, mit der Twitter-API zu kommunizieren und die empfangenen Daten als Tweet-Stream bereitzustellen. `TwitterNews` liest diesen Tweet-Stream und erstellt daraus Statistiken über Tweets für einen bestimmten Zeitraum. Diese Statistiken bestehen aus den meistgetweeteten Worten, den meistgeretweeteten Tweets und den meistdiskutierten Tweets. Für jede dieser drei Teilstatistiken wird ein Stream bereitgestellt, der immer aktualisiert wird, sobald ein neuer Tweet analysiert wird. Diese drei Streams werden schließlich vom

`Application`-Controller im JSON-Format an die verbundenen Clients gesendet, wo die Daten im Web-Browser angezeigt werden. Abb. 5.2 stellt den beschriebenen Datenfluss grafisch dar.

5.3.2. Das Twitter-Model

Die `Twitter`-Klasse ist für die Kommunikation mit der Twitter-API zuständig. Sobald der Tweet-Stream angefordert wird, wird eine Verbindung zur Twitter-API aufgebaut. Die daraufhin empfangenen Tweets werden in eine interne `Tweet`-Repräsentation konvertiert und schließlich als Tweet-Stream ausgeliefert. Die Besonderheit hierbei ist, dass die Twitter-API nach einiger Zeit keine Tweets mehr versendet und die Verbindung erneuert werden muss, bevor neue Tweets empfangen werden können. Mögliche Gründe hierfür sind beispielsweise das Öffnen einer weiteren Verbindung mit dem gleichen API-Schlüssel (es ist nur eine einzige Verbindung erlaubt) oder das Neustarten eines Streaming-Servers auf Seiten Twitters (vgl. Twitter 2013). Der Autor beobachtet allerdings, dass ca. alle 90 Sekunden keine Tweets mehr empfangen werden, was seinem Verständnis nach nicht mit den oben beschriebenen Gründen vereinbar ist.

Um zu erkennen, wann die Verbindung erneuert werden muss, wird bei Empfang neuer Daten ein Timer gesetzt, der nach Ablauf einer bestimmten Zeitspanne eine neue Verbindung aufbaut. Sobald aber weitere Daten empfangen werden, wird der zuvor erstellte Timer gestoppt und erneut gestellt. Falls keine weiteren Daten empfangen werden und eine neue Verbindung aufgebaut wird, werden alle über die neue Verbindung empfangenen Tweets wieder an den ursprünglichen Tweet-Stream geleitet. Um letzteres zu ermöglichen, wird der Tweet-`Enumerator` zusammen mit einem `Channel` mittels der Methode `play.api.libs.iteratee.Concurrent.broadcast` erstellt, welche in Abschnitt 3.3.9 (unter Erstellung durch Konstruktormethode im Concurrent-Objekt) behandelt wurde. Über diesen `Channel` können auch aus neuen Verbindungen heraus Daten an den ursprünglichen `Enumerator` übergeben werden.

5.3.3. Das TwitterNews-Model

`TwitterNews` konsumiert den von `Twitter` bereitgestellten Tweet-Stream und stellt auf Basis dieses Streams selber drei eigene Streams bereit, wie in Listing 5.1 zu sehen.

```
1 class TwitterNews(twitter: Twitter, relevantDuration: Duration) {
2   def mostTweetedEnumerator:   Enumerator[Map[String, Int]] = ???
3   def mostRetweetedEnumerator: Enumerator[Map[Tweet,  Int]] = ???
4   def mostDiscussedEnumerator: Enumerator[Map[Tweet,  Int]] = ???
```

```
}
```

Listing 5.1: Die TwitterNews-Klasse

Verarbeitung eingehender Tweets

Um den `Enumerator[Tweet]` zu verarbeiten, wird dieser intern auf einen `Iteratee` angewendet. Dieser interne `Iteratee` aktualisiert die Menge der für den beobachteten Zeitraum relevanten Tweets und die damit zusammenhängenden Statistiken. Diese aktualisierten Daten werden dann an die drei eigenen `Enumerators` übergeben. Die beschriebene Form der Datenverarbeitung ist in vereinfachter Form in Listing 5.2 zu sehen, in dem gezeigt wird, wie der Stream für die meistgetweeteten Wörter aktualisiert wird.

```
var relevantTweets = ???
val (mostTweetedEnumerator, mostTweetedChannel) =
  Concurrent.broadcast[Map[String, Int]]

twitter.statusStream(Iteratee.foreach[Tweet] { tweet =>
  relevantTweets =
   updatedRelevantTweets(relevantTweets, tweet, relevantDuration)
  val mostTweeted = updatedMostTweeted(relevantTweets)
  mostTweetedChannel.push(mostTweeted)
})
```

Listing 5.2: Die Verarbeitung eingehender Tweets in der TwitterNews-Klasse

Sonderfall: meistdiskutierte Tweets

Bei der Elementgenerierung für den Stream der meistdiskutierten Tweets besteht folgende Herausforderung: Tweets, die eine Antwort auf einen anderen Tweet sind, beinhalten nicht den gesamten Tweet, auf den sie antworten, sondern nur die ID des anderen Tweets. Im Gegensatz zu Retweets, über die der originale Tweet ausgelesen werden kann, muss für jeden diskutierten Tweet eine Anfrage an die Twitter-API gesendet werden, um den ursprünglichen Tweet zu empfangen. Um nicht für jede einzelne Tweetantwort eine HTTP-Anfrage versenden zu müssen, erfolgen einige aufwandsersparende Maßnahmen.

Intern wird ein Stream der meistdiskutierten Tweet-IDs vom Typ `Enumerator[Map[Long, Int]]` erstellt, wobei die IDs durch `Longs` und die Anzahl der Antworten durch `Ints` repräsentiert werden. Um für diesen `Enumerator` den `Enumerator[Map[Tweet, Int]]` mit den eigentlichen Tweets zu erstellen, werden die Tweet-IDs jedes generierten Elements des

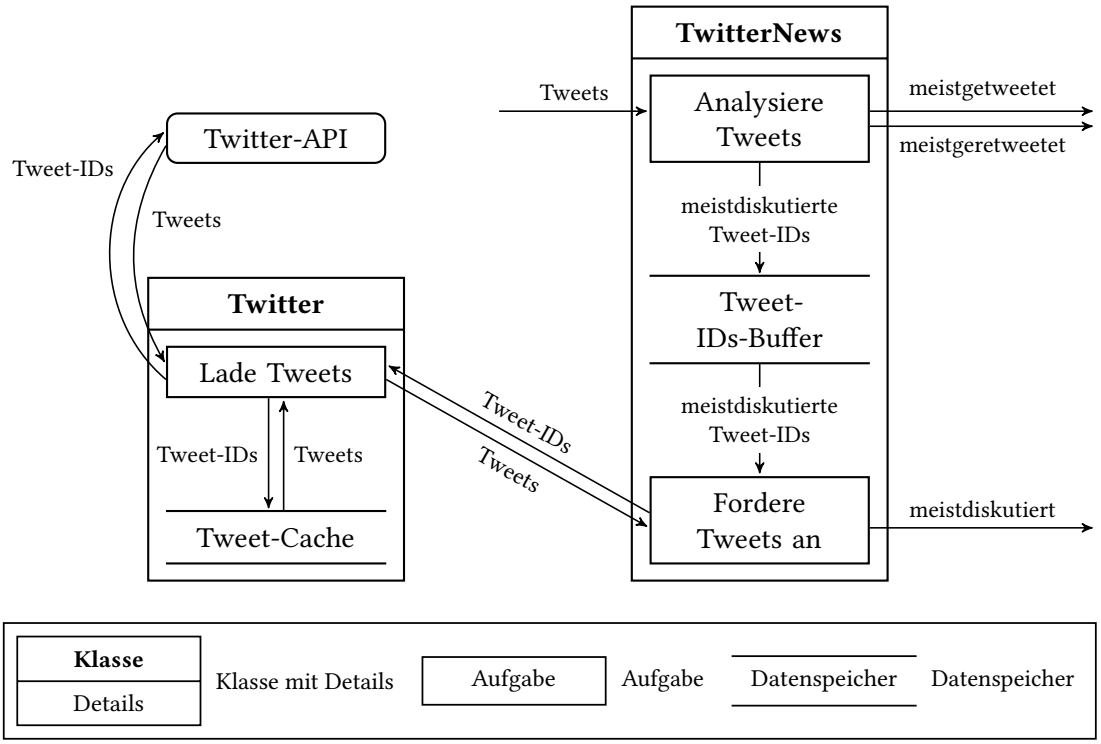

Abbildung 5.3.: Der Datenfluss zur Auflösung meistdiskutierter Tweet-IDs

`mostDiscussedIdsEnumerator` über die Twitter-API zu Tweets aufgelöst. Die Kommunikation mit der Twitter-API wird wieder von der `Twitter`-Klasse erledigt, die die Tweets zu aufgelösten Tweet-IDs cached, damit für einen Tweet nicht mehrere Anfragen gesendet werden müssen. Diese aufgelösten Tweets werden dann zusammen mit dem jeweiligen Antwortzähler als Element des `mostDiscussedEnumerators` generiert.

Weil das Auflösen der IDs einige Zeit in Anspruch nimmt, ist es wahrscheinlich, dass in der Zwischenzeit mehrere Element des `mostDiscussedIdsEnumerator` generiert worden sind. Deshalb werden die Elemente des Tweet-IDs-Streams vor der Verarbeitung in einen Buffer geschrieben. Dieser Buffer fasst immer nur ein Element, und zwar immer die neueste `Map` der meistdiskutierten Tweet-IDs. Der `mostDiscussedEnumerator` übersetzt also nicht alle Elemente des Tweet-IDs-Streams, aber immer die neuesten. Das Übersetzen aller Elemente ist aufgrund des Zeitaufwands für die HTTP-Anfragen an die Twitter-API schlicht unmöglich. Der Durchsatz wird durch das Cachen der Antworten allerdings wesentlich erhöht. Falls nach dem Versenden einer Anfrage an die Twitter-API nach dem Verlauf einer bestimmten Zeitspanne keine Antwort oder eine Antwort, die kein Tweet ist, empfangen wird, so wird kein Element

für den `mostDiscussedEnumerator` generiert. Abb. 5.3 zeigt das beschriebene Verfahren als Diagramm.

In Listing 5.3 ist zu sehen, wie der `mostDiscussedEnumerator` mit Hilfe von `Enumeratees` implementiert worden ist. Wie zuvor beschrieben, werden die Elemente des `mostDiscussedIdsEnumerators` in einen Buffer geschrieben. Play liefert bereits einen passenden `Buffer-Enumeratee`, der im `play.api.libs.iteratee.Concurrent`-Objekt zu finden ist. Anschließend werden die Elemente aus dem Buffer durch einen weiteren `Enumeratee` transformiert, der das jeweilige Element vom Typ `Map[Long, Int]` nimmt und die Tweet-IDs mit Hilfe der `Twitter`-Klasse auflösen lässt. `twitter.fetchTweets` gibt ein `Future[Seq[Tweet]]` zurück, dessen zukünftiges Ergebnis mit Hilfe der `map`-Methode dann in `Map[Tweet, Int]` transformiert wird. Falls die Tweet-IDs nicht aufgelöst werden konnten, ist das `Future` fehlgeschlagen und kann mit einer Prüfung auf die Ursache, in diesem Fall ein Wert vom Typ `TimeoutException` oder `InvalidTweetFormatException`, in einen erfolgreichen Zustand gebracht werden. Der verwendete `Enumeratee` `mapM` ist extra für asynchrone Berechnungen wie diese ausgelegt, weshalb der monadische `Future`-Wert als Ergebnis ausreicht, ohne dass in der Berechnung selber auf ein Ergebnis gewartet werden muss. Im letzten Schritt werden die positiven Ergebnisse herausgefiltert und als Elemente des `mostDiscussedEnumerators` generiert.

```
1  val mostDiscussedEnumerator: Enumerator[Map[Tweet, Int]] =
2    mostDiscussedIdsEnumerator.through(
3      Concurrent.buffer(1).compose(
4        Enumeratee.mapM[Map[Long, Int]] { map =>
5          val (ids, replyCounts) = map.unzip
6          val futureTweets = twitter.fetchTweets(ids.toSeq)
7          val futureMostDiscussed =
8            futureTweets.map(_.zip(replyCounts).toMap)
9          futureMostDiscussed.map(Some(_)).recover {
10           case _: TimeoutException |
11                _: InvalidTweetFormatException => None
12         }
13       }
14     ).compose(Enumeratee.collect { case Some(tweets) => tweets })
15   )
```

Listing 5.3: Der mostDiscussedEnumerator

5.3.4. Der Application-Controller

Der `Application`-Controller ist der Teil der Anwendung, der direkt mit den Nutzern zu tun hat. Dieser Controller sorgt dafür, dass Anfragen für die Hauptseite und die `EventSources` für die Server Sent Events beantwortet werden. Für die Hauptseite wird ein passendes View-Template gerendert. Für die einzelnen `EventSources` werden die jeweiligen Streams des `TwitterNews`-Models verwendet. Diese Streams werden allerdings so verlangsamt, dass z. B. nur alle fünf Sekunden ein Element generiert wird. Elemente, die in der Zwischenzeit generiert werden, werden ignoriert, sodass z. B. nur alle fünf Sekunden das aktuellste Element an die Clients versendet wird. Würden die Elemente sofort an die Clients gesendet, so könnte die ständige Neudarstellung der Daten so viel Rechenaufwand bei den Clients erzeugen, dass der gesamte PC des Nutzers/der Nutzerin ausgebremst würde.

5.3.5. Die Client-Seite

Auf der Client-Seite werden mittels Server Sent Events die Streams der meistgetweeteten Wörter, der meistgeretweeteten Tweets und der meistdiskutierten Tweets empfangen. Aufgabe der Client-Seite ist hierbei nur die Veranschaulichung der Daten. Dazu werden die meistgetweeteten Wörter in einer Word-Cloud angezeigt, die sich im Sekundentakt aktualisiert und so die meistgetweeteten Wörter der letzten zwei Minuten darstellt. Die meistgeretweeteten und meistdiskutierten Tweets werden in einer jeweils eigenen Spalte dargestellt und zeigen neben den jeweiligen Tweets auch die Anzahl der Retweets, bzw. Antworten, nach denen sie auch sortiert sind. Diese beiden Spalten werden nur alle fünf Sekunden aktualisiert, weil die angezeigten Daten weniger Veränderungen unterliegen, als die meistgetweeteten Wörter. Abb. 5.1 auf S. 58 zeigt einen Screenshot der fertigen TwitterNews-Anwendung.

6. Fazit und Ausblick

In diesem abschließenden Kapitel wird ein Fazit aus den in dieser Arbeit vorgestellten Inhalten gezogen. Anschließend wird ein Ausblick gegeben, was in dieser Arbeit nicht behandelt werden konnte, aber was aufbauend auf dieser Arbeit oder in ihrem thematischen Umfeld für Forschungsmöglichkeiten bestehen.

6.1. Fazit

Ziel dieser Arbeit war es, herauszufinden, wie mit Play Web-Anwendungen für das Real-Time-Web entwickelt werden können. Dazu wurde in den Grundlagen erklärt, wie eine einfache statische Web-Anwendung entwickelt wird. Diese Anwendung wurde mit den Erkenntnissen der Arbeit später um den Real-Time-Aspekt erweitert. Der statische Teil ließ sich ohne Schwierigkeiten umsetzen, wohingegen der dynamische Teil komplizierter war. Der komplizierteste Teil war das Verständnis der Iteratee-Streams und das Wissen um die denen zugrundeliegenden Konzepte, wie Monaden und speziell die `Future`-Monade. Erschwerend hinzu kam die Entdeckung eines Bugs, wodurch die Implementierung verkompliziert wurde.

Iteratee-Streams bestehen aus den drei Teilen `Enumerators`, `Iteratees` und `Enumeratees`, also den Datenquellen, den Datensenken und den Stream-Transformatoren. In dieser Arbeit wurde nicht nur gezeigt, wie diese Komponenten angewendet werden, sondern auch, wie eigene `Enumerators`, `Iteratees` und `Enumeratees` erstellt werden können. Dieser Blick auf die Implementierungsebene ermöglicht aus Sicht des Autors ein wesentlich umfassenderes Verständnis von Plays Iteratee-Streams.

Plays Implementierung der Iteratee-Streams weicht von der ursprünglichen Implementierung Kiselyovs vor allem in dem Punkt ab, dass sie nicht mit beliebigen Monaden, sondern nur mit `Future`-Monaden arbeitet. Der Grund hierfür wird sein, dass Play das Ziel hat, durch reaktive Programmierung, ohne zu Blockieren, möglichst effizient zu sein, weshalb sich die `Future`-Monade anbietet, auch wenn im Gegenzug Flexibilität eingebüßt wird. Dies ist aus Sicht des Autors vertretbar und hat für ihn beim Arbeiten mit Play keine Probleme dargestellt.

Dass durch diese Streams aber auch komplexere Prozesse als die erste Beispielanwendung implementiert werden können, wurde mit der Twitter News-Anwendung gezeigt. Dabei

wurden mit `Iteratees` Daten von der Twitter-API empfangen und verarbeitet. Mit verknüpften `Enumeratees` wurden weitere Tweets heruntergeladen und zwischengespeichert. Erst nach vollständiger Verarbeitung wurden diese Daten wieder als `Enumerators` der Außenwelt zur Verfügung gestellt und deren Elemente an die verbundenen Clients gesendet.

Play wird aktiv entwickelt und es wurden während der Entstehung dieser Arbeit mehrere neue Versionen veröffentlicht. Bei der Aktualisierung von Play 2.1 auf Play 2.2 mussten Teile des Codes angepasst werden, was bei einem solchen Versionssprung zwar zu erwarten ist, bei häufigen Versionsänderungen aber zur Last werden kann. Außerdem wurde beim Schreiben dieser Arbeit ein Bug entdeckt, der aber in der nächsten Version korrigiert wird. Durch die aktive Entwicklung und Erweiterung des Frameworks ist Play möglicherweise weniger stabil, als ältere Frameworks, bekanntgewordene Fehler werden aber dadurch auch schnell korrigiert. Insgesamt bilden die von Play zur Verfügung gestellten Werkzeuge aus Sicht des Autors eine solide Grundlage, um Real-Time-Web-Anwendungen in einem reaktiven Stil zu entwickeln.

6.2. Ausblick

In dieser Arbeit wurde das Hauptaugenmerk auf Play, also die Server-Seite, gerichtet und die Client-Seite nur oberflächlich behandelt. Durch Einsatz eines passenden JavaScript-Frameworks könnte der clientseitige Code wesentlich besser strukturiert werden, wodurch sich Anwendungen mit einer komplexeren Client-Seite besser umsetzen ließen. Eine weitere Arbeit könnte sich auf die Programmierung der Browser-Seite mit z. B. AngularJS (vgl. Google Inc. 2013) oder Ember.js (vgl. Tilde Inc. 2013) konzentrieren, die mit einem in Play geschriebenen Server-Backend kommuniziert.

Des Weiteren hat sich diese Arbeit fast ausschließlich mit den reaktiven Möglichkeiten des Play-Frameworks beschäftigt, weil diese für die Entwicklung von Real-Time-Web-Anwendungen nötig sind. Auf viele andere Teile von Play wurde gar nicht oder nur wenig eingegangen, weil dies den Rahmen dieser Arbeit gesprengt hätte. So wurde beispielsweise das Arbeiten mit Datenbanken komplett ignoriert und für Controller wurde nur die Programmierung einfacher Actions erklärt. Auch in dieser Richtung gibt es also weitere Möglichkeiten für Forschungen.

Play legt großen Wert auf reaktive Programmierung. Dieser Trend reduziert sich allerdings nicht alleine auf Play, sondern wird auch von Martin Odersky, dem Begründer von Scala, in einer Online-Vorlesung (vgl. Meijer u. a. 2013) gelehrt. Die in dieser Arbeit vorgestellten reaktiven Konzepte sind nur ein kleiner Teil der Dinge, die reaktive Programmierung ausmachen. Jonas Bonér, der Mitarbeiter bei Typesafe ist, hat unter Mitwirkung anderer Persönlichkeiten

der Scala-Community das Reactive Manifesto (vgl. Jonas Bonér et al. 2013) veröffentlicht, in dem die Kernpunkte reaktiver Programmierung beschrieben werden. Dieses Thema bietet genügend Stoff für eine weiterführende Arbeit.

A. Vorbereitung

In diesem Anhang geht es darum, wie die Entwicklungsumgebung aufgesetzt wird, mit welcher Version der eingesetzten Software gearbeitet wird und wie sich das Play-Framework installieren lässt.

A.1. Entwicklungsumgebung

Zum Anlegen von Play-Projekten und Steuern des Web-Servers wird eine Kommandozeile, wie die Eingabeaufforderung unter Windows oder dem Terminal und Mac OS X benötigt. Zum Programmieren kann entweder ein einfacher Text-Editor, oder auch eine IDE verwendet werden. Für die einzelnen Programme können teilweise Plugins heruntergeladen werden, die beispielsweise Code Completion oder Syntax Highlighting für Play-spezifische Funktionalitäten bereitstellen.

Für **Eclipse** kann die Scala IDE von Typesafe Inc. 2013b heruntergeladen werden. Diese IDE ist für die Entwicklung von Scala-Anwendungen mit Eclipse notwendig. Für die Scala IDE existiert außerdem ein Plugin für das Play-Framework. Eine genaue Installationsanleitung dafür ist unter Typesafe Inc. 2013c zu finden. Um ein Play-Projekt in Eclipse zu bearbeiten, muss dafür nach dem Anlegen des Projekts auf der Kommandozeile ein Eclipse-Projekt exportiert werden. Dazu muss in der Play-Konsole des Projekts der Befehl `eclipse` ausgeführt werden. Dieser Befehl muss jedes Mal ausgeführt werden, wenn die Konfigurationsdateien geändert werden. Anschließend kann der Ordner des Play-Projekts als Eclipse-Projekt importiert und geöffnet werden.

Beim Exportieren eines **IntelliJ IDEA**-Projekts verhält es sich ähnlich, wie im Falle von Eclipse. Der einzige Unterschied ist, dass der Befehel `idea` verwendet werden muss, um ein IntelliJ IDEA-Projekt zu erstellen. Für Text-Editoren, wie z.B. **VIM**, **Emacs** oder **Sublime Text** ist prinzipiell kein Plugin nötig. Falls aber ein Plugin für den verwendeten Editor existiert, kann es natürlich installiert werden. Weitere Informationen zu diesen und **anderen Entwicklungsumgebungen** sind in der offiziellen Dokumentation zu finden (vgl. Zenexity und Typesafe Inc. 2013n).

A.2. Software-Version

Zum Zeitpunkt dieser Arbeit ist die aktuellste stabile Version des Play-Frameworks die Version 2.2.1. Diese Version wird in allen Beispielen und der eigenen entwickelten Anwendung verwendet. Für den Einsatz des Frameworks wird das JDK in Version 6 oder 7 benötigt. Der Autor arbeitet mit dem JDK 7 unter Mac OS X. Die aktuell verwendete Scala-Version ist 2.10.2, diese ist unabhängig von der systemweit installierten Version und wird beim Anlegen eines Play-Projekts für das jeweilige Projekt installiert.

A.3. Installation

Zum Installieren von Play kann eine vorkompilierte Version der Software von der offiziellen Website unter Zenexity und Typesafe Inc. (2013d) heruntergeladen werden. Nach dem Entpacken der heruntergeladenen Datei kann das Programm namens `play`, das sich im entpackten Ordner befindet, auf der Kommandozeile verwendet werden. Um für die Verwendung des Programms nicht immer den absoluten Pfad angeben zu müssen, kann der Pfad zum entpackten Ordner in die Systemumgebungsvariable PATH eingetragen werden.

Unter **Mac OS X** muss dazu auf der Kommandozeile folgender Befehl ausgeführt werden: `export PATH=$PATH:<Pfad zum entpackten Ordner>`. Ein konkretes Beispiel könnte so aussehen: `export PATH=$PATH:~/Downloads/play-2.2.1`. Damit sich der Ordner auch nach dem Schließen des Terminals noch in der `PATH`-Variable befindet, sollte der obige Befehl zusätzlich in die `~/.profile`-Datei oder in die Konfigurationsdatei der verwendeten Shell eingetragen werden. Falls diese Datei nicht existiert, muss sie vorher angelegt werden.

Unter **Windows** muss dazu in der Eingabeaufforderung folgender Befehl ausgeführt werden: `setx PATH "%PATH%;c:\path\to\play" /m`, wobei `c:\path\to\play` durch den tatsächlichen Pfad zu ersetzen ist (vgl. Hilton u. a. 2013, S. 9).

B. Inhalt der beiliegenden CD

Dieser Arbeit liegt ein Datenträger mit folgender Verzeichnisstruktur bei:

/README.txt	die Verzeichnisstruktur der CD
/thesis.pdf	dieses Dokument im PDF-Format
/age_statistics_http/	die in Kap. 2 entwickelte Altersstatistikenanwendung
/age_statistics_sse/	die in Kap. 2 entwickelte Anwendung mit den in Unterabschnitt 4.2.3 eingeführten Server Sent Events-Erweiterungen
/age_statistics_ws/	die in Kap. 2 entwickelte Anwendung mit den in Unterabschnitt 4.1.3 eingeführten Web Sockets-Erweiterungen
/examples/	SBT-Projekt (vgl. Harrah 2013), das den in Kap. 3 gezeigten Code enthält (inklusive automatisierter Tests im `test`-Ordner)
/twitter_news/	die in Kap. 5 entwickelte Twitter News-Anwendung (mit Installationsanweisungen in der `README`)

Literaturverzeichnis

[Aceto u. a. 2013] Aceto, Luca ; Goriac, Eugen-Ioan ; Ingólfsdóttir, Anna: SOS Rule Formats for Idempotent Terms and Idempotent Unary Operators. In: Emde Boas, Peter van (Hrsg.) ; Groen, Frans C. A. (Hrsg.) ; Italiano, Giuseppe F. (Hrsg.) ; Nawrocki, Jerzy R. (Hrsg.) ; Sack, Harald (Hrsg.): *SOFSEM* Bd. 7741, Springer, 2013, S. 108–120. – URL http://www.ru.is/~luca/PAPERS/idemop.pdf. – ISBN 978-3-642-35842-5, 978-3-642-35843-2

[Barr und Wells 1999] Barr, Michael ; Wells, Charles: Category Theory - Lecture Notes for ESSLLI. (1999). – URL http://www.math.upatras.gr/~cdrossos/Docs/B-W-LectureNotes.pdf. – Zugriffsdatum: 2013-09-12

[Bidelman 2010] Bidelman, Eric ; Rocks, HTML5 (Hrsg.): *Server Sent Events vs. WebSockets*. 2010. – URL http://www.html5rocks.com/en/tutorials/eventsource/basics/. – Zugriffsdatum: 2013-10-15

[Bozdag u. a. 2007] Bozdag, E. ; Mesbah, A. ; Deursen, A. van: A Comparison of Push and Pull Techniques for AJAX. In: *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4380239. – Zugriffsdatum: 2013-04-11, 2007, S. 15–22

[Chen 2000] Chen, Yifeng: Specification for reactive bulk-synchronous programming. In: *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=823411&isnumber=17807. – Zugriffsdatum: 2013-11-07, 2000, S. 190–196. – ISSN 1066-6192

[Deveria 2013a] Deveria, Alexis: *Can I use Server-sent DOM events?* 2013. – URL http://caniuse.com/eventsource. – Zugriffsdatum: 2013-10-01

[Deveria 2013b] Deveria, Alexis: *Can I use Web Sockets?* 2013. – URL http://caniuse.com/websockets. – Zugriffsdatum: 2013-10-01

[Eriksen 2012] Eriksen, Marius ; Twitter Inc. (Hrsg.): *Effective Scala*. 2012. – URL http://twitter.github.io/effectivescala/#Functional%20programming-Case%20classes%20as%20algebraic%20data%20types. – Zugriffsdatum: 2013-05-28

[Google Inc. 2013] Google Inc.: *AngularJS*. 2013. – URL http://angularjs.org/. – Zugriffsdatum: 2013-11-06

[Haller u. a. 2013] Haller, Philipp ; Prokopec, Aleksandar ; Miller, Heather ; Klang, Viktor ; Kuhn, Roland ; Jovanovic, Vojin: *Futures and Promises*. 2013. – URL http://docs.scala-lang.org/overviews/core/futures.html. – Zugriffsdatum: 2013-10-07

[Harrah 2013] Harrah, Mark: *Simple Build Tool*. 2013. – URL http://www.scala-sbt.org/. – Zugriffsdatum: 2013-11-07

[Hilton u. a. 2013] Hilton, Peter ; Bakker, Erik ; Canedo, Francisco: *Play for Scala : Covers Play 2*. 1. Aufl. Shelter Island, New York : Manning Publications Co., 2013. – ISBN 9781617290794

[Hudak u. a. 2007] Hudak, Paul ; Hughes, John ; Peyton Jones, Simon ; Wadler, Philip: A history of Haskell: being lazy with class. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA : ACM, 2007 (HOPL III), S. 12–1–12–55. – URL http://doi.acm.org/10.1145/1238844.1238856. – Zugriffsdatum: 2013-05-28. – ISBN 978-1-59593-766-7

[IETF 2013] IETF: *The WebSocket Protocol*. 2013. – URL http://www.rfc-editor.org/rfc/pdfrfc/rfc6455.txt.pdf. – Zugriffsdatum: 2013-10-01

[Jonas Bonér et al. 2013] Jonas Bonér et al.: *The Reactive Manifesto*. 2013. – URL http://www.reactivemanifesto.org/. – Zugriffsdatum: 2013-11-06

[Kilp u. a. 2000] Kilp, M. ; Knauer, U. ; Mikhalev, A.V.: *Monoids, Acts and Categories: With Applications to Wreath Products and Graphs : a Handbook for Students and Researchers*. de Gruyter, 2000 (De Gruyter expositions in mathematics). – ISBN 9783110152487

[Kiselyov 2012a] Kiselyov, Oleg: *Incremental multi-level input processing and collection enumeration*. 2012. – URL http://okmij.org/ftp/Streams.html. – Zugriffsdatum: 2013-04-25

[Kiselyov 2012b] Kiselyov, Oleg: Iteratees. In: *Proceedings of the 11th international conference on Functional and Logic Programming*. Berlin, Heidelberg : Springer-Verlag, 2012 (FLOPS'12), S. 166–181. – URL http://okmij.org/ftp/Haskell/Iteratee/describe.pdf. – Zugriffsdatum: 2013-04-25. – ISBN 978-3-642-29821-9

[Lato 2010] Lato, John W.: Iteratee: Teaching an Old Fold New Tricks. In: *The Monad.Reader* (2010), Nr. 16, S. 19–35. – URL http://themonadreader.files.wordpress.com/2010/05/issue16.pdf. – Zugriffsdatum: 2013-04-25

[Mac Lane 1998] Mac Lane, Saunders: *Categories for the Working Mathematician*. 2nd. New York : Springer-Verlag, September 1998. – URL http://www.maths.ed.ac.uk/~aar/papers/maclanecat.pdf. – Zugriffsdatum: 2013-08-29. – ISBN 0-387-98403-8

[Meijer u. a. 2013] Meijer, Erik ; Odersky, Martin ; Kuhn, Roland: *Principles of Reactive Programming*. 2013. – URL https://class.coursera.org/reactive-001. – Zugriffsdatum: 2013-11-06

[Moggi 1989] Moggi, E.: Computational lambda-calculus and monads. In: *Proceedings of the Fourth Annual Symposium on Logic in computer science*. Piscataway, NJ, USA : IEEE Press, 1989, S. 14–23. – URL http://ieeexplore.ieee.org/ielx2/249/1547/00039155.pdf?tp=&arnumber=39155&isnumber=1547. – Zugriffsdatum: 2013-08-29. – ISBN 0-8186-1954-6

[Nehlsen 2013a] Nehlsen, Matthias: *BirdWatch*. 2013. – URL https://github.com/matthiasn/BirdWatch. – Zugriffsdatum: 2013-11-06

[Nehlsen 2013b] Nehlsen, Matthias: *Server Sent Events vs. WebSockets*. 2013. – URL http://matthiasnehlsen.com/blog/2013/05/01/server-sent-events-vs-websockets/. – Zugriffsdatum: 2013-10-15

[Odersky und Spoon 2010] Odersky, Martin ; Spoon, Lex: *Package Objects*. 2010. – URL http://www.scala-lang.org/docu/files/packageobjects/packageobjects.html. – Zugriffsdatum: 2013-08-09

[Odersky u. a. 2010] Odersky, Martin ; Spoon, Lex ; Venners, Bill: *Programming in Scala: Second Edition*. 2. Ausg. USA : Artima Press, 2010. – ISBN 0981531644

[O'Sullivan u. a. 2008] O'Sullivan, Bryan ; Goerzen, John ; Stewart, Don: *Real World Haskell*. 1st. O'Reilly Media, Inc., 2008. – URL http://pv.bstu.ru/flp/RealWorldHaskell.pdf. – Zugriffsdatum: 2013-09-05. – ISBN 0596514980, 9780596514983

[Pucella 1998] Pucella, R.R.: Reactive programming in Standard ML. In: *Computer Languages, 1998. Proceedings. 1998 International Conference on*, URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=674156&isnumber=14813. – Zugriffsdatum: 2013-11-07, 1998, S. 48–57. – ISSN 1074-8970

[Rompf u. a. 2009] Rompf, Tiark ; Maier, Ingo ; Odersky, Martin: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: *SIGPLAN Not.* 44 (2009), August, Nr. 9, S. 317–328. – URL http://doi.acm.org/10.1145/1631687.1596596. – Zugriffsdatum: 2013-07-02. – ISSN 0362-1340

[Taivalsaari und Mikkonen 2011] Taivalsaari, A. ; Mikkonen, T.: The Web as an Application Platform: The Saga Continues. In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6068340. – Zugriffsdatum: 2013-04-11, 2011, S. 170–174

[Tilde Inc. 2013] Tilde Inc.: *Ember.js*. 2013. – URL http://emberjs.com/. – Zugriffsdatum: 2013-11-06

[Twitter 2012] Twitter: *The Streaming APIs*. 2012. – URL https://dev.twitter.com/docs/streaming-apis. – Zugriffsdatum: 2013-10-27

[Twitter 2013] Twitter: *Connecting to a streaming endpoint*. 2013. – URL https://dev.twitter.com/docs/streaming-apis/connecting. – Zugriffsdatum: 2013-10-28

[Typesafe Inc. 2013a] Typesafe Inc.: *Futures*. 2013. – URL http://doc.akka.io/docs/akka/2.2.1/scala/futures.html. – Zugriffsdatum: 2013-10-07

[Typesafe Inc. 2013b] Typesafe Inc.: *Scala IDE*. 2013. – URL http://scala-ide.org/. – Zugriffsdatum: 2013-07-30

[Typesafe Inc. 2013c] Typesafe Inc.: *Setup and use Play framework 2.1 in Scala IDE 3.0*. 2013. – URL http://scala-ide.org/docs/tutorials/play/. – Zugriffsdatum: 2013-07-30

[W3C 2012a] W3C: *Server-Sent Events*. 2012. – URL http://www.w3.org/TR/eventsource/. – Zugriffsdatum: 2013-09-28

[W3C 2012b] W3C: *The WebSocket API*. 2012. – URL http://www.w3.org/TR/websockets/. – Zugriffsdatum: 2013-09-16

[W3C 2013] W3C: *HTML 5.1 Nightly EventHandlers*. 2013. – URL http://www.w3.org/html/wg/drafts/html/master/webappapis.html#eventhandler. – Zugriffsdatum: 2013-09-25

[Wadler 1990] Wadler, Philip: Comprehending monads. In: *Proceedings of the 1990 ACM conference on LISP and functional programming*. New York, NY, USA : ACM, 1990 (LFP '90), S. 61–78. – URL http://doi.acm.org/10.1145/91556.91592. – Zugriffsdatum: 2013-08-29. – ISBN 0-89791-368-X

[Zenexity und Typesafe Inc. 2011] Zenexity ; Typesafe Inc.: *Play Enumeratee map-Signaturänderung*. 12 2011. – URL https://github.com/playframework/Play20/commit/d2053af2517eddc93c1af6b05e246d470e976535#L1L239. – Zugriffsdatum: 2013-06-18

[Zenexity und Typesafe Inc. 2013a] Zenexity ; Typesafe Inc.: *Actions, Controllers and Results*. 2013. – URL http://www.playframework.com/documentation/2.1.x/ScalaActions. – Zugriffsdatum: 2013-08-07

[Zenexity und Typesafe Inc. 2013b] Zenexity ; Typesafe Inc.: *Anatomy of a Play application*. 2013. – URL http://www.playframework.com/documentation/2.1.x/Anatomy. – Zugriffsdatum: 2013-08-02

[Zenexity und Typesafe Inc. 2013c] Zenexity ; Typesafe Inc.: *Comet sockets*. 2013. – URL http://www.playframework.com/documentation/2.2.x/ScalaComet. – Zugriffsdatum: 2013-10-04

[Zenexity und Typesafe Inc. 2013d] Zenexity ; Typesafe Inc.: *Download Play Framework*. 2013. – URL http://www.playframework.com/download. – Zugriffsdatum: 2013-07-30

[Zenexity und Typesafe Inc. 2013e] Zenexity ; Typesafe Inc.: *EventSource Java library should not escape message content*. 2013. – URL https://github.com/playframework/playframework/issues/1862. – Zugriffsdatum: 2013-11-19

[Zenexity und Typesafe Inc. 2013f] Zenexity ; Typesafe Inc.: *Handling data streams reactively: Enumerators*. 2013. – URL http://www.playframework.com/documentation/2.1.1/Enumerators. – Zugriffsdatum: 2013-04-20

[Zenexity und Typesafe Inc. 2013g] Zenexity ; Typesafe Inc.: *Handling data streams reactively: Iteratees*. 2013. – URL http://www.playframework.com/documentation/2.1.1/Iteratees. – Zugriffsdatum: 2013-04-20

[Zenexity und Typesafe Inc. 2013h] Zenexity ; Typesafe Inc.: *Handling data streams reactively: The realm of Enumeratees*. 2013. – URL http://www.playframework.com/documentation/2.1.1/Enumeratees. – Zugriffsdatum: 2013-10-15

[Zenexity und Typesafe Inc. 2013i] Zenexity ; Typesafe Inc.: *Play Enumeratee Source Code*. 2013. – URL https://github.com/playframework/playframework/blob/2.2.0/framework/src/iteratees/src/main/scala/play/api/libs/iteratee/Enumeratee.scala. – Zugriffsdatum: 2013-06-09

[Zenexity und Typesafe Inc. 2013j] Zenexity ; Typesafe Inc.: *Play Enumerator Source Code*. 2013. – URL https://github.com/playframework/playframework/blob/2.2.0/framework/src/iteratees/src/main/scala/play/api/libs/iteratee/Enumerator.scala. – Zugriffsdatum: 2013-06-05

[Zenexity und Typesafe Inc. 2013k] Zenexity ; Typesafe Inc.: *Play Framework*. 2013. – URL http://www.playframework.com/. – Zugriffsdatum: 2013-11-07

[Zenexity und Typesafe Inc. 2013l] Zenexity ; Typesafe Inc.: *Play Iteratee Package - API-Dokumentation*. 2013. – URL http://www.playframework.com/documentation/api/2.1.1/scala/index.html#play.api.libs.iteratee.package. – Zugriffsdatum: 2013-07-03

[Zenexity und Typesafe Inc. 2013m] Zenexity ; Typesafe Inc.: *Play Iteratee Source Code*. 2013. – URL https://github.com/playframework/playframework/blob/2.2.0/framework/src/iteratees/src/main/scala/play/api/libs/iteratee/Iteratee.scala. – Zugriffsdatum: 2013-10-09

[Zenexity und Typesafe Inc. 2013n] Zenexity ; Typesafe Inc.: *Setting up your preferred IDE*. 2013. – URL http://www.playframework.com/documentation/2.1.x/IDE. – Zugriffsdatum: 2013-07-30

[Zenexity und Typesafe Inc. 2013o] Zenexity ; Typesafe Inc.: *Streaming HTTP responses*. 2013. – URL http://www.playframework.com/documentation/2.2.x/ScalaStream. – Zugriffsdatum: 2013-10-01

[Zenexity und Typesafe Inc. 2013p] Zenexity ; Typesafe Inc.: *The template engine*. 2013. – URL http://www.playframework.com/documentation/2.1.x/ScalaTemplates. – Zugriffsdatum: 2013-08-08

[Zenexity und Typesafe Inc. 2013q] Zenexity ; Typesafe Inc.: *Using the Play console*. 2013. – URL http://www.playframework.com/documentation/2.1.x/PlayConsole. – Zugriffsdatum: 2013-08-03

[École Polytechnique Fédérale de Lausanne 2013] École Polytechnique Fédérale de Lausanne: *NonFatal*. 2013. – URL http://www.scala-lang.org/api/2.10.2/#scala.util.control.NonFatal$. – Zugriffsdatum: 2013-10-07

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 21. November 2013 Till Theis