Projektgruppe Web-Technologie: EpisodeFever

Pascal Hertleif Andreas Diesendorf

1. März 2015

Inhaltsverzeichnis

1 Ziele

Ziel des Projekts "EpisodeFever" ist es, eine Plattform zur Bewertung von Fernseh-Serien zu erstellen.

Dazu sollen Daten über Serien, Episoden, Benutzer und Bewertungen gespeichert werden. Die Daten über Serien und Episoden sollen von externen Diensten abgefragt bzw. aktualisiert werden. Es soll möglich sein, Benutzer-Konten zu erstellen und zu verwalten, als Benutzer Zugriff auf ausgewählte Daten zu haben und Bewertungen zu Episoden abzugeben.

Im Rahmen des Projekts soll eine JSON-Schnittstelle erstellt werden, über welche die genannten Daten abgefragt und verändert werden können. Diese soll den Prinzipien von REST folgen [1] und es ermöglichen, dass verschiedene Anwendungen darauf zugreifen.

Des Weiteren soll das Projekt dazu dienen, das Team mit den verwendeten Technologien vertraut zu machen und eine effiziente Architektur für *node.js*-basierte Server-Anwendungen zu finden. Dies beinhaltet auch das Schreiben von automatisierten Tests für die implementierte Software. Durch das Programmieren wiederverwendbarer Module soll außerdem die Entwicklung zukünftiger Systeme vereinfacht werden.

2 Überblick über die verwendeten Technologien

Falls nichte explizit erwähnt, ist jede im weiteren Verlauf erwähnte Software unter einer OpenSource-Lizenz verfügbar (vgl. [2]).

Als Software-Plattform wurde *node.js* ("node", [3]) gewählt[1], welches JavaScript-Applikationen ausführt und Zugriff auf System-Schnittstellen bietet. Da es auf Basis eines Event-Loop arbeitet, geschieht jeglicher Zugriff auf System-Ressourcen wie Dateisystem oder Netzwerk asynchron. Es sind sehr viele[2] mit *node.js* kompatible Bibliotheken über *npm* [4] verfügbar, welche leicht in eigene Software integriert werden können.

Zum Speichern und Abfrage von Daten wurde *PostgreSQL* ("Postgres", [5]) ausgewählt, ein stabiles und performantes relationales Datenbanksystem. Die Alternative *MongoDB* [6], ein Dokument-basiertes ("NoSQL") Datenbanksystem, wurde ebenfalls betrachtet. Da die zu speichernden Daten in einem fest vorgegeben Schema vorliegen und untereinander verknüpft sind (es aber nicht sinnvoll ist, die verknüpften Daten einzubetten), erschienen mögliche Vorteile eines NoSQL-Ansatzes für nicht relevant[3]. Neuere Versionen von PostgreSQL unterstützen außerdem NoSQL-ähnliche Funktionen wie das Speichern und Abfragen von JSON-Strukturen (seit Version 9.3) sowie das Erzeugen von Volltext-Such-Indizes (seit Version 8.3). Diese Funktionen sind auch Gründe, warum nicht ein anderes SQL-Datenbanksystem wie MariaDB [7] verwendet wurde.

2.1 Node.js-spezifische Module

express.js in Version 4 [8] wird zur Abstraktion der über HTTP bereitgestellten Ressourcen verwendet. Mit Hilfe von "Middlewares" lassen sich HTTP-Anfragen in mehreren Stufen verarbeiten und Antworten senden. Mit express wird auch der HTTP-Server selbst gestartet.

Alternativen zu *express.js* sind *restify* [9], *Hapi* [10] oder *koa* [11], welche ebenfalls auf *node.js* aufsetzen. Ansonsten hätte man auch *Ruby on Rails* (Ruby, [12]), *Django* (Python, [13]), *Laravel*

[1] Da es ein explizites Ziel des Projektes war, eine Plattform auf Basis von *node.js* zu entwickeln, wurden Alternativen nicht weiter betrachtet. Da *node.js* im Grunde eine Bibliothek und Laufzeit-Umgebung für JavaScript ist, sind mögliche Alternativen andere Programmiersprachen wie PHP, Ruby oder Python, welche ähnlich Schnittstellen bieten, aber auch Java, C# oder C++ mit entsprechenden Bibliotheken.

[2] Am 7. Februar 2015 01:32 Uhr (MEZ) waren laut *npmjs.org* [4] 123.800 Pakete verfügbar.

[3] Weitere Informationen zum verwendeten Datenbank-Schema sind weiter unten (Kapitel 3.1.1, Seite 4) zu finden.

(PHP, [14]) oder *Martini* (Go, [15]) wählen können. Express wurde ausgewählt, da es weit verbreitet ist (weshalb viele daran angepasste Module und Dokumentation verfügbar sind) und es seit Version 4 möglich ist, Applikationen aus mehreren *Router*-Instanzen zusammenzusetzen (was modulare Applikationen ermöglicht).

Um Zugriffe auf die Datenbank zu vereinfachen und ausgelesene Daten direkt verarbeiten zu können, wird *bookshelf.js* [16] eingesetzt. Dieses erlaubt es, Datenbank-Inhalt wie JavaScript-Objekte zu verwenden und Relationen im Code abzubilden. Intern wird *knex.js* [17] verwendet, um SQL-Abfragen zu generieren und Schema-Migrationen durchzuführen. Alternative hierzu ist vor allem *mongoose* [18], wäre statt PostgreSQL MongoDB gewählt worden.

2.2 Module bezüglich Code-Struktur

2.2.1 Promises

Ebenso wie die in *node.js* integrierten Module, verwenden auch viele externe Bibliotheken *Callbacks*, um Rückgabewerte asynchroner Schnittstellen zu übertragen. Dies führt bei vielen voneinander abhängigen Aufrufen zu Software, deren Programmfluss auf Grund von ineinander verschachtelten Funktionsaufrufen schwer nachzuvollziehen muss. Ebenso ist das Behandeln aller Fehlerfälle in solchen Programmen oft komplex.

Viele dieser Probleme werden durch den Einsatz von Promises [19] gelöst. Die Standard-konforme Implementierung *bluebird* [20] wird in EpisodeFever verwendet, da sie sehr performant ist [21] und viele Hilfsfunktionen mitliefert (um beispielsweise in *node* integrierte Module mit Promises zu verwenden).

2.2.2 Tests

Das Schreiben und Ausführen von automatisierten Tests wird durch *mocha* ermöglicht, einem Test-Framework, welches mit dem von Ruby bekannten *rspec* [22] vergleichbar ist. Alternativen sind *Jasmine* [23] oder *Vows* [24].

Alle drei verwenden die für Behavior-driven Development [25] typischen Bezeichnungen (Tests sind Blöcke mit den Funktionen `describe` und `it`). Während *Jasmine* sich sehr ähnlich zu *mocha* verhält und ähnliche Interfaces bietet, ist es nicht so mächtig; es fehlt beispielsweise die Unterstützung Promises als Rückgabewerte von Tests auszuwerten, was asynchrone Tests sehr verkürzt. *Vows* fokussiert sich auf asynchrone Tests, die parallel ausgeführt werden können. Dies scheint für Unit-Tests hilfreich zu sein, bei Integrationstests wie dem Abfragen einer REST-API und dem Überprüfen von Datenbank-Inhalten führt dies aber dazu, dass Abhängigkeiten von Tests explizit angebeben werden müssen.

Die Bibliothek *chai.js* [26] bietet eine Vielzahl von Hilfs-Funktionen, mit welchen Werte von Objekten überprüft werden können. Da im Projekt nur die Funktionen basierend auf `expect` verwendet werden, wurde zunächst überlegt, das minimlistische Modul *expect.js* [27] zu verwenden. Letztendlich bietet *chai.js* jedoch mehr Funktionen, wird aktivier entwickelt und hat eine Schnittstelle für Erweiterungen (sodass beispielsweise die Datentypen von *bookshelf* einfacher geprüft werden können).

Um Anfragen an den zu testenden Teil des Servers zu simulieren, wurde *supertest* [28] eingesetzt. *Supertest* basiert auf *superagent* [29], welches schon für den Zugriff auf externe APIs verwendet wird, wie in “XML-Daten abfragen und verarbeiten” (Kapitel 3.6.1, Seite 12) beschrieben. *Supertest* bietet einfache Möglichkeiten zur Überprüfung von HTTP-Antworten. Es wurde um die Verwendung von *Promises* erweitert, sodass asynchrone Tests einfach zu schreiben sind. (Ist der Rückgabe-Wert eines Tests ein Promise, wird dieses von *mocha* automatisch ausgewertet.)

3 Vorgehen bei der Projekt-Umsetzung

3.1 Informations-Architektur

EpisodeFever soll sowohl *Serien*- als auch *Episoden*-Daten umfassen. Zusätzlich soll es *Benutzer* geben, welche *Bewertungen* anlegen können. Diese vier Entitäten sollen über die API verfügbar sein.

Serien und Episoden können nur gelesen werden; Aktualisierung dieser Daten findet über das Import-Modul (Kapitel 3.6, Seite 12) statt. Benutzer können erstellt und (eingeschränkt) bearbeitet werden. Zudem können Benutzer pro Episode eine Bewertung abgeben.

3.1.1 Datenbank-Schema

Aus den Ansprüchen an die API lässt sich ableiten, welche Daten zur Verfügung stehen müssen. Daraus lässt sich wiederum ein Datenbank-Schema entwerfen. Dieses sollte normalisiert und erweiterbar sein [30 S. 75–81].

Wie oben (Kapitel 2, Seite 2) erwähnt, wurde als Alternative zu einer SQL-Datenbank auch die NoSQL-Datenbank MongoDB in Betracht gezogen. Diese eignet sich besonders, wenn zu bestimmten Datensätzen immer zusätzliche Relationen ausgelesen werden. Würden beispielsweise zu Serien immer alle Episoden geladen werden, könnte man in MongoDB die Episoden in das Serien-Dokument einbetten.

In EpisodeFever ist dies aber nicht gegeben. Episoden sollten keine eingebetteten Dokumente sein, da es möglich sein soll, die in den nächsten Tagen ausgestrahlten Episoden leicht zu bestimmen. Ebenso können Bewertungen nicht problemlos in andere Dokument eingebettet werden (z.B. als Teil der bewerteten Episode oder des bewertenden Benutzers), da man sie im Kontext eines Benutzers, einer Episode oder einer Serie abfragen können soll. Aus diesen Gründen erschien eine SQL-Datenbank die bessere Wahl zu sein.

Das Diagramm “EpisodeFevers Datenbankschema” (Abbildung 1, Seite 5) stellt das verwendete Schema als gerichteten Graphen dar. Verbindung zwischen Feldern symbolisieren Relationen zwischen Tabellen (mit Foreign Keys). Die Episoden-Tabelle beinhaltet z.B. eine Referenz auf einen Eintrag der Serien-Tabelle.

Zu den vier abgebildeten Tabellen existiert auch noch eine `knex_migrations`-Tabelle. Diese wird von *knex* [17] automatisch angelegt und gefüllt, um festzuhalten, welche Schema-Migrationen in dieser Datenbank bereits durchgeführt wurden.

3.2 Projekt-Struktur: Verzeichnis-Struktur, Aufteilung nach Services

Zu Beginn des Projektes wurde neben der Informationsarchitektur auch beschlossen, wie die Code-Struktur sein soll. Diese lässt sich gut durch die verwendete Verzeichnisstruktur (Abbildung 2, Seite 5) darstellen.

3.2.1 Services

Die Anwendung wurde in verschiedene *Services* aufgeteilt, welche möglichst isoliert von einander funktionsfähig sein sollen. Deren Code ist in einzelnen Verzeichnissen in `server/services/` zu finden.

Diese Aufteilung soll es ermöglichen, einzelne Teile der Applikation einfacher überblicken und getrennt voneinander entwickeln zu können. Idealerweise sind Services so modular aufgebaut, dass sie in zukünftigen Projekten wiederverwendet werden können.

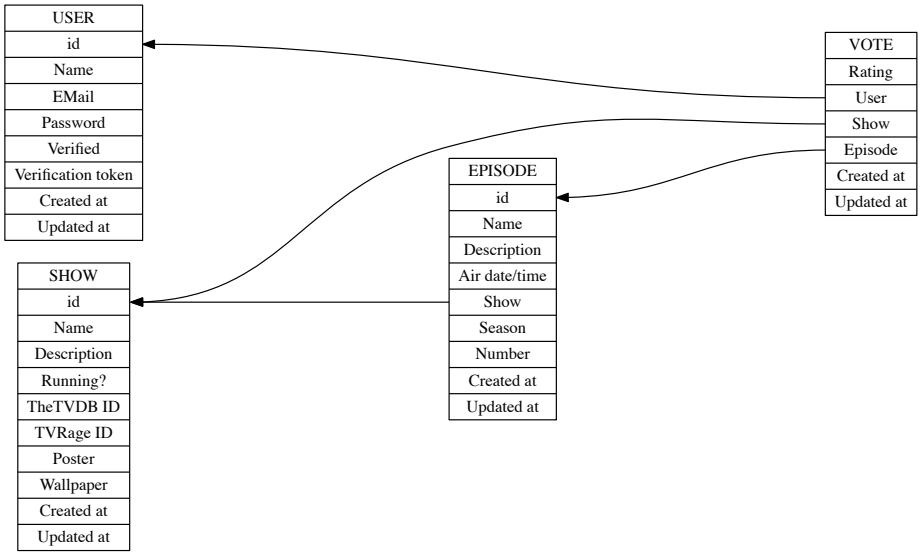

Abbildung 1: EpisodeFevers Datenbankschema

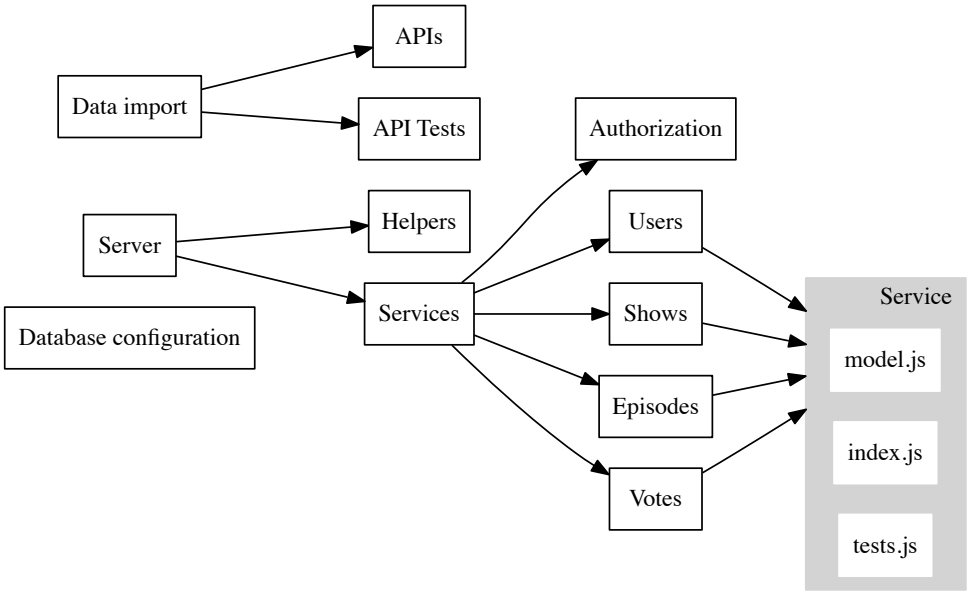

Abbildung 2: Verzeichnisstruktur

Wie in der Verzeichnisstruktur (Abbildung 2, Seite 5) zu sehen ist, sind Kern-Bestandteile eines Services:

- der Einstiegspunkt (`index.js`), welcher die Schnittstellen des Services exportiert,
- das Daten-Modell (`model.js` über welches Abfragen und Änderungen an der Datenbank ausgeführt werden,
- Tests (`tests.js`), welche beschreiben, was der Service behandelt und sicherstellen, dass keine Regressionen auftreten.

3.2.2 Alternativen

Eine alternative Strukturierung ist das Gruppieren nach Typen, d.h. in Verzeichnisse wie `models`, `controllers` und `specs` (Tests). Dies ist beispielsweise bei Anwendungen basierend auf *Ruby on Rails* [12] typisch. Da dies auf den Code aber nur minimale Auswirkungen hat, ist es letztendlich Geschmacks-Sache. Wir entschieden uns für die oben beschriebe Aufteilung, da diese die inhaltliche Aufteilung in den Vordergrund stellt, nicht die strukturelle.

3.3 Benutzer-Authentifizierung

Schon im Planungsstadium des Projekts war vorgesehen, dass es eine Benutzer-Authentifizierung geben muss, da nur registrierte Benutzer die Möglichkeit haben sollen abzustimmen.

Dafür ist der *Auth* Service zuständig, der im wesentlichen die folgenden Module umfasst:

- *Register.* Die Registrierung der Benutzer unter Angabe der E-Mail Adresse, des Passworts und des Benutzernamen. Die Validierung der Eingabe und bei Erfolg das Speichern der Daten in der Datenbank, wobei das Passwort verschlüsselt hinterlegt wird. Anschließend das versenden einer E-Mail mit einem Verifizierungs-Link, der den Benutzer verifiziert und die Registrierung abschließt.
- *Login.* Das Anmelden mit dem der E-Mail Adresse. Wobei der Benutzer bei erfolgreicher Anmeldung einen Session Token bekommt.
- *Verify.* Die Session-Verwaltung, die sicherstellt, dass der Benutzer einen validen Session-Token bekommt, vorausgesetzt der Benutzer ist verifiziert.

Nachfolgend ein detaillierter Einblick in die einzelnen Module.

3.3.1 Registrierung

Die Registrierung läuft folgendermaßen ab.

Der Benutzer gibt seine Daten ein (Name, E-Mail, Password) und gibt beim Absenden diese als POST Request an den Server. Die Daten werden aus dem Request extrahiert, normalisiert und validiert.

Für die Validierung wurde die Library *CheckIt* [31] verwendet. Wir entschieden uns für *CheckIt*, da es vom selben Autor wie *bookshelf* und *knex* ist (Tim Griesser) und daher gut mit diesen zusammen arbeitet. Alternativen sind *Joi*[4] oder *is-my-json-valid*[5].

CheckIt ermöglicht es, Javascript Objekte zu validieren, indem für die Daten bestimmte Formate und Anforderungen definiert werden. Diese sind z.B. bei uns, dass das Feld E-Mail auch das Format

[4] https://www.npmjs.com/package/joi
[5] https://github.com/mafintosh/is-my-json-valid

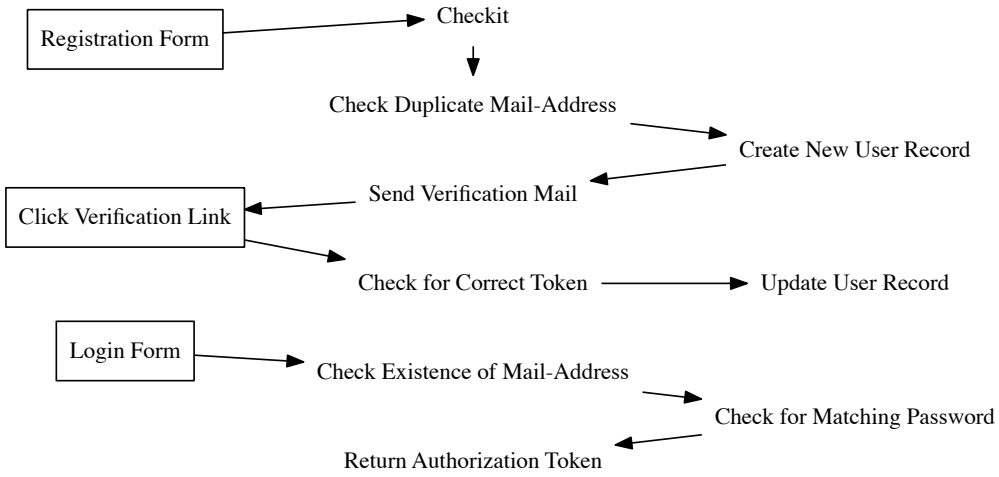

Abbildung 3: Authorisierungs-Ablauf

einer E-Mail Adresse hat, das Passwort aus mindestens 6 Zeichen besteht und kein Felder leer sein darf. Damit stellen wir sicher, dass keine Spam-Anmeldungen möglich sind.

Wenn die vorliegenden Daten den geforderten Standards entsprechen, wird überprüft, ob die Daten nicht bereits existieren. Sollte das nicht der Fall sein, wird das Passwort an einen Helper[6] übergeben, der mittels *bcrypt* [32], einen Salt generiert und aus diesem und dem Passwort einen Hash-Wert ermittelt[7]. Zusätzlich wird ein Token für die Verifizierung erzeugt. Der generierte Hash, sowie der Token werden dann zusammen mit den eingegebenen Daten in der Datenbank hinterlegt und anschließend eine E-Mail mit einem Verifizierungs-Link versandt, welcher den Verifizierungs-Token in URL-geeigneter Form enthält.

Sobald dieser Link aufgerufen wird, wird der mit dem Token verbundene Account als "verifiziert" markiert. Die Registrierung ist damit abgeschlossen.

3.3.2 Login

Ursprünglich wollten wir die Open Source Library *Passport* [34] für die Anmeldung verwenden, doch im Laufe des Projekts haben wir festgestellt, dass dies kaum Vorteile bringt, da wir den größten Teil der Funktionalitäten trotzdem selber schreiben mussten. So stellte *Passport* beispielsweise nur Cookies für Sessions zur Verfügung, sodass wir die Generierung von Session-Tokens selbst schreiben mussten. Selbst für die Überprüfung der Login Daten haben wir festgestellt, dass die Benutzung von *Passport* keine wesentlichen Vorteile gegenüber einer selbst geschriebenen Lösung bietet und haben deshalb beschlossen, auf die zusätzliche Abhängigkeit zu verzichten, die Library nicht zu verwenden und stattdessen eine eigene Lösung zu schreiben. Wäre ein Anmeldung über Soziale Dienste wie *Facebook* oder *Twitter* vorgesehen, würde sich eine erneute Betrachtung, der Library unter Umständen lohnen, da *Passport* dazu eine Schnittstelle bereitstellt. Doch da dies nicht als Feature vorgesehen war, hatte das keinen Einfluss auf unsere Entscheidung.

Die Anmeldung wurde von uns auf folgende Art gelöst.

[6] `server/services/auth/password_helper.js`
[7] Für genauere Beschreibung und Analyse dieses Vorgehens siehe Referenz 33.

Der Benutzer gibt seine Anmeldedaten (E-Mail und Passwort) ein und gibt beim Absenden den POST Request an den Server. Die versendeten Daten werden aus dem Request extrahiert. Mit den extrahierten Daten wird zuerst überprüft, ob es in der Datenbank einen Benutzer mit der angegebenen E-Mail gibt und ob dieser Benutzer verifiziert ist. Sollte dies der Fall sein wird ein *User*-Objekt erstellt, welches die Informationen über den Benutzer aus der Datenbank enthält.

Ein Helper[8] überprüft, dann ob das vom Benutzer eingegebene Passwort mit dem hinterlegten Passwort übereinstimmt. Da das in der Datenbank gespeicherte Passwort gehasht vorliegt, benutzen wir die compare Methode von bcrypt, die uns erlaubt das eingegebene Passwort mit dem gehashten Passwort abzugleichen. Stimmen diese überein wird ein Session-Token generiert und zurückgegeben. Damit ist der Anmeldeprozess abgeschlossen.

3.3.3 Verifizierung

Die Verifizierung wurde bereits bei der Planung vorgesehen, da wir sicherstellen wollten, dass die bei der Registrierung eingegebenen E-Mail-Adressen auch wirklich existieren und der Benutzer dies durch das Klicken auf einen Verifizierungs-Link bestätigt. Wir sahen diese Funktionalität deshalb als wichtig an, da wir die Erstellung von Spam-Accounts einschränken wollten.

Die Verifizierung läuft wie folgt ab.

Wie bereits vorher erläutert, wurde beim Registrieren ein Verifizierungs-Token erstellt und in der Datenbank abgelegt. Dieser Token wird dann mithilfe der Library *jsonwebtoken* [35] in einen sogenannten JSON-Web-Token (JWT) konvertiert (eine serialisierte und kryptographisch signierte From eines JSON-Objektes), welcher den zuvor generierten Verifizierungs-Token und die ID des Benutzers enthält. Der JWT wird dann an die vom Benutzer eingegebene E-Mail in Form eines Verifizierungs-Links gesendet.

Zum Versenden der E-Mail haben wir die *Nodemailer*-Library [36] benutzt, weil diese sehr leicht zu implementieren war und viele Funktionalitäten bot (um beispielsweise mit externen Mailling-Anbietern wie *Sendgrid*[9] zu kommunizieren, welche neben SMTP auch weitere Funktionen über eine REST-API bieten).

Beim Aufruf des Verifizierungs-Links dekodieren wir zuerst den JWT, extrahieren daraus die Benutzer ID und den Verifizierungs-Token und überprüfen, ob die Daten in valider Form vorliegen. Dann überprüfen wir, ob der Benutzer bereits verifiziert ist und ob der extrahierte Token mit dem Token in der Datenbank übereinstimmen. Falls der Benutzer nicht bereits verifiziert und der Token korrekt ist, wird der Benutzer verifiziert und die Verifizierung ist abgeschlossen.

3.4 Datenabfrage: Endpunkte für Serien und Episoden

Einer der ersten Schritte im Projekt-Verlauf war das Implementieren der der JSON-Endpunkte zur Abfrage von Serien und Episoden. Es soll für beide Entitäten eine Liste mit Einträgen sowie einzelne Einträge abgefragt werden können.

Dazu wurden zunächst die zwei *Services* “shows” und “episodes” erstellt. Wie oben (Kapitel 3.2, Seite 4) beschrieben, sind dies Verzeichnisse, welche möglichst isolierten Code für einen Bereich der Anwendung beinhalten. Für beide *Services* wurde eine `model.js` erstellt, welche Informationen zur Datenbank-Repräsentation der Daten beschreibt, sowie eine `index.js`, welche die möglichen HTTP-Anfragen beschreibt und einen *express Router* exportiert.

Der initiale Inhalt einer solchen `index.js` sieht so aus:

[8] `server/services/auth/password_helper.js`
[9] https://sendgrid.com/

```
var express = require('express');
var app = express.Router();
module.exports = app;
```

3.4.1 Abfragen vieler Einträge

Unter dem relativen Pfad `/` soll eine Liste von Einträgen abgefragt werden können. Eine triviale Variante eines solchen Endpunkts könnte wie folgt geschrieben werden (anschließend an den oben gezeigten Code der `index.js`):

```
var Show = require('./model.js');

app.get('/', function (request, response) {
  Show.query()
  .then(function (shows) {
    response.send(shows);
  })
  .catch(function (error) {
    response.status(500).send({error: error});
  });
});
```

3.4.2 Fehlerfälle abstrakt behandeln

Im vorigen Code-Beispiel wird ein möglicher Fehlerfall bei der Abfrage von Serien dadurch behandelt, dass eine Antwort mit HTTP-Status `500` und der JSON-Darstellung des erhaltenen Fehlers gesendet wird. Dieser Fall ist sehr allgemein, muss aber standardmäßig in jedem Endpunkt behandelt werden.

Um doppelten Code zu vermeiden, wurde die Funktion `wrapRoute` geschrieben (vgl. `server/helpers/wrap_route.js`). Diese wird statt einem direkten Callback beim Erstellen des Endpunkts verwendet und erwartet als einzigen Parameter eine Funktion, die ein Promise zurückgibt. Je nach Wert des aufgelösten Promises wird die entsprechende Antwort zurückgesendet.

Der benötigte Code für den trivialen Endpunkt von oben reduziert sich damit drastisch:

```
var wrapRoute = require('../../helpers/wrap_route');
var Show = require('./model.js');

app.get('/', wrapRoute(function (request) {
  return Show.query();
}));
```

3.4.3 HTTP-Antworten bei Fehlerfällen

Es sollte nicht auf jeden Fehler mit HTTP-Status `500` geantwortet werden, da dieser für "Internal Server Error" steht und daher für Fehler steht, deren Ursache nicht genauer durch einen HTTP-Status repräsentiert werden kann. Häufige Fehlerfälle, die zusätzlich beachtet werden sollen, und ihre korrekten Status-Codes, sind [1; 37]:

- `401`: Keine Zugriffsberechtigung.
- `404`: Angefragte Ressource konnte nicht gefunden werden.
- `409`: Daten-Konflikt (beim Erstellen eines neuen Eintrags).
- `422`: Daten konnten nicht verarbeitet werden (z.B. weil eine Validierung gescheitert ist).

3.4.4 Filterung und Sortierung

Standardmäßig werden alle Einträge ausgegeben (später wird die Anzahl limitiert und Seiten-weises Abfragen eingeführt). Diese Liste kann je nach Entität mit bestimmten Filtern versehen werden, welche als Teil der URL in Form von Query-Parametern übertragen werden.

Für Serien ist beispielsweise die Abfrage `/?show_ids=4,8,15,16,23,42` möglich, wodurch nur Einträge zurückgegeben werden, deren ID angegeben wurde. Da das Auslesen von Query-Paramtern und Ergänzen der Datenbank-Abfrage um die korrekten Bedingungen für viele Entitäten sehr ähnlich ist, wurden in `server/helpers/query_params.js` einige Hilfsmethoden dazu geschrieben.

3.4.5 Abfragen eines einzelnen Eintrags

Ähnlich wie die Abfrage nach einer Liste von Einträgen mit bestimmten IDs gestaltet sich auch die Abfrage eines einzelnen Eintrags. Jeder Eintrag ist unter der URL `/:id` verfügbar (wobei `:id` durch die ID des Eintrags ersetzt wird).

Zusätzlich zu den Feldern des Eintrags ist es bei bestimmten Entitäten auch möglich, verwandte Daten anderer Entitäten mit abzufragen, um z.B. eine Serie und die Liste der IDs aller dazugehöriger Episoden abzufragen. Diese Daten werden als Teil eines speziellen `link`-Attributs in der JSON-Antwort übertragen (vgl. Referenz 38).

3.4.6 Zukünftige Episoden als Kalender-Feed

Bisher liefert die Applikation auf jede Anfrage JSON-Daten zurück. Dies ist praktisch, wenn die Daten von einer Anwendung gelesen werden, die an EpisodeFever angepasst wurde (z.B. ein HTML5-basiertes Frontend für EpisodeFever). Auf Grund der beim Import ermittelten (Kapitel 3.6.3, Seite 13) genauen Zeit-Informationen ist aber gerade bei den Episoden-Daten auch noch ein anderes Format sinnvoll: Kalender-Feeds (*ics*- bzw. *iCal*-Format).

Ein Benutzer kann einen solchen Feed in sein Kalender-Programm (z.B. *Google Calendar* oder *Kalender* von OS X) und erhält so die Termine zukünftiger Episoden der von ihm betrachteten Serien.

Das *ics*-Format ist ein Text-Format, welches *Events* (Termine) in Zeilenblöcken anhand von bestimmten Attributen beschreibt. Da es ein sehr verbreitetes Format ist (**s. Standard**), gibt es eine Reihe von Libraries, welche dieses aus JSON-Daten erzeugen können. Eine, welche sehr umfangreich ist, aktiv entwickelt wird und außerdem explizit Unterstützung für Zeitzonen bieten, ist *cozy-ical* [39].

3.5 Testen von REST-Anfragen

Um sicherzustellen, dass die für einen Service vorgesehenen Funktionen korrekt implemtiert wurden und um zu vermeiden, dass in Zukunft Änderungen gemacht werden, welche die erwarteten Funktionsweisen brechen, werden zu jedem Service automatisierte Tests geschrieben. Dazu werden die oben beschriebenen (Kapitel 2.2.2, Seite 3) Module *mocha*, *chai* und *supertest* verwendet.

Durch *supertest* ist es möglich, einen auf *Express.js* basierten Server in einem Test-Kontext zu starten, ohne ihn auf einem bestimmten Port zu starten. An diesen Server werden dann Anfragen gestellt, und die Antworten auf erwartet Werte und Strukturen überprüft. Ein einfacher Test sieht so aus:

```
var request = require('supertest');
var app = require('express')();
```

```
app.use('/', require('./index'));

// `describe` beginnt Test-Suite
describe("API index", function () {
  var agent = request.agent(app);

  // `it` beschreibt einen konkreten Test-Fall
  it("returns a JSON response", function () {
    return agent.get('/')
    .expect(200) // Test auf HTTP-Status (Supertest)
    .set('Accept', 'application/json')
    .expect('Content-Type', /json/)
    .exec() // Konvertiere Anfrage-Objekt zu Promise
    .then(function (response) {
      // Teste Antwort auf korrekt Struktur
      expect(response.body).to.be.an('object');
    });
  });
});
```

3.5.1 Test-Daten

Eine Herausforderung beim Schreiben von Tests ist es, dynamisch Test-Daten in die Datenbank einfügen zu können. Um beispielsweise testen zu können, ob ein Entpunkt die Liste aller Episoden ausgibt, welche nach einem bestimmten Datum ausgestrahlt werden, müssen zunächst Episoden-Daten mit verschiedenen Ausstrahlungsdaten eingefügt werden.

Um dies zu vereinfachen, wurde jedem *bookshelf*-Model eine statische `fake`-Methode hinzugefügt. Diese generiert standardmäßig einen Datensatz mit zufällen Daten (mit Hilfe von *faker.js* [40]), es können jedoch einzelne Felder beliebig überschrieben werden.

Da `fake()` ein Promise mit dem zu speichernden Datensatz zurückliefert, kann das Erstellen von Test-Daten direkt innerhalb eines Tests ausgeführt werden. Sollen die Daten in allen Tests einer Test-Suite verliegen, können sie auch in einem `before`-Block eingefügt werden (auf der selben Ebene wie die mit `it` registrierten konkreten Tests). In solchen `before`-Blöcken werden zu Beginn einer Tests-Suite oft auch die schon in der Datenbank vorhandenen Einträge gelöscht, um konsistent testen zu können.

Im folgenden Beispiel wird eine Serie eingefügt und anschließend ein Test für `/shows` ausgeführt:

```
var request = require('supertest');
var Show = require('./model');
var app = require('express')();
app.use('/', require('./index'));
var F = require('../../helpers/faking_helpers');

describe("Shows API", function () {
  var agent = request.agent(app);

  before(function () {
    return F.dropAllTheData()
    .then(function () {
      return Show.fake();
    })
  });
```

```
  it("returns a list of shows", function () {
    return agent.get('/')
    .expect(200)
    .exec()
    .then(function (res) {
      expect(res.body).to.be.an('object');
      expect(res.body.shows).to.be.an('array');
      expect(res.body.shows).to.have.length(1);
    });
  });
});
```

3.6 Import von Daten

Informationen zu TV-Serien und deren Episoden sind von verschiedenen Diensten verfügbar. Ein wichtiger Teil des Episode-Fever-Projektes ist es, diese Informationen abzufragen und in der lokalen Datenbank so zu speichern, dass die restliche Anwendung effizient darauf zugreifen kann.

Eine der bekanntesten Plattformen für diese Daten ist TheTVDB [41], auf welcher Freiwillige Metadaten, Beschreibungen und sogar Grafiken zu Serien in verschiedenen Sprachen eintragen können, welche dann unter einer freien Lizenz[10] zur Verfügung stehen.

Außerdem bietet TheTVDB eine XML-API, mit der es möglich ist, nach Serien zu suchen und die gesamten Daten zu einer Serie (inkl. Daten zu Episoden) abzufragen.

3.6.1 XML-Daten abfragen und verarbeiten

Um mit der XML-API über HTTP zu kommunizieren, wurde *superagent* [29] eingesetzt. Basierend auf den von Node mitgelieferten HTTP-Client-Funktionen[11] bietet es eine übersichtliche Schnittstelle zum Erstellen von komplexen HTTP-Requests. Um den Umgang mit asynchronen Funktionen zu vereinfachen, wurde der Prototyp von *superagent* um die Methode `.exec()` erweitert, welche ein *Promise* zurückgibt (siehe verwendete Technologien (Kapitel 2, Seite 2)).

Um das Arbeiten mit den API-Antworten im XML-Format zu vereinfachen, wurde das Modul *xml2js* [43] ausgewählt. Dieses basiert auf dem Streaming-Parser *sax-js* [44] und konvertiert XML-Strukturen in JavaScript-Objekte. Hierbei werden einige Optionen angeboten, welche das resultierende Objekt stark vereinfachen können, u.a. um Knoten mit nur einem Kind als direkten Datensatz (und nicht als Array) auszugeben. Da die APIs XML-Dokumente zurückgeben, welche sehr wenig Gebrauch von verschachtelten Knoten oder Attributen machen, kann *xml2js* Objekte mit flacher Struktur erzeugen, welche einfach zu verwenden sind.

Die gesamte Konfiguration zur Abfrage und Verarbeitung der API-Anfragen kann in der Datei `import/apis/xml_api_helper.js` gefunden werden.

3.6.2 Verwenden der XML-API von TheTVDB

Jede Anfrage zu der TheTVDB-API muss mit einem API-Key versehen werden (als Teil der URL). Ein solcher Schlüssel kann über ein Formular[12] beantragt werden.

[10] "Creative Commons Attribution 3.0 United States", vgl. Referenz 42.

[11] *superagent* kann auch im Browser-Kontext verwendet werden und bietet so eine einheitliche Schnittstelle auf beiden Plattformen.

[12] http://thetvdb.com/?tab=apiregister

Um die Daten einer TV-Serie auszulesen, muss zunächst die TVDB-eigene ID dieser Serie gefunden werden. Mit dieser kann dann die URL zu dem gesamten Datensatz der Serie generiert werden. Vergleiche hierzu die Abbildung zu API-Abfragen (Abbildung 4, Seite 13).

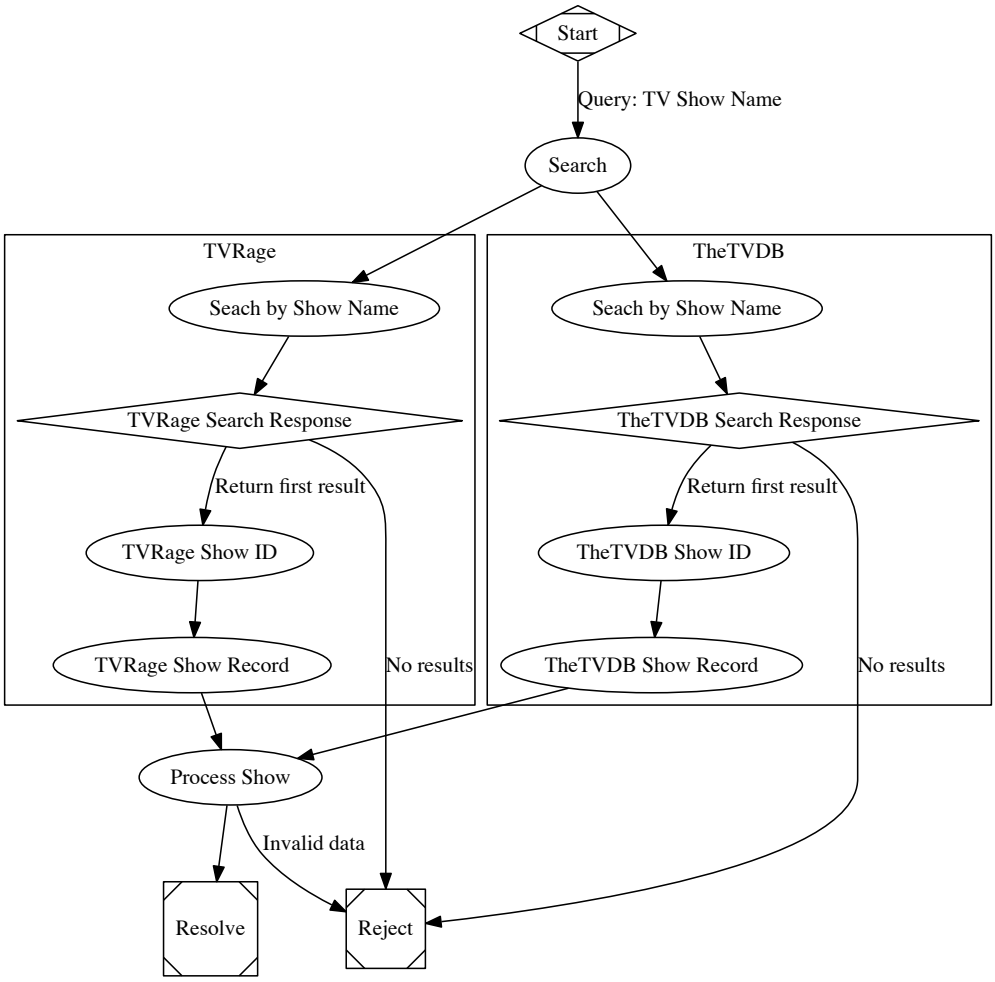

Abbildung 4: Abfragen und Verarbeiten der Daten von TheTVDB und TVRage

Obwohl die Daten der API nun als XML (bzw. JavaScript Objekt) vorliegen, müssen noch kleine Transformationen durchgeführt werden, um sie verwenden zu können. So beinhalten einige Felder zwar Zeichenketten, inhaltlich handelt es sich jedoch um Listen von Werten. Das “Genre”-Feld einer Serie kann beispielsweise den Wert `"|Action|Adventure|Comedy|Drama|"` haben.

3.6.3 Datum und Zeit einer Episode auslesen

Des Weiteren ist das Verarbeiten von Datumsformaten notwendig. Ziel ist es, jeder Episode sowohl Datum als auch Uhrzeit zuzuordnen, wann sie (zuerst) ausgestrahlt wurde. Dies ermöglicht es insbesondere, zukünftige Episoden abzufragen, z.B. um einen Kalender zu implementieren und Benutzer im Voraus zu benachrichtigen.

Die API von TheTVDB beinhaltet hierzu nur ungenaue Daten [45]. Jede Episode besitzt zwar ein `FirstAired`-Feld, dieses beinhaltet jedoch nur das Datum (z.B. `"2009-02-03"`). Die Uhrzeit

der Ausstrahlung ist jedoch im Datensatz der Serie verfügbar (`Airs_Time`), vermutlich unter der Annahme, dass diese typischerweise nicht variiert. Das Format der Uhrzeit ist jedoch nicht eindeutig, da Werte wie `"8:00 PM"` verwendet werden, und keine Zeitzone angegeben wird. Es scheint immer die Zeitzone des TV-Senders, auf dem die Serie initial ausgestrahlt wird, verwendet zu werden. Da aber keine Zuordnung von Sendern zu Zeitzonen verfügbar ist, kann hierdurch auch keine genaue Zeitangabe berechnet werden.

Diese Lücke in der TheTVDB-API bedeutet für EpisodeFever, dass entweder nur unvollständige Daten verfügbar sind, oder zusätzlich Daten aus einer zweiten Quelle geladen werden müssen.

3.6.4 Kombination von TheTVDB und TVRage

Eine weitere Quelle für Daten zu TV-Serien ist TVRage [46]. Diese Webseite bietet ähnliche Daten wie TheTVDB und wird (zum Teil) ebenfalls von Freiwilligen gepflegt. TVRage bietet ebenfalls eine XML-API, über die Metadaten zu Serien und Episoden abgefragt werden können. Die Endpunkte von TVRage sind ähnlich zu denen von TheTVDB, verwenden aber unterschiedliche URL-Strukturen und Feld-Bezeichnungen.

Die Daten von TVRage beinhalten vor allem aber neben dem Feld `airtime` (welches zudem das 24-Stunden-Format für Uhrzeiten verwendet) auch noch das Feld `timezone`, welches Zeitzonen in Repräsentationen wie `"GMT-5 +DST"` beinhaltet.

An TVRage werden die identischen Anfragen gestellt wie an TheTVDB (mit Angepassten URLs und Parametern), sodass nun pro Serie zwei Datensätze zur Verfügung stehen[13]. Da die Daten von TVRage keine Beschreibungen beinhalten[14] und die weiteren Informationen identisch sein sollten, werden nur die Informationen zum Ausstrahlungs-Zeitpunkt von TVRage übernommen.

3.6.5 Aktualisieren von Serien

Beim initialen Hinzufügen von Serien werden diese anhand ihres Namens gesucht. Das Aktualisieren von Serien kann diesen Schritt überspringen, da mit jedem Serien-Eintrag in der Datenbank ebenfalls die IDs von TheTVDB und TVRage gespeichert werden.

Dazu werden zugehörige Episoden anhand von Staffel und Nummer eideutig identifiziert und können so auch aktualisiert werden.

Im derzeitigen Stadium werden alle Serien aktualisiert. Eine mögliche Verbesserung wäre, nur dann neue Daten abzufragen, wenn Aktualisierungen am wahrscheinlichsten oder am relevantesten sind. Das könnte für laufende Serien beispielsweise am Tag vor dem Ausstrahlen neuer Episoden sein. Außerdem könnte für beendete Serien ein sehr viel geringerer Rhythmus gewählt werden.

3.6.6 Import automatisiert testen

Wie die restliche Anwendung auch, soll der Import von Serien testbar sein, um Regressionen zu vermeiden. Da der Import jedoch von externen Diensten abhängt, gibt es hier einige zusätzliche Problemquellen. Die Dienste könnten offline sein, die Struktur ihre Antworten ändern oder auch abgeschaltet werden.

[13] Tatsächlich wird bei der initialen Abfrage der Daten der von TheTVDB zurückgegebene Name der Serie für die Suche mit der TVRage-API verwendet. So wird mit großer Sicherheit die selbe Serie von beiden APIs geliefert. Durch Vergleiche zusätzlicher Daten beim Import kann dies zusätzlich geprüft werden.

[14] Die Beschreibungen von TVRage können nur mit API-Keys mit speziellen Berechtigungen geladen werden.

HTTP-Anfragen aufzeichnen Es erschien sinnvoll, die HTTP-Anfragen an externe Dienste zwischen zu speichern und nur zu bestimmten Zeitpunkten zu aktualisieren[15].

Das Modul *replay* [47] erlaubt es, das in *node* integriert HTTP-Modul so zu überschreiben, dass es in verschiedenen Modi operieren kann, vor allem `record` and `replay`. (Ursprünglich konnte *replay* URLs nicht anhand ihrer Query-Parameter unterscheiden, dies wurde als Patch hinzugefügt, wie in "Beiträge zu Open Source" (Kapitel 5.1, Seite 24) erwähnt.)

So können Tests ganz normal auf externe HTTP-Server zugreifen und *replay* schreibt im `record`-Modus alle erhaltenen Antworten in ein vorher definiertes Verzeichnis. Sobald die Tests erfolgreich ausgeführt wurden, kann auch `replay` umgestellt werden. Von nun an werden die aufgezeichneten Antworten verwendet.

Dieses Vorgehen hat auch den Vorteil, dass Tests sehr viel schneller laufen und unabhängig von einer Internetverbindung sind.

replay zeichnet außerdem sehr viele Header auf und speichert den HTTP-Body komprimiert, wenn die ursprüngliche Antwort komprimiert war. Um die Lesbarkeit der Test-Daten zu verbessern, wurden die Daten manuell entpackt, überflüssige Header entfernt und sprechend benannt.

Tests für zwei APIs Da zwei kleine Module verwendet werden, um den Zugriff auf die APIs zu abstrahieren, war es möglich, jedem Modul ein identische Interface zu geben. Die Daten-Strukturen in den Antworten der APIs werden jedoch nicht vereinheitlicht.

Alle API-Tests werden unabhängig von der verwendeten API geschrieben und sind Teil einer `testApi`-Funktion. Diese wird mit Test-Daten und einer API-Schnittstelle aufgerufen. So können mit jeder API die identischen Tests aufgerufen werden und es ist sichergestellt, dass die Daten auf die selbe Art und Weise verarbeitet werden können.

3.7 Bewertungen abgeben

Kern-Funktion von EpisodeFever ist das Bewerten von Episoden. Ein Bewertungs-Datensatz beinhaltet zu der Bewertung-Zahl selbst[16] Referenzen zu einem Benutzer, einer Serie und einer Episode. Jeder Benutzer kann genau eine Bewertung zu einer Episode abgeben, diese kann jedoch bearbeitet werden.

Aus diesen Informationen lassen sich viele weitere Daten berechnen, beispielsweise die Durchschnittsbewertungen für Episoden, Staffeln, Serien oder Benutzer. Außerdem lassen sich darüber die Serien bestimmen, die ein Benutzer schaut, ohne dass diese explizit angegeben werden müssen[17].

3.7.1 Motivation: Auflisten von zukünftig relevanten Episoden

Als Benutzer möchte ich es möglichst einfach haben, die Episoden zu finden, die ich bewerten möchte. Meist sind das genau die Episoden, die auf die zuvor von mir bewerteten folgen.

Daher ist eine sehr wichtige Funktion für ein Interface zu EpisodeFever das Auflisten der von einem Benutzer zuvor bewerteten Episoden sowie der darauf folgenden. Wurde zu einer Serie die 14. Episode der dritten Staffel bewerten, sollen z.b. die Episoden 14 bis 19 angezeigt werden. (Es ist zusätzlich zu beachten, dass nur Episoden bewerten werden können, die bereits ausgestrahlt wurden.)

[15] Es könnte beispielsweise alle zwei Wochen ein spezieller Test laufen, welcher die Antworten der APIs vergleicht und einen Entwickler benachrichtigt, wenn es Änderungen gibt. Dies ist unter Umständen auch zuverlässiger als z.B. ein News-Feed zu API-Änderungen eines Dienstes.

[16] Aktuell sind die Bewertungen "Gut", "Mittel" und "Schlecht" (als Zahlen 3 bis 1 gepeichert) möglich. Diese Skala kann in Zukunft jedoch einfach geändert werden.

[17] Es wurde überlegt, die Relation zwischen Benutzer und Serien, die dieser schaut, explizit zu speichern. Die Vorteile hiervon werden im Kapitel "Ausbaustufen" (Kapitel 4.2.4, Seite 23) beschrieben.

3.7.2 Zuletzt bewertete Serien abfragen

Um an die zuvor beschriebenen Daten zu gelagen, wird zunächst eine Liste mit den zuletzt abgegebenen Bewertungen benötigt, gruppiert nach Serie. Eine passende SQL-Abfrage lässt sich wie folgt formulieren:

```
SELECT *
FROM (
  SELECT
    ROW_NUMBER() OVER (
      PARTITION BY
        show_id
      ORDER BY
        updated_at DESC NULLS LAST,
        id DESC NULLS LAST
    ) AS row_number,
    t.*
  FROM
    votes AS t
  WHERE
    user_id = ?
) AS latest_votes
WHERE "latest_votes"."row_number" <= ?
LIMIT ?
```

Die Parameter für diese Abfrage (im Code als `?` gekennzeichnet) sind die Benutzer-ID und die Anzahl der abzufragenden Bewertungen.

Hat man durch diese Abfrage nun eine Liste mit Bewertungen, lassen sich über die darin referenzierten Episoden- und Serien-IDs einfach die weiteren Episoden abfragen, indem man Episoden nach der Serien filtert, nach Staffel und Nummer sortiert und dann nur die ausgibt, welche auf die zuletzt bewertete folgen.

3.8 Suche

Eine Suchfunktionalität ein wichtiger Bestandteil jeder Anwendung, die große Mengen an Daten für die Benutzer zur Verfügung stellt. Sie soll zum einen die Suche nach Information erleichtern und beschleunigen.

Die wesentlichen Features, die diese bieten sollte sind unter anderem:

- Wörter auf ihren Wortstamm reduzieren
- Stopwörter entfernen (ein, das, im, mit, etc.)
- Gewichtung von Suchergebnissen
- Rechtschreibkorrektur

Aufgrund der von *PostgreSQL* zur Verfügung gestellten Funktionen für die Volltextsuche hat sich unsere Entscheidung, eben dieses Datenbanksystem zu benutzen, als richtig erwiesen, da es all die oben genannten Features, aber auch andere nützliche Funktionen, wie z.B. den Support von Fremdsprachen bietet.

Ein alternatives Systeme für eine dedizierte Volltextsuche ist *ElasticSearch* [48]. Dies ist ein auf Apache Lucence [49] basierendes NoSQL-System, welches darauf ausgelegt ist, große Mengen an Textdaten zu indexieren und effizient durchsuchbar zu machen.

3.8.1 Anforderungen an die Suche

Die Überlegung die wir uns dazu gemacht haben waren folgende: Es gibt 3 Arten von Benutzern:

1. Der Benutzer weiß genau was er sucht. Er will etwas in die Suche eingeben, das richtige Ergebnis zurückbekommen und schnell auf die Daten zugreifen.

   Beispiel: Der Benutzer will eine Bewertung zu seiner Lieblingsserie abgeben und will möglichst schnell auf die entsprechende Seite geleitet werden. Er gibt den Titel der Serie in die Suchleiste ein und gelangt zur Serie.

2. Der Benutzer hat wenig Informationen und möchte, dass ihm dementsprechend Vorschläge gemacht werden, die am ehesten seiner Informationen entsprechen.

   Beispiel: Der Benutzer hat 5 Minuten einer ihm unbekannten Serie/Episode geschaut und weiß deshalb nur die Namen von 1-2 Charakteren oder nur einen Teil der Hintergründe, aber nicht den Titel der Serie/Episode. Er tippt was er weiß ein und bekommt nach Relevanz sortierte Vorschläge.

3. Der Benutzer hat keine Informationen und sucht nichts spezifisches. Er wird die Suchfunktion deshalb nicht benutzen und sich höchstens umschauen, welche Serien es gibt, bzw. welche Bewertungen diese haben.

Da der dritte Benutzertyp für die Suchfunktion keine Rolle spielt sind nur die Anforderungen der ersten beiden Benutzer wichtig.

Die Anforderungen an die Suchfunktion sind somit zusammengefasst: Schnelligkeit, richtige Ergebnisse bei präzisen Anfragen und nach Relevanz sortierte Vorschläge bei unpräzisen Anfragen.

3.8.2 Die Queries

Im wesentlichen besteht die Suche aus 2 SQL Anfragen, der Rechtschreibkorrektur und der Suche nach der Episode oder der Serie, wobei die Rechtschreibkorrektur vor der Suche nach den Serien/Episoden stattfindet. Weshalb dies so umgesetzt wurde, wird im Kapitel “Idee und Umsetzung” (Kapitel 3.8.5, Seite 19) erläutert.

3.8.3 Suche nach Episoden

```
SELECT
  shows.name AS show,
  episodes.name AS episode,
  episodes.season, episodes.number, episodes.id, episodes.show_id
FROM episodes JOIN shows ON shows.id = episodes.show_id
WHERE
    to_tsvector('english_nostop', coalesce(episodes.name, '')) ||
    to_tsvector('english', coalesce(episodes.description, '')) @@
    to_tsquery(input)
ORDER BY
    ts_rank((
      setweight(to_tsvector('english_nostop',coalesce(episodes.name,'')),'A') ||
      setweight(to_tsvector('english',coalesce(episodes.description,'')),'B')),
      to_tsquery('english_nostop', input)) DESC
```

Wie man sieht wird der Name, die Season, die Episodennummer, sowie der Name und die ID der Serie zurückgegeben. Dafür werden die Tabellen `episodes` und `shows` gejoint. Der interessante Teil

ist der `WHERE` und `ORDER BY`-Teil, denn dabei werden die Features der *PostgreSQL*-Volltextsuche in Anspruch genommen:

`to_tsvector([ config regconfig , ] document text)` reduziert Text zu einem `tsvector`, der die Lexeme und deren Position innerhalb des Textes enthält. Ein Lexem ist die "Einheit des Wortschatzes, die die begriffliche Bedeutung trägt" [50].

`to_tsquery([ config regconfig , ] query text)` normalisiert Wörter und wird zum durchsuchen des ts_vector benutzt.

`setweight(tsvector, "char")` Gibt den Lexemen des tsvectors Gewichtungen. "Char" ist dabei A,B,C oder D, mit A höchstes und D niedrigstes Gewicht.

`ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ])` gibt dem Query einen Rang.

Der `@@` Operator überprüft, ob `tsvector` und `tsquery` übereinstimmen und liefert `true` oder `false`.

Beispiel	Ergebnis
`to_tsvector('english', 'The Fat Rats')`	`'fat':2 'rat':3`
`to_tsquery('english', 'The & Fat & Rats')`	`'fat' & 'rat'`
`setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')`	`'cat':3A 'fat':2A,4A 'rat':5A`
`ts_rank(textsearch, query)`	`0.818`
`to_tsvector('fat cats') @@ to_tsquery('cat')`	`t`

(Beispiele aus PostgreSQL Dokumentation zu *Text Search Functions*[18].)

Die Episoden-Namen und -Beschreibungen werden zu `tsvector` umgewandelt, gewichtet und mit `tsquery` untersucht. Was besonders ins Auge fällt, ist dass für die Beschreibungen die Sprache 'english' und für die Titel 'english_nostop' verwendet wurde. Die Sprache 'english_nostop' wurde von uns angelegt und der Unterschied zu 'english' besteht darin, dass Stop Wörter nicht entfernt werden. Dies ist besonders wichtig, da Titel durchaus Stop Wörter enthalten können.

Beispiel: Wenn man die Serie "Doctor Who" sucht und in die Suche "who" eingibt würde, die Suche die Serie nicht finden, weil "who" ein Stop Wort ist.

Die Ergebnisse der Suche werden abschließend nach Relevanz absteigend sortiert, sodass das Ergebnis mit dem höchsten Rang als erstes ausgegeben wird.

Query zur Rechtschreibkorrektur

```
SELECT word
FROM unique_lexeme
WHERE word % input AND similarity(word, input) >= 0.5
ORDER BY similarity(word,input) DESC
LIMIT 1;
```

Für die Rechtschreibkorrektur haben wir eine Materialized View `unique_lexeme` angelegt, die alle Lexeme aus den Serien und Episoden Tabellen enthält. Dafür haben wir die `ts_stat` Funktion von *PostgreSQL* benutzt, welche Statistiken über jedes einzelne Lexem aus den `tsvector`-Daten zurückgibt.

Zusätzlich haben wir die *PostgreSQL* Extension `pg_trgm` (Trigram) verwendet: Diese stellt uns einige wichtige Funktionen und Operationen zur Verfügung um Wörter auf Ähnlichkeit zu untersuchen. Dazu wird ein String in die sogenannten Trigramme zerlegt, diese sind die aufeinander-

[18] http://www.postgresql.org/docs/8.3/static/functions-textsearch.html

folge von 3 Buchstaben aus dem String. Die Trigramme von “Trigram” wären also beispielsweise `[Tri],[rig],[igr],[gra],[ram]`, wobei Leerzeichen als Unterstriche dargestellt werden.

Der %-Operator und die `similarity`-Funktion Der `%`-Operator untersucht, ob die Ähnlichkeit von 2 Argumenten über einem bestimmten Wert liegt (Default: `0.3`) und gibt, falls dem so ist true zurück. Similarity gibt eine Zahl zurück, wie Ähnlich 2 Argumente sind, wobei `0` keine Ähnlichkeit entspricht und `1`, dass sie identisch sind.

Wir benutzen beide, da wir zum einen, einen Index benutzen und dieser mit dem `%`-Operator effizienter genutzt wird und zum anderen benutzen wir
Similarity als Post-Filter bei dem wir die Ergebnisse vom `%`-Operator nochmal filtern.

3.8.4 Such-Indizes

Um die Laufzeit der SQL-Abfragen drastisch zu verringern, stellt *PostgreSQL* sogenannte Indizes zur Verfügung. Diese sind sozusagen eine Verknüpfung oder eine Art Lesezeichen, auf die bei der Indexerstellung definierte Spalte einer Tabelle.

Dabei gibt es 2 Arten von Indizes: Den GIN und GiST Index.

> “As a rule of thumb, GIN indexes are best for static data because lookups are faster. For dynamic data, GiST indexes are faster to update. Specifically, GiST indexes are very good for dynamic data and fast if the number of unique words (lexemes) is under 100,000, while GIN indexes will handle 100,000+ lexemes better but are slower to update.”
>
> – PostgreSQL Dokumentation zu *Full Text Search*[19]

Da die Serien- und Episoden-Daten zum größten Teil statisch sind, eine niedrige Laufzeit der Queries für uns wichtig ist und die GiST Indizes auch falsche Ergebnisse liefern können, haben wir entschieden, die GIN-Indizes sowohl für die Suche nach Serien/Episoden, als auch für die Rechtschreibkorrektur zu verwenden.

3.8.5 Idee und Umsetzung

In diesem Abschnitt behandeln wir unsere Idee für die Suche und deren Umsetzung.

Die Überlegung war es, dass wir im Frontend eine Library benutzen, welche es uns ermöglicht, die Suchanfragen periodisch bzw. live, also während der Benutzer noch eintippt, zu senden.

Wir haben uns mehrere Librarys angeschaut, unter anderem *typeahead.js* [51] und *rx.js* [52], haben uns aber letztendlich für *kefir.js* [53] entschieden, da dieses sehr kompakt und besonders performant ist. Wie *rx.js* bietet *kefir.js* eine Implementierung von Event-Streams und ermöglicht durch zahlreiche Hilfsmethoden effiziente, funktional reaktive Programmierung [54].

Im wesentlichen sieht das Skript für das Autocompletion-Feature wie folgt aus:

```
var queries = Kefir.fromEvent(inputField, 'keyup')
.debounce(250)
.map(function (ev) { return ev.target.value; })
.filter(function (val) { return val.length > 0; })
.skipDuplicates()
.onValue(function (ev) { /* trigger http request for search */ });
```

[19] http://www.postgresql.org/docs/9.4/static/textsearch-indexes.html

Tippt der Benutzer tippt etwas ein, wird die Suchanfrage als `GET`-Request gesendet und das eingegebene Wort wird mit den Lexemen aus der Materialized View `unique_lexeme` verglichen. Dabei wird das Lexem mit der größten Ähnlichkeit ausgewählt und dieses an die Suche nach der Serie/Episode als Parameter übergeben. Anschließend werden die Ergebnisse nach Relevanz sortiert zurückgegeben.

Auf diese Weise haben wir somit alle unsere Anforderungen an die Suche erfüllt: Schnelligkeit, richtige Ergebnisse bei präzisen Anfragen und nach Relevanz sortierte Vorschläge bei unpräzisen Anfragen.

3.8.6 Beispiel für eine Suche

Wir wollen "Mike", den Namen des Protagonisten aus der Serie "Suits" in die Suche eingeben und ebendiese Serie zurückbekommen.

Wir tippen also "m" ein, der `GET`-Request wird abgeschickt, die Fehlerkorrektur bestimmt das Lexem das die höchste Bewertung für "m" hat z.B. "mia", übergibt es an die Suche und liefert alle relevanten Ergebnisse.

Während die Suche durchgeführt wird, tippen wir aber weiter, sodass wir "mik" in der Suchleiste stehen haben.

Der `GET`-Request wird erneut abgeschickt, die Fehlerkorrektur bestimmt, dass das Lexem mit der höchsten Bewertung für "mik", "mike" ist und gibt es an die Suche weiter. Diese gibt uns wieder alle relevanten Ergebnisse zurück. In diesem Beispiel gehen wir davon aus, dass eine andere Serie relevanter ist. Deshalb präzisieren wir unsere Suche und tippen ein "mike suit". `GET`-Request wird abgeschickt, da "mike" und "suit" 2 Wörter sind die durch ein Leerzeichen getrennt sind, aber wir nur 1 Wort an die Fehlerkorrektur übergeben können, wird der Input getrennt und ein Array erzeugt. Zuvor wird die Eingabe jedoch bei jeder Suchanfrage normalisiert. Dafür werden unter anderem Leer- und Sonderzeichen durch '+' ersetzt, sodass immer eine gültige Suchanfrage vorliegt.

```
function normalize(word) {
  var word = word.toLowerCase()
  .replace("%20","+")
  .replace(/([+])+/g,"+")
  .match(/([A-Za-z0-9+])+/g);

  if ((word === undefined) || (word === '') || (word === null)) {
    throw new E.BadRequestError("Please enter a valid search query!");
  } else {
    return word.join("+");
  }
}
```

Dann wird die normalisierte Eingabe getrennt und ein Array erzeugt. Das Array wird dann an die Rechtschreibkorrektur übergeben, diese gibt für "suit", "suits" zurück.

"mike" und "suits" werden dann zusammengefügt und durch das Zeichen für ein logisches Und getrennt:

```
Promise.all(
  input.split('+').map(function (word) { return spellcheck(word); })
)
.then(function (words) {
  var query;
  input = words
```

```
  .filter(function (word) { return word && word[0]; })
  .map(function (word) { return word[0].word; })
  .join('&');
  // ...
});
```

Als Input wird nun `"mike&suits"` an die Suche übergeben und wir bekommen als relevantestes Ergebnis die Serie “Suits”.

4 Fazit

Die zu Beginn des Projektes gesetzten Ziele – Import von Serien- und Episoden-Daten, autorisierte Benutzer können Episoden bewerten – wurden erreicht. Darüber hinaus wurden auch noch weitere Funktionen implementiert, u.a. eine Suche und eine Übersicht zukünftig für den Benutzer relevante Daten.

Zudem konnten wir uns im Laufe des Projektes auch mit vielen verschiedenen Technologien vertraut machen. Das Vorgehen nach Test-driven Development und dem Entwurf der JSON-API nach REST-Prinzipen [1] und den Best Practices, die in Referenz 38 beschrieben wurden, konnten uns helfen, strukturierten Code zu schreiben.

4.1 Rückblick

Nach Abschluss des Projekts gibt es einige Punkte, die wir im Nachhinein anders machen würden. Im Allgemeinen sind dies jedoch eher kleiner Punkte; keine der im Verlauf der Entwicklung aufgetretenen Probleme waren ein unüberwindbare Hindernisse.

Die in “Technologie” (Kapitel 2, Seite 2) beschriebenen Entscheidungen haben sich bewährt. Sie haben es ermöglicht, alle gewollten Funktionen umzusetzen, sowie einige zusätzliche zu ermöglichen, z.B. den Kalender-Feed oder die Suche. Auf dieser Basis könnten auch viele Erweiterungen problemlos umgesetzt werden. Außerdem sind wir zuversichtlich, dass die Software in einer Produktivumgebung stabil und performant laufen wird.

4.1.1 Technologie-Entscheidungen

Zwei Entscheidungen zu Modulen, die wir einsetzen wollten, haben wir revidiert.

Zu Beginn war geplant, die Benutzer-Authentifizierung auf Basis von *Passport* [34] zu implementieren. Dieses Modul ist im Grunde jedoch nur eine Sammlung von Adaptern verschiedener Authentifizierungsmethoden, um Benutzer z.B. über deren vorhandene Facebook-, Twitter- und OpenID-Accounts zu registrieren. Das Modul für eine lokale Benutzerverwaltung, *passport-local* ist ein recht einfacher Adapter, welcher jedoch verlangt, dass die Repräsentation des Benutzers in der Datenbank (Auslesen und Ändern) selbst geschrieben wird. Da dies das mit Abstand komplexeste Unterfangen der Benutzer-Authentifizierung ist und *passport-local* sonst kaum Funktionen mitbringt, haben wir uns entschieden ganz darauf zu verzichten und den gesamten Workflow selbst zu implementieren. Hätten wir MongoDB verwendet, wäre es jedoch möglich gewesen, *passort-local-mongo* zu verwenden, welches Session-Handling und das Benutzer-Schema für diese Datenbank implementiert.

Ein weiteres Modul, welches wir letztendlich nicht verwendet haben, ist *node-tvdb* [55], eine Implementierung der TheTVDB-API. Dieses Modul ließ sich schnell durch die direkte Verwendung von *xml2js* [43] und *superagent* [29] ersetzen. Dies hat den zusätzlichen Vorteil, dass auf die selbe Weise auch die Schnittstelle zur TVRage-API geschrieben werden konnte.

4.1.2 Zu node.js und der Zukunft von JavaScript

Das ursprüngliche Vorhaben für das Projekt war, eine Applikation auf Basis von *node.js* zu entwickeln. Abschließend sind wir zufrieden mit dieser Wahl. *Node* ist zwar noch nicht als "1.0"-Version erschienen, da es aber von einigen großen Unternehmen bereits für wichtige Projekte verwendet wird, sind große Teile von *Node* dennoch als stabil und gut getestet einzustufen. Es werden eine Reihe aktueller (wie auch älterer Versionen) regelmäßig mit Sicherheits-Aktualisierungen versehen und viele APIs der Kern-Module sind als stabil markiert.

Dass *node* trotz seinem Fokus auf Asynchronität auf dem Ausführen von JavaScript basiert, bedeutet zur Zeit, dass viele Programmier-Techniken zur Abstraktion und Verbesserung der Ergonomie nur als Module verfügbar sind, die Sprache selbst diesen aber agnostisch gegenüber ist. Auch, wenn sich dies mit einer zukünftigen Version von *node* ändern sollte (z.B. durch die Unterstützung neuer Funktionen aus dem "ECMAScript 2015"-Standard [56]), wird es noch viele Module geben, die diese Techniken nicht verwenden, sowie andere, welche ähnliche, aber inkompatible oder obsolete Implementierungen einsetzen.

Ein Beispiel hierfür ist die von uns verwendete Bibliothek *bluebird* [20], um Promises abzubilden. Unsere Wahl fiel explizit auf diese Bibliothek, da sie verspricht, kompatibel zu den nativen Promises in ECMAScript 2015 zu sein. Dass andere von uns verwendeten Module wie *knex* und *bookshelf* ebenfalls *bluebird* als Abhängigkeit haben, war ein ziemlich Glücksgriff. Aus anderen Projekten war bekannt, dass es sonst nötig gewesen wäre, an vielen Stellen Promise-Instanzen der einen in Promise-Instanzen der anderen Bibliothek zu konvertieren, was weder für Performance noch für Entwickler-Effizient gut ist.

Sobald *node* in einer stabilen Version einen Großteil der Funktionen von ECMAScript 2015 (und nachfolgenden Revisionen) unterstützt, wird es möglich sein, einige andere Pattern zu verwenden, um Applikationen auf eine andere Weise zu implementieren. Konkret existiert beispielsweise mit *koa.js* [11] schon eine Alternative zu *express.js* [8], welche Middlewares auf Basis von Generatoren[20] implementiert.

Die Grenzen von JavaScript sind auch an anderen Stellen zu sehen. Gerade für größere Projekte kann es problematisch sein, Interfaces korrekt zu definieren und zuversichtlich verwenden zu können, da JavaScript als dynamische Sprache hier von sich aus kaum Sicherheiten bietet. Abhilfe schaffen hier Ansätze wie statische Code-Überprüfung (z.B. mit *ESLint* [57]) oder auf JavaScript aufsetzende Type Systeme wie *TypeScript* [58] oder *Flow* [59].

Bei der Entwicklung EpisodeFever konnten wir mit den Nachteilen von *Node* und JavaScript gut umgehen und die Vorteile – eine einfach zu lernende Sprache und ein großen Ökosystem von Modulen – erfolgreich ausnutzen. Für ähnliche Projekte, gerade für kleinere Webserver, würden wir es wieder verwenden.

4.2 Ausbaumöglichkeiten

4.2.1 Zeitpunkte für Daten-Aktualisierung

Mit steigender Anzahl von Serien in der Datenbank wird das Aktualisieren der Daten und damit auch das Laden neuer Episoden über die APIs immer länger dauern. Aktuell existiere nur die Möglichkeit, alle Serien-Daten zu aktualisieren.

Eine Verbesserungs-Möglichkeit wäre, ein Scheduling-System zu verwenden, indem einzelne Serien zu bestimmten Zeitpunkten aktualisiert werden. Während laufende Serien so z.B. einmal pro Woche (etwa zwei Tage vor dem Ausstrahlen einer neuen Episode) aktualisiert werden, könnten beendete Serien weniger häufig aktualisiert werden, da sich diese Daten mit hoher Wahrscheinlichkeit nicht mehr ändern werden.

[20] Generatoren sind vergleichbar mit den z.B. aus Lua bekannten Coroutinen. Im Grunde sind es Funktionen, welche an gewissen Punkten pausiert werden und später fortgesetzt werden können.

4.2.2 Durchschnitts-Bewertungen speichern

Aktuell werden Durchschnitts-Bewertungen immer dynamisch berechnet. Für Serien ist dies jedoch mit steigender Anzahl Bewertung recht zeitaufwendig, da immer alle Bewertungs-Einträge gelesen werden müssen. Würde man diese Durchschnitt beim Anlegen bzw. Aktualisieren einer Bewertung berechnen und in der Serie oder Episode speichern, könnte diese Zeit sparen. (Es wird angenommen, dass die Serien bzw. Episoden und deren Durchschnitte häufiger abgefragt werden als Bewertungen gespeichert werden.)

Dies könnte entweder in der *node*-Anwendung beim Speichern einer Bewertung oder aber in der *Postgres*-Datenbank über *Trigger* realisiert werden.

4.2.3 Empfehlungen basierend auf bisherigen Bewertungen

Als Benutzer von EpisodeFever möchte ich nicht nur mir bekannte Serien bewerten, sondern auch neue entdecken. Ein bekannter und hilfreicher Mechanismus dazu ist das Darstellen von Empfehlungen, welche auf Basis von den von mir und den von anderen Benutzern abgegebenen Bewertungen.

Viele Algorithmen zum Bestimmen von Empfehlungen beziehen sich auf den E-Commerce-Markt, bei dem es darum geht, Benutzern Produkte zu empfehlen, die sie vielleicht auch kaufen wollen. Häufig sind die einzigen Metriken dazu positive Ereignisse, z.B. “Benutzer A hat Produkt X angesehen” oder “Benutzer A kaufte Produkt X”.

Die von EpisodeFever erfassten Bewertungen bieten dabei detailliertere Informationen. Sie bilden dabei nicht nur die Relation “Benutzer A mag Episode X” (und darüber transitiv auch “Benutzer A mag Serie Z”) ab, sondern auch “Benutzer gefällt Episode B nicht” bzw. “Benutzer ist Episode C gegenüber indifferent”. Ein möglicher Empfehlungs-Mechanismus muss auch diese Zu- oder Abneigung verarbeiten, um alle Informationen auszunutzen und möglichst genau zu arbeiten.

4.2.4 Benutzern das explizite Verwalten von betrachteten Serien erlauben

In der aktuellen Form der Anwendung werden die Relationen zwischen Benutzern und Serien nur implizit über die abgegebenen Bewertungen gesetzt. Wurde eine Serie bewertet, so wird angenommen, dass der Benutzer sie schaut.

Dies hat jedoch zwei Schwachstellen:

1. Der Benutzer kann keine Serie explizit entfernen, sodass sie nicht mehr in seiner Liste zukünftiger Episoden auftaucht.
2. Es können keine Serien vorgemerkt werden, welche zwar schon angekündigt wurden, wo aber noch keine Episode ausgestrahlt wurde.

Möglicher Lösungsansatz wäre hier, eine neue Relation *Watch*[21] hinzuzufügen. In dieser wird bei der ersten Bewertung automatisch ein Eintrag mit Benutzer- und Serien-ID erzeugt. Zusätzlich wird dem Benutzer aber auch ermöglicht, selbst diese Relation zu erzeugen, eine Liste mit *Watches* einzusehen und Einträge daraus zu entfernen.

4.2.5 Server-Infrastruktur

Um die entwickelte Software betreiben zu können, wird (mindestens) ein korrekt konfigurierter Server benötigt, auf welchen die *node*-Anwendung sowie *Postgres* ausgeführt wird. Dies ist bereits

[21] Von engl. *schauen*, *betrachten* (“eine Fernsehserie schauen”), nicht *Armbanduhr*.

nötig, um die Software entwickeln und testen zu können. Die Hardware-Anforderungen für einen solchen Server sind im Vergleich zu existierenden Angeboten[^vserver-preise] nicht besonders hoch; Die *node*-Anwendung braucht auf dem Entwicklungssystem etwa 36MB RAM und minmal CPU-Leistung. Da die in der Postgres-Datenbank gespeicherten Daten ebenfalls nicht sehr groß sind, sollte es möglich sein, auf einem Server mit 2GB RAM die gesamte Datenbank im Hauptspeicher zu halten.

Um statische Dateien performant auszuliefern (z.B. Grafiken) und komprimierte Verbindungen sowie verschlüsselte Protokolle wie HTTPS und HTTP2 verwenden, sollte zudem ein Web-Server wie nginx[22] als *Reverse Proxy* verwendet werden, welcher im vor der *node*-Anwendung liegt und an diese die relevanten HTTP-Anfragen weitergibt.

Es ist zu bedenken, dass Anwendungen auf Basis von *node* standardmäßig zwar asynchronen Kontrollfluss verwenden, jedoch JavaScript nur in einem Thread ausführen. Auf diese Weise kann der JavaScript-Teil einer *node*-Anwendung keinen Gebrauch von Mehrprozessor-Systemen machen. Eine recht einfache Lösung dafür ist es, die Anwendung in einem *Cluster* zu verwenden. Dabei startet ein Master-Prozess die Anwendung in einer beliebigen Anzahl von Kind-Prozessen und gibt Netzwerk-Anfragen an diese weiter.

Cluster ist ein mit *node* mitgeliefertes Modul. Es ist jedoch von Vorteil, auf einen komplexeres Tool wie PM2[23] zurückzugreifen. Dieses kann nicht nur mehrere Anwendungen ohne Code-Änderungen als Cluster starten, sondern stellt auch fest, wenn diese abstürzen und kann sie dann neu starten. Des Weiteren bietet es einige Möglichkeiten, Log-Ausgaben mitzuschreiben und CPU- und Speicher-Auslastung pro Prozess einzusehen.

4.2.6 Analyse von Protokoll-Daten

Die Server-Anwendung schreibt Informationen zu jeder gesendeten HTTP-Antwort sowie zu jeder SQL-Abfrage auf `stdout` (d.h., das Terminal, wenn die Anwendung im Vordergrund läuft, ansonsten das System-Log). Die Protokoll-Daten beinhalten einen Zeitstempel, Art der Anfrage, de Rückgabewert (Fehler-Code) und wie lange der Server brauchte, um die Anfrage zu bearbeiten. Diese Daten müssen im aktuellen Zustand von einem Entwickler oder Administrator persönlich ausgewertet werden, was das Auffinden von Fehlern von Geschwindigkeits-Problemen erschwert.

Um diese Daten analysieren und darstellen zu können, gibt es passende Software, welche teilweise unter einer Open-Source-Lizenz (vgl. [2]) verfügbar ist. Ein Beispiel hierfür ist die Kombination aus *Logstash*, *Elasticsearch* und *Kibana* (“ELK”, [60]).

Mit einer angepassten Konfiguration sollte es möglich sein, die Protokolle der EpisodeFever-Anwendung mit *Logstash* einlesen zu lassen und die Daten in eine *Elasticsearch*-Datenbank zu schreiben. Mit *Kibana* lassen sich diese dann grafisch darstellen, filtern und auswerten.

5 Anhang

5.1 Beiträge zu Open Source

Im Rahmen des Projektes wurden einige Beiträge zu Open-Source-Software gemacht bzw. neue Software-Module unter einer Open-Source-Lizenz veröffentlicht.

- `sortStringToSql`[24]

[22] http://nginx.org/
[23] https://github.com/Unitech/pm2
[24] https://github.com/killercup/sortStringToSql

Ein von Pascal Hertleif geschriebenes Node-Module, welches eine Funktion bereitstellt um URL-kompatible Anweisungen für Sortierung in SQL zu konvertieren. So wird z.B. `date,id"` zu `"date DESC NULLS LAST, id ASC NULLS LAST"` konvertiert.

- `replay`[25] [47]

  Pascal Hertleif reichte hierzu einen Patch ein, um Query-Strings beim Aufnehmen von HTTP-Anfragen zu speichern (und somit auch URLs anhand der Query-Strings unterscheiden zu können). Dieser Patch ist als Teil von Version 0.11 verfügbar.

Bibliographie

[1] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures“, Doctoral dissertation, University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[2] „Open Source Licenses“. http://opensource.org/licenses. (Zugegriffen: 2. Februar 2015)

[3] D. Ryan, „Node.js“ (Version 0.10.32), 2009. http://nodejs.org/. (Zugegriffen: 22. September 2014)

[4] I. Z. Schlueter, „node packaged modules“. http://npmjs.org/. (Zugegriffen: 1. Februar 2015)

[5] The PostgreSQL Global Development Group, „PostgreSQL“ (Version 9.4.0). http://postgresql.org/. (Zugegriffen: 1. Februar 2015)

[6] MongoDB, Inc., „MongoDB“ (Version 2.6.7). http://www.mongodb.org/. (Zugegriffen: 1. Februar 2015)

[7] MariaDB Foundation, „MariaDB“ (Version 10.0.16). https://mariadb.org/. (Zugegriffen: 8. Februar 2015)

[8] T. Holowaychuk, A. Heckmann, C. Jessup, D. C. Wilson, G. Rauch, J. Ong, R. Shtylman, und Y. J. Sim, „express.js“ (Version 4.11.1). http://expressjs.org/. (Zugegriffen: 1. Februar 2015)

[9] M. Cavage, „Restify“ (Version 2.8.5). http://mcavage.me/node-restify/. (Zugegriffen: 1. Februar 2015)

[10] E. Hammer, „HAPI“ (Version 8.1.0). http://hapijs.com/. (Zugegriffen: 2. Februar 2015)

[11] T. Holowaychuk, J. Ong, J. Gruber, und Y. He, „Koa.js“ (Version 0.16.0). http://koajs.com/. (Zugegriffen: 2. Februar 2015)

[12] D. H. Hansson, „Ruby on Rails“ (Version 4.2.0). http://rubyonrails.org/. (Zugegriffen: 2. Februar 2015)

[13] Django Software Foundation, „Django“ (Version 1.7.4). https://www.djangoproject.com/. (Zugegriffen: 2. Februar 2015)

[14] T. Otwell, „Laravel“ (Version 4.2.0). http://laravel.com/. (Zugegriffen: 2. Februar 2015)

[15] J. Saenz, „Martini“ (Version 1.0.0). http://martini.codegangsta.io/. (Zugegriffen: 2. Februar 2015)

[16] T. Griesser, „bookshelf.js“ (Version 0.7.9). http://bookshelfjs.org/. (Zugegriffen: 2. Februar 2015)

[17] T. Griesser, „knex.js“ (Version 0.7.3). http://knexjs.org/. (Zugegriffen: 2. Februar 2015)

[18] G. Rauch, „mongoose“ (Version 3.8.23). http://mongoosejs.com/. (Zugegriffen: 7. Februar 2015)

[25] https://github.com/assaf/node-replay/pull/50

[19] B. Cavalier und D. Denicola, „Promises/A+“. 6. Dezember 2012. http://promisesaplus.com/. (Zugegriffen: 26. September 2014)

[20] P. Antonov, „Bluebird“ (Version 2.9.5). https://github.com/petkaantonov/bluebird. (Zugegriffen: 2. Februar 2015)

[21] G. Kosev, „Perfomance of Promise Libraries“. http://spion.github.io/posts/why-i-am-switching-to-promises.html. (Zugegriffen: 2. Februar 2015)

[22] S. Baker, „RSpec“ (Version 3.1.7). http://rspec.info/. (Zugegriffen: 2. Februar 2015)

[23] C. Amavisca, J. Boyens, und G. Van Hove, „Jasmine“ (Version 2.2.1). http://jasmine.github.io/. (Zugegriffen: 8. März 2015)

[24] A. Sellier, C. Robbins, und J. Sievert, „Vows“ (Version 0.8.1). http://vowsjs.org/. (Zugegriffen: 8. März 2015)

[25] C. Solís und X. Wang, „A study of the characteristics of behaviour driven development“, in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, 2011, S. 383–387.

[26] J. Luer, „Chai“ (Version 1.10.0). http://chaijs.com/. (Zugegriffen: 2. Februar 2015)

[27] G. Rauch, „expect.js“ (Version 0.3.1). https://github.com/Automattic/expect.js. (Zugegriffen: 8. März 2015)

[28] T. Holowaychuk, „SuperTest“ (Version 0.15.0). https://github.com/tj/supertest. (Zugegriffen: 2. Februar 2015)

[29] T. Holowaychuk, „SuperAgent“ (Version 0.21.0). https://github.com/visionmedia/superagent. (Zugegriffen: 6. Februar 2015)

[30] P. Kleinschmidt und C. Rank, *Relationale Datenbanksysteme: Eine praktische Einführung*, 3. Aufl. Springer Verlag, 2005.

[31] T. Griesser, „bcrypt“ (Version 0.5.1). https://github.com/tgriesser/checkit. (Zugegriffen: 16. März 2015)

[32] N. Campbell, „bcrypt“ (Version 0.8.1). https://github.com/ncb000gt/node.bcrypt.js. (Zugegriffen: 16. März 2015)

[33] M. Dietz, „Improving user authentication on the web: Protected login, strong sessions, and identity federation“, Doctoral dissertation, Rice University, 2014. https://scholarship.rice.edu/bitstream/handle/1911/76484/Dissertation_final.pdf?sequence=1

[34] J. Hanson, „passport“ (Version 0.2.1). http://passportjs.org/. (Zugegriffen: 16. März 2015)

[35] Auth0, Inc., „jsonwebtoken“ (Version 4.2.0). https://github.com/auth0/node-jsonwebtoken. (Zugegriffen: 16. März 2015)

[36] A. Reinman, „Nodemailer“ (Version 1.3.2). https://github.com/andris9/Nodemailer

[37] V. Sahni, „Best Practices for Designing a Pragmatic RESTful API“, 28. Oktober 2013. http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api. (Zugegriffen: 8. Februar 2015)

[38] S. Klabnik und Y. Katz, „JSON API“, 25. November 2014. http://jsonapi.org/

[39] Cozy Cloud, „cozy-ical“ (Version 1.1.5). https://github.com/mycozycloud/cozy-ical

[40] M. Squires und M. Bergman, „faker.js“ (Version 2.1.2). https://github.com/Marak/faker.js. (Zugegriffen: 8. März 2015)

[41] J. Walters, S. Zsori, und P. Taylor, „TheTVDB“. http://thetvdb.com/. (Zugegriffen: 5. Februar 2015)

[42] „Creative Commons Attribution 3.0 United States“. http://creativecommons.org/licenses/by/3.0/us/. (Zugegriffen: 6. Februar 2015)

[43] M. Kubica, „xml2js“ (Version 0.4.4). https://github.com/Leonidas-from-XIV/node-xml2js. (Zugegriffen: 4. Februar 2015)

[44] I. Z. Schlueter, „sax-js“ (Version 0.6.1). https://github.com/isaacs/sax-js/. (Zugegriffen: 4. Februar 2015)

[45] E. Naggum, „The Long, Painful History of Time“. 11. Oktober 1999. http://naggum.no/lugm-time.html. (Zugegriffen: 4. Februar 2015)

[46] K. Ainsworth, B. Drummond, R. Anne, und V. Troughton, „TVRage“. http://tvrage.com/. (Zugegriffen: 5. Februar 2015)

[47] A. Assaf, „replay“ (Version 1.12.0). https://github.com/assaf/node-replay. (Zugegriffen: 7. Februar 2015)

[48] S. Banon und Elasticsearch BV, „ElasticSearch“ (Version 1.4.4). https://www.elastic.co/products/elasticsearch. (Zugegriffen: 16. März 2015)

[49] The Apache Software Foundation, „Apache Lucence“ (Version 5.0.0). http://lucene.apache.org/. (Zugegriffen: 16. März 2015)

[50] Bibliographisches Institut GmbH, „Duden: Lexem“. http://www.duden.de/node/704634/revisions/1395925/view. (Zugegriffen: 16. April 2015)

[51] J. Harding, V. Skarich, und T. Trueman, „typeahead.js“ (Version 0.10.5). http://twitter.github.io/typeahead.js/. (Zugegriffen: 16. April 2015)

[52] Microsoft Open Technologies, Inc., „Reactive-Extensions“ (Version 2.4.7). http://reactivex.io/. (Zugegriffen: 16. April 2015)

[53] R. Pominov, „kefir.js“ (Version 1.3.1). https://pozadi.github.io/kefir/. (Zugegriffen: 16. April 2015)

[54] I. Maier, T. Rompf, und M. Odersky, „Deprecating the observer pattern“, EPFL, EPFL Report 148043, Apr. 2010. http://infoscience.epfl.ch/record/148043/files/DeprecatingObserversTR2010.pdf. (Zugegriffen: 24. September 2014)

[55] E. Wellbrook, „node-tvdb“ (Version 0.4.13), 15. April 2015. https://github.com/edwellbrook/node-tvdb. (Zugegriffen: 16. April 2015)

[56] ECMA International, „Draft Specification for ES.next (Ecma-262 Edition 6) (Draft Rev 27)“, 24. August 2014. http://wiki.ecmascript.org/lib/exe/fetch.php?id=harmony%3Aspecification_drafts&cache=cache&media=harmony:working_draft_ecma-262_edition_6_08-24-14-nomarkup.pdf. (Zugegriffen: 23. September 2014)

[57] N. C. Zakas, „ESLint“ (Version 0.19.0), 11. April 2015. http://eslint.org/

[58] Microsoft Corp., „TypeScript“ (Version 1.4.0), 13. Januar 2015. http://www.typescriptlang.org/. (Zugegriffen: 16. April 2015)

[59] Facebook, Inc., „Flow“ (Version 0.8.0), 3. April 2015. http://flowtype.org/. (Zugegriffen: 16. April 2015)

[60] Elasticsearch, „The Elasticsearch ELK Stack“. http://www.elasticsearch.org/overview/. (Zugegriffen: 8. Februar 2015)