Software Engineering I (IB)

Blatt 4

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 06.12.2018 06:56

Aufgabe — Node.js (Abgabe bis 16.01.2019, 08:00 Uhr)

Implementieren Sie ein Backend für eine Multiplex-Kino-App, in Form eines Node.js-Server.

Sie können diese Aufgabe wieder zu zweit oder alleine abgeben. Nutzen Sie den GitHub-Classroom-Link `https://classroom.github.com/g/yBCs8MxR`. Gehen Sie nach dem Git-Flow bzw. GitHub-Flow vor. Die Abgabe erfolgt wieder per Pull-Request gegen den `master`-Branch. Im Rahmen des Praktikums am 17.01.2019 müssen Sie Ihr Backend auf dem Beamer kurz (max. 5 Minuten) präsentieren.

Lesen Sie sich die `README.md` in Ihrem Repository durch.

Das interne Modell

Der Aufbau des Modells innerhalb des Backend ist durch die folgenden Klassen vorgegeben. Erstellen Sie alle diese Klassen in einem Unterverzeichnis `model` innerhalb des `src`-Verzeichnisses. Setzen Sie in **allen** Klassen **alle** Objektvariablen auf `private`. Nutzen Sie zum Zugriff von außen Accessors, die Sie bei Bedarf hinzufügen. Erzeugen Sie die Objekte immer mit Konstruktoren.

Filme

Ein `Movie`-Objekt enthält den Titel des Filmes, sein FSK-Alter als Zahl und die Laufzeit des Filmes in Minuten.

Kinos

Ein `Cinema`-Objekt hat eine ID, eine Anzahl von Sitzreihen und eine Anzahl von Plätzen pro Reihe. Das es damit nicht möglich ist Kinos mit unterschiedlicher Anzahl von Sitzplätzen in verschiedenen Reihen zu modellieren, nehmen wir für diese erste Version in Kauf.

Kunden

Ein `Customer`-Objekt hat einen Namen und eine E-Mail-Adresse. Nutzen Sie das Package email-validator um im Konstruktor zu validieren ob die übergebene E-Mail-Adresse die richtige Form hat. Wenn nicht, setzen Sie im Konstruktor die E-Mail-Adresse auf `null`. Außerdem hat der Kunde eine Map die seine Reservierungen enthält. Schlüssel ist die Reservierungsnummer.

Vorstellung

Ein `Screening`-Objekt enthält einen Film und ein Kino in dem er gezeigt wird, sowie die geplante Länge der Werbung und Vorfilme in Minuten. Außerdem zwei `Date`-Objekte `start` und `end`. Dem Konstruktor wird das Kino, der Film, ein `Date`-Objekt für den Startzeitpunkt und **optional** die geplante Länge der Werbung und Vorfilme in Minuten. Wird keine Länge übergeben, soll als Default 15 Minuten genommen werden. Das `Date`-Objekt für das Ende wird im Konstruktor dann über die Filmlänge und die Werbungslänge berechnet.

Außerdem hat das `Screening`-Objekt eine Map von Maps in der es die Reservierungs-IDs speichert:

```
reservations: Map<number, Map<number, number>>
```

Schlüssel der äußeren Map sind die Reihen, beginnend mit 0. Werte sind jeweils Maps, deren Schlüssel die Sitznummern, beginnend mit 0, innerhalb einer Reihe sind. Als Werte darin stehen die Reservierungsnummern. Ein Sitzplatz Reihe x, Sitz y ist genau dann belegt, wenn es einen Eintrag in der Map der Reihe x gibt, bei dem y der Schlüssel ist. Ein Platz ist frei, wenn es **keinen** entsprechenden Eintrag gibt **und** x eine gültige Reihe ist und y ein gültiger Sitzplatz innerhalb der Reihe.

Die `Screening`-Klasse implementiert außerdem folgende zwei Methoden

```
isReservationPossible(seats: Array<[number, number]>): boolean
addReservation(reservationID: number, seats: Array<[number, number]>): boolean
```

Die Methode `isReservationPossible` überprüft ob alle übergebenen Sitzplätze existieren und frei sind. Die Elemente im Array sind Tupel bestehend aus der Reihen- und Sitznummer. Sobald ein einziger Platz nicht frei ist oder es einen Sitz gar nicht gibt, ist die gesamte Reservierung nicht möglich.

Die `addReservation`-Methode führt eine Reservierung durch indem die Einträge in der Map gemacht werden, aber nur wenn die gesamte Reservierung möglich ist!

Reservierung

Abgesehen davon, dass die Reservierung im Screening-Objekt gespeichert werden, gibt es noch die `Reservation`-Objekte, die die Reservierung einem Kunden zuordnen. Ein `Reservation`-Objekt enthält die Kundennummer, die Vorstellungsnummer, die reservierten Plätze als Array von Tupeln, bestehend aus Reihen- und Sitznummer und den Reservierungsstatus.

Wird eine Reservierung neu erstellt, ist sie im Zustand `Reserved`. Sobald Sie bezahlt wurde, ist Sie im Zustand `Paid`. Dies kann nur durch die Methode `pay` umgesetzt werden. Mit der Methode `isPaid` kann überprüft werden, ob schon gezahlt ist, oder nicht. Eine Reservierung kann dann, solange sie noch nicht gezahlt ist, kostenfrei storniert werden. Nach dem Stornieren mit der Methode `cancel` ist die Reservierung im Zustand `Cancelled`. Gecancelte Reservierungen bleiben aber drundsätzlich im System. Die Methode `cancel` gibt einen Wahrheitswert zurück, der aussagt ob das Canceln erfolgreich war. Eine stornierte Reservierung kann übrigens auch erfolgreich storniert werden. Sie bleibt dann einfach storniert.

Die drei Zustandswerte der Reservrierung implementieren Sie am Besten als `enum ReservationState`.

Multiplex-Kino

Ein `Multiplex`-Objekt verwaltet verschiedene Maps, deren Schlüssel immer eindeutige IDs (`number`) bezüglich der Werte in der Map sind. Maps gibt es für

- Filme
- Kinos
- Kunden
- Vorstellungen
- Reservierungen

Außer bei den Kinos werden die IDs von dem `Multiplex`-Objekt fortlaufend generiert.

Das `Multiplex`-Objekt ist verantwortlich dafür die verschiedene Maps in einem konsistenten Zustand zu halten. Das bedeutet z.B. dass beim Erzeugen einer Reservierung die entsprechenden Einträge sowohl bei den Vorstellungen als auch bei den Reservierungen gemacht werden.

Anmerkung: Grundsätzlich sind wir beim Design einer Anwendung bemüht, dass keine Daten an redundant mehreren Stellen gehalten werden. Allerdings kann das die Anwendungen ausbremsen. Wenn wir die Reservierungsinformation z.B. nur in der Reservierungs-Map speichern würden, müssten zur Berechnung der Information,

welche Sitzplätze in einer Vorstellung noch frei sind, alle Einträge der Reservierungs-Map durchlaufen werden. Daher wird in diesem Design an manchen Stellen davon abgewichen, grundsätzlich keine redundanten Daten zu speichern. Das kann evtl. in einer zukünftigen Version (die wir nicht mehr implementieren werden), durch Nutzung eines DBMS wieder vermieden werden.

Das `Multiplex`-Objekt hat in unserer ersten Version Methoden für folgendes:

- Hinzufügen eines Kinos.
  - ID muss übergeben werden, aber trotzdem eindeutig sein. Es dürfen keine zwei Kinos die selbe ID haben.
  - Ergebnis ist ein Wahrheitswert, der aussagt ob das Hinzufügen funktioniert hat.
- Hinzufügen eines Filmes.
  - Die ID wird generiert, beginnend mit 1.
  - Ergebnis ist die ID.
- Hinzufügen eines Kunden.
- Die ID wird generiert, beginnend mit 1.
- Ergebnis ist die ID.
- Hinzufügen einer Vorstellung.
  - Die ID wird generiert, beginnend mit 1.
  - Ergebnis ist die ID oder ein `string` der den Grund angibt, warum das Hinzufügen nicht erfolgreich war.
  - Gründe für den Misserfolg können sein:
    * Die übergebene Kino-ID gibt es gar nicht.
    * Die übergebene Film-ID gibt es gar nicht.
    * In dem selben Kino findet in einem überlappenden Zeitraum bereits eine Veranstaltung statt. Zwischen den Veranstaltungen muss immer mindestens eine Viertel Stunde liegen.
- Sitzplätze reservieren.
  - Die ID wird generiert, beginnend mit 1.
  - Ergebnis ist die ID oder ein `string` der den Grund angibt, warum das Reservieren nicht erfolgreich war.
  - Gründe für den Misserfolg können sein:
    * Die übergebene Kunden-ID gibt es gar nicht.
    * Die übergebene Vorstellungs-ID gibt es gar nicht.
    * Die Sitze gibt es nicht **alle** oder nicht **alle** sind frei.
  - Wenn die Reservierung erfolgreich war, muss Sie an allen notwendigen Stellen (siehe Beschreibung der verschiedenen Klassen) eingetragen sein. Implementieren Sie noch eventuell notwendige Methoden `addReservation` in den betroffenen Klassen.

- Canceln einer Reservierung.
- Bezahlen einer Reservierung.

Eine eigene Initialisierung

Schreiben Sie sich eine Funktion `bootstrap` in einer Datei `src/db.ts` in der Sie mit Hilfe der von Ihnen implementierten Methoden ein Multiplex-Kino erzeugen, das

- mindestens 2 Kinos,
- mindestens 2 Filme,
- mindestens 2 Vorstellungen,
- mindestens 2 Kunden und
- mindestens 2 Reservierungen hat.

Das Web-API

Über einen Express-Server soll Zugriff auf die Funktionalität des Multiplex-Kinos geschaffen werden. Für diese Abgabe müssen Sie nur ein paar wenige Zugriffsmöglichkeiten implementieren. Es ist aber eine gute Übung (und Vorbereitung auf das kommende Semester) wenn Sie es weiter ausimplementieren und sogar die Angular-App von Blatt 3 so anpassen, dass sie dieses Backend nutzt.

Die vom Backend per HTTP übergebenen Werte entsprechen nicht den intern verwalteten Objekten. Daher müssen Schritt für Schritt alle Klassen um Getter ergänzt werden, die die internen Daten entsprechend repräsentiert zurück geben.

Bootstrappen Sie in der Datei `server.ts` zunächst das Multiplex-Kino mit der von Ihnen implementierten Bootstrap-Funktion.

Implementieren Sie anschließend am Besten eine Zugriffsmethode nach der anderen und fügen Sie dabei nur die jeweils dafür benötigten Getter zu den anderen Klassen hinzu. Im Folgenden wird die jeweilige URL angegeben und das im Browser zu sehende Ergebnis beschrieben.

Mit dem Browser ohne eigenes Frontend funktionieren allerdings nur die GET-Requests. Sie können sich z.B. den in der Vorlesung gezeigten Insomnia-Client herunterladen um auch POST-, PUT- und DELETE-Requests auszuprobieren.

GET `http://localhost:3000/api/movies`

Ergebnis ist ein JSON-Array aller im Kino laufenden Filme in der folgenden Struktur, z.B.:

```
[
  {
```

```
    "id": 1,
    "title": "Am Limit",
    "duration": 95,
    "fsk": 6
  },
  {
    "id": 2,
    "title": "Inglourious Basterds",
    "duration": 153,
    "fsk": 16
  }
]
```

Für dieses Array, reicht es z.B. einen Getter `getMovies` in der Klassen `Multiplex` und Getter (Accessors) für die einzelnen Felder eines Filme in der Klasse `Movie` zu implementieren.

Sie können dann die Map in `Multiplex` einfach mit einer Schleife durchlaufen, z.B.

```
for (const [id, movie] of this.movies.entries()) {
  ...
}
```

und haben dann die Movie-ID als `id` und den Movie als `movie`. In der Schleife erzeugen Sie dann so ein JSON-Objekt "on-the-fly", z.B. mit

```
const exportMovie = {
  id, // Kurzform für `id: id`, sonst meckert `npm run lint`
  title: movie.title,
  duration: movie.duration,
  fsk: movie.fsk,
};
```

und fügen es zu einem, vor der Schleife noch leerem, Array hinzu, das Sie dann zurück geben. Wenn Sie dieses Array dann einfach in der Datei `server.ts` mit `res.send` an den Aufrufer zurück senden, kommt wie das oben dargestellte JSON-Array im Browser an.

GET `http://localhost:3000/api/movies/:id`

Wenn die `:id` nicht in der Movies-Map existiert, ist das Ergebnis z.B.

```
{
  "error": "movie id 4 is unknown"
}
```

Existiert die `:id` so soll der eine Film mit allen seinen Vorstellungen in der folgenden, durch das Beispiel illustierten, Form zurück gegeben werden:

```
{
  "id": 2,
  "title": "Inglourious Basterds",
  "duration": 153,
  "fsk": 16,
  "screenings": [
    {
      "id": 1,
      "date": "22.12.2018",
      "time": "20:00:00"
    },
    {
      "id": 2,
      "date": "22.12.2018",
      "time": "22:00:00"
    },
    {
      "id": 3,
      "date": "23.12.2018",
      "time": "22:00:00"
    }
  ]
}
```

Die Angaben für `date` und `time` können Sie mit den Methoden `toLocaleDateString` und `toLocalTimeString` berechnen. Nutzen Sie als `locales`-Parameter den String `'de-DE'`.

POST `http://localhost:3000/api/screenings/:id`

Mit diesem POST-Request soll eine neue Reservierung durchgeführt werden. Die Screening-ID ist bereits Teil der URL. Die zusätzlichen Informationen werden durch ein JSON-Objekt im Body übertragen, z.B.

```
{
    "customerID": 1,
    "seats": [[1,3],[1,4]]
}
```

Wenn Sie den `bodyParser` eingebunden haben (ist im Startercode schon gemacht):

```
app.use(bodyParser.json());
```

ist das JSON-Objekt im Server dann unter `req.body` verfügbar, also z.B.

```
app.post(apiurl + '/screenings/:id', (req, res) => {
  const customerID = req.body.customerID;
  const seats = req.body.seats;
  // Achtung: Das ist **nicht** das richtige Ergebnis:
  res.send({ customerID: customerID, seats: seats });
});
```

Wenn die Screening-ID nicht existiert, soll das Ergebnis z.B. sein:

```
{
  "error": "Screening ID 4 unknown"
}
```

Wenn die Kundennummer nicht existiert:

```
{
  "error": "Customer ID 5 unknown"
}
```

Wenn die Sitze nicht (mehr) zur Verfügung stehen oder gar nicht vorhanden sind:

```
{
  "error": "seats not available"
}
```

Wenn die Buchung erfolgreich war, wird als Ergebnis die Reservierungsnummer in einem JSON-Objekt zurück gegeben, z.B.

```
{
  "reservationID": 3
}
```

Weitere Methoden

Alle weiteren Zugriffsmethoden können Sie zur Vervollständigung gerne optional implementieren. Für die Abgabe und den Zulassungsschein ist das nicht notwendig.

Deployment

Deployen Sie Ihren Node.js-Server, spätestens zur Abgabe, auf Heroku, wie in der Vorlesung gezeigt wurde.