Schnittstellen von Node-Applikationen effizient testen mit Supertest

Node.js-Module: Supertest

Die leichtgewichtige Bibliothek Supertest bietet Hilfe bei der Formulierung von Schnittstellentests für eine Node-Applikation. Sie erlaubt Zugriff auf alle Elemente der Anfrage und der Antwort des Servers. Außerdem lässt sie sich komfortabel mit anderen Test-Frameworks wie Koa, Jest oder Express kombinieren, um den API-Layer Ihrer Applikation ohne großen Aufwand abzusichern.

von Sebastian Springer

Eines der häufigsten Einsatzgebiete von Node.js ist die Implementierung von Web-Backends. Das legen allein schon die Downloadzahlen von Paketen wie Express nahe, die mittlerweile die Marke von zehn Millionen pro Woche geknackt haben. Express basiert auf dem HTTP-Paket von Node und erweitert es um Routingfunktionalität sowie um ein Plug-in-System, das als Middleware-Layer bezeichnet wird. Je wichtiger eine solche Express-Applikation für ein Unternehmen ist, desto größer wird auch die Notwendigkeit, eine Applikation zu testen. Durch die Vielzahl von Test-Frameworks sollte die automatisierte Überprüfung solcher Applikationen mittlerweile kein Problem mehr darstellen. Und trotzdem gibt es noch eine viel zu große Zahl von Applikationen, die nach wie vor ungetestet ist. In diesem Artikel möchte ich Ihnen mit Supertest eine Bibliothek vorstellen, die es Ihnen erlaubt, die Schnittstellen Ihrer Applikation mit überschaubarem Aufwand zu testen.

Ursprünglich wurde Supertest von TJ Holowaychuk entwickelt, dem Entwickler, der auch hinter Express und zahlreichen weiteren Bibliotheken steht. Die Ursprünge von Supertest gehen bis zum Juni 2012 zurück. Mittlerweile ist etwas Ruhe in die Entwicklung der Bibliothek eingekehrt und es laufen nur noch vereinzelte Commits ins Repository. Dennoch wird es aktiv maintaint und hat ein stabiles API. Das bedeutet, dass einem Einsatz in einer Applikation nichts im Weg steht. Supertest basiert auf der Bibliothek Superagent, die asynchrone Requests für JavaScript-Applikationen ermöglicht. Superagent kann sowohl im Browser als auch in Node verwendet werden. Auch Superagent wurde von TJ Holowaychuck entwickelt. Hier liegt der initiale Commit sogar noch länger zurück (August 2011). Superagent wird in Supertest zur Formulierung der Anfragen gegen die Schnittstelle Ihrer Applikation verwendet. In Ihren Tests können Sie auf den gesamten Funktionsumfang von Superagent zurückgreifen und haben sämtliche Aspekte der Anfrage unter Ihrer Kontrolle.

Wie nahezu alle Bibliotheken in der JavaScript-Welt wird auch Supertest auf GitHub maintaint und ist als npm-Paket verfügbar. Installiert wird Supertest mit dem Kommando *npm install -D supertest*. Da es sich bei der

Bibliothek um ein Testwerkzeug handelt, installieren Sie es als *devDependency*, die in der Regel nicht mit in die Produktivapplikation einfließt.

Ein erster Test

Mit Abstand am häufigsten wird Supertest mit Express verwendet, Sie können die Bibliothek jedoch nicht nur mit diesem Framework, sondern auch mit zahlreichen weiteren Lösungen wie beispielsweise Koa verwenden. Voraussetzung, dass Supertest funktioniert, ist, dass Sie eine *http.Server*-Instanz übergeben. Für die folgenden Beispiele kommt ein Express-Server als Applikation zum Einsatz. Bevor Sie mit der eigentlichen Entwicklung beginnen, sollten Sie zunächst eine *package.json*-Datei mit dem Kommando *npm init -y* anlegen und anschließend Express mit dem Kommando *npm install express* installieren. Für einen ersten Test reicht eine Route, die eine JSON-Struktur zurückgibt. Listing 1 enthält den initialen Servercode.

Supertest kann zwar auch ohne weitere Bibliotheken eingesetzt werden, es ist jedoch erheblich komfortabler, wenn Sie es mit einem Test-Framework wie beispielsweise Jest kombinieren. Also installieren Sie am besten gleich noch Jest mit dem Kommando *npm instal -D jest*. Mit diesem Set-up können Sie nun Ihren ersten API-Test verfassen, mit dem Sie die */user*-Route testen können. Den Test finden Sie in Listing 2.

Die *describe*- und *it*-Funktionen werden von Jest zur Verfügung gestellt, um die Tests zu gruppieren und selbst zu schreiben. Die *request*-Funktion stammt aus dem Supertest-Paket und akzeptiert, wie schon erwähnt, ein *http.Server*-Objekt. Das *app*-Objekt der Express-Applikation erfüllt diese Anforderung, also kann es als Argument verwendet werden. Eine Besonderheit bei Supertest ist, dass der Server an einen freien Port gebunden wird, falls das noch nicht der Fall ist. Im Beispiel ist der Server bereits auf Port 8080 gebunden, also entfällt die zusätzliche Bindung. Supertest implementiert ein Fluent Interface, das bedeutet, dass jede Methode wiederum ein Request-Objekt zurückgibt. Dadurch lassen sich die einzelnen Methodenaufrufe zu einer übersichtlichen Kette zusammenfügen. Nachdem Sie das initiale *request*-Objekt erzeugt haben, geben Sie an, welche HTTP-Methode verwendet und welcher Pfad angefragt werden soll. Im Beispiel ist das eine *get*-Anfrage auf den Pfad */user*. Mit Hilfe der *expect*-Methoden können Sie Ihre Erwartungen an die Antwort definieren. So soll die Antwort des Servers den Statuscode 200 und einen Content-Type-Header mit dem angegebenen Wert aufweisen. Entspricht die Antwort des Servers nicht den Erwartungen, die Sie im Test formuliert haben, wirft

Eine Besonderheit von Supertest ist die Bindung an einen freien Port, falls noch nicht geschehen.

Supertest eine Exception aus und der Test wird mit einer entsprechenden Fehlermeldung abgebrochen.

Die Methoden von Supertest unterstützen Promises, also können Sie auch im Test die *async/await*-Schlüsselwörter verwenden, um auf die Antwort des Servers zu warten und weitere Überprüfungen durchzuführen. Über das *Response*-Objekt können Sie ebenfalls auf alle Aspekte der Antwort des Servers zugreifen und so überprüfen, ob die Antwort eine bestimmte Form aufweist. Der letzte *expect*-Aufruf stammt nicht von Supertest, sondern von Jest. Hier übergeben Sie den Body der Antwort und überprüfen mit dem *toEqual-*

Listing 1: Initialer Servercode

```
const express = require('express');

const app = express();

app.get('/user', (req, res) => {
  res.json({ id: 1, name: 'Klaus'});
});

app.listen(8080);

module.exports = app;
```

Listing 2: Test der „/user"-Route

```
const request = require('supertest');
const app = require('./index');

describe('/user', () => {
  it('should return user infos', async () => {
    const response = await request(app)
      .get('/user')
      .expect(200)
      .expect('Content-Type', 'application/json;&nbspcharset=utf-8');

    expect(response.body).toEqual({ id: 1, name: 'Klaus' });
  });
});
```

Als etwas schwieriger erweist es sich, Fehlerfälle in einer Schnittstelle zu simulieren.

Matcher, ob das angegebene Objekt vom Server zurückgegeben wurde.

Fehlerfälle testen

So, wie Sie den Erfolgsfall überprüfen können, können Sie auch Fehlerfälle überprüfen. Greifen Sie beispielsweise auf den nicht implementierten Endpunkt */news* zu, gibt der Server automatisch den Statuscode 404 „Not found" zurück. Diesen Fall können Sie mit dem in Listing 3 dargestellten Test überprüfen.

Etwas schwieriger wird die Situation, wenn es darum geht, Fehlerfälle in einer Schnittstelle zu simulieren, also beispielsweise zu testen, was passiert, wenn die Verbindung zur Datenbank nicht einwandfrei funktioniert. In diesem Fall können Sie sich die Tatsache zunutze machen, dass Supertest und Jest im gleichen Prozess laufen wie Ihre Applikation. Das ermöglicht es Ihnen, das Mock-Feature von Jest zu nutzen. Der Code in Listing 4 sorgt dafür, dass statt einer JSON-Antwort mit dem Statuscode 200 eine Antwort mit dem Code 500 gesendet wird, wenn beim Laden der Benutzer ein Fehler aufgetreten ist.

Um einen Fehler auszulösen, mocken Sie im Test das Modul, das Ihnen die *getUsers*-Funktion zur Verfügung stellt. Die Mock-Implementierung besteht dann aus einer Funktion, die als einziges Statement einen Fehler wirft. Den Test, der den Mock lädt, finden Sie in Listing 5.

Der Aufruf der *jest.mock*-Methode sorgt dafür, dass die Datei *user.js* aus dem Verzeichnis *__mocks__* statt der eigentlichen Implementierung geladen wird. Der Aufruf der *getUsers*-Funktion führt dann zu einer Exception, die in der Route gefangen und mit dem Statuscode 500 beantwortet wird. Damit läuft auch dieser Test erfolgreich durch.

Weitere Anwendungsfälle

Wie schon erwähnt, können Sie dank der Superagent-Bibliothek in einem Test mit Supertest auf alle Aspekte einer Anfrage und der zugehörigen Antwort zugreifen. Mit der *set*-Methode des Requests können Sie beispielsweise Headerfelder der Anfrage setzen. In Listing 2 haben Sie außerdem am Beispiel des Content-Type-Headers gesehen, wie Sie den Header einer Antwort überprüfen können.

Doch auch hier enden die Möglichkeiten von Supertest noch nicht. Sie können auch ausgefallenere Features wie beispielsweise Dateiuploads testen.

Zusammenfassung

Supertest ist eine vergleichsweise leichtgewichtige Bibliothek, die Sie bei der Formulierung von Schnittstellentests für eine Node-Applikation unterstützt. Sie verfügt über alle Hilfsmittel, die Sie für einen solchen Schnittstellentest benötigen und lässt Sie auf alle Elemente der Anfrage und der Antwort des Servers zugreifen. In Kombination mit einem der etablierten Test-Frameworks können Sie den API-Layer Ihrer Applikation mit überschaubarem Aufwand absichern.

Listing 3

```
describe('/news', () => {
  it('should&nbspnot&nbspexist', async () => {
    await request(app)
      .get('/news')
      .expect(404);
  });
});
```

Listing 4

```
app.get('/users', (req, res) => {
  try {
    res.json(getUsers());
  } catch (e) {
    res.sendStatus(500);
  }
});
```

Listing 5: Einsatz von Mocks

```
jest.mock('./user');

describe('/users', () => {
  it('should fail', async () => {
    await request(app)
      .get('/users')
      .expect(500);
  });
});
```

Sebastian Springer ist JavaScript-Entwickler bei MaibornWolff in München und beschäftigt sich vor allem mit der Architektur von client- und serverseitigem JavaScript. Sebastian ist Berater und Dozent für JavaScript und vermittelt sein Wissen regelmäßig auf nationalen und internationalen Konferenzen.