2 Frameworks

Im ersten Kapitel dieses Buchs wurden Ihnen die Grundlagen der testgetriebenen Entwicklung nahegebracht. Bevor Sie nun mit der testgetriebenen Entwicklung beginnen können, benötigen Sie noch eine funktionierende Arbeitsumgebung. Dieses Kapitel stellt Ihnen die verschiedenen Arten von Testframeworks vor und geleitet Sie Schritt für Schritt durch den Installations- und Konfigurationsprozess. Am Ende dieses Kapitels verfügen Sie über eine funktionsfähige Arbeitsumgebung, in die die verschiedenen Werkzeuge integriert sind, sodass Sie umgehend mit der Arbeit beginnen können.

Sie erhalten außerdem Tipps und Tricks aus der Praxis im Umgang mit diesen Werkzeugen.

2.1 Die Frameworks im Überblick

Die Grundlage testgetriebener Entwicklung sind Unit-Tests. Zur Erstellung dieser Tests sollten Sie auf ein bestehendes Framework zurückgreifen. Der Einsatz eines derartigen Frameworks nimmt Ihnen eine Vielzahl von Aufgaben ab, mit denen Sie im Umgang mit Unit-Tests täglich konfrontiert werden.

Zahlreiche Testframeworks

Im Gegensatz zu anderen Sprachen wie beispielsweise Java oder PHP, bei denen es mit JUnit und PHPUnit quasi Standard-Frameworks zur Erstellung von Unit-Tests gibt, existiert in JavaScript eine große Vielfalt von Testframeworks. Nahezu jedes Framework in JavaScript bringt sein eigenes Testframework mit. So ist QUnit beispielsweise das Testframework hinter jQuery, doch wird zur Absicherung der Qualität des Dojo-Toolkits verwendet und Siesta kommt beim Testen von ExtJS zum Einsatz.

Bei den verfügbaren Testframeworks lassen sich generell mehrere Arten unterscheiden. Es existieren Frameworks, bei denen Sie die Tests mithilfe einer HTML-Datei direkt im Browser ausführen können.

Demgegenüber stehen Frameworks, die Ihnen eine komplette Infrastruktur zur Verfügung stellen.

Im Verlauf dieses Kapitels lernen Sie beide Arten von Frameworks und deren generelle Verwendung kennen. Den Anfang machen die clientseitigen Frameworks QUnit und Jasmine.

2.2 Clientseitige Frameworks

Ein clientseitiges Testframework bezeichnet in JavaScript ein Testframework, das nur im Browser ausgeführt wird und ganz ohne Serverkomponente auskommt. Das bedeutet, dass Ihnen eine HTML-Infrastruktur zur Verfügung gestellt wird, innerhalb derer Sie sowohl Ihren Quellcode als auch Ihre Tests laden. Die Tests werden anschließend auf Basis des Quellcodes ausgeführt und das Ergebnis grafisch aufbereitet dargestellt.

Im Zuge dieses Kapitels lernen Sie mit QUnit und Jasmine zwei der populärsten Vertreter der clientseitigen Testframeworks kennen.

2.3 QUnit

QUnit wurde im Zuge von jQuery entwickelt, da die Entwickler die Notwendigkeit eines Testframeworks sahen, um ihren eigenen Code abzusichern. Die enge Kopplung zwischen jQuery und QUnit wurde allerdings im Jahr 2009 durch die Entwickler aufgehoben, sodass Sie QUnit mittlerweile auch losgelöst von jQuery verwenden können. Seit 2008 hat das Projekt eine eigene Website, *http://qunitjs.com*, und kann von dort heruntergeladen werden.

Wie jQuery ist auch QUnit ein Open-Source-Framework, das auf GitHub weiterentwickelt wird. Den aktuellen Quellcode finden Sie unter *http://github.com/jquery/qunit*.

Installation

Da es sich bei QUnit um ein clientseitiges Testframework handelt, ist die Installation mit sehr wenig Aufwand verbunden. Die aktuelle stabile Version von QUnit finden Sie, wie bereits erwähnt, auf *http://qunitjs.com*. Auf der Startseite gibt es einen Download-Bereich. Hier können Sie sich die beiden Dateien herunterladen, die Sie zum Betrieb von QUnit benötigen. Bei der Verwendung von QUnit sind Sie nicht an ein bestimmtes System gebunden. Die einzige Voraussetzung ist das Vorhandensein eines JavaScript-fähigen Browsers.

Grundsätzlich benötigen Sie die JavaScript-Datei, die den Quellcode von QUnit enthält. Ihr Name lautet `qunit-<major>.<minor>.<patch>.js`, wobei <major>, <minor> und <patch> für den jeweiligen Teil der Versionsnummer stehen. Der Name lautet nach diesem Schema beispielsweise qunit-1.12.0.js.

Neben dem Quellcode benötigen Sie außerdem ein Stylesheet, das für die Formatierung der Ergebnisse verantwortlich ist. Der Name dieser Datei lautet analog zur JavaScript-Datei `qunit-<major>.<minor>.<patch>.css`, also beispielsweise `qunit-1.12.0.js`.

Zusätzlich zu diesen beiden Dateien müssen Sie noch selbst eine Datei erstellen. Diese Datei stellt den Test-Runner dar. Er bindet sowohl den Quellcode als auch das Stylesheet von QUnit ein und stellt die Infrastruktur zur Ausgabe der Tests zur Verfügung. Außerdem binden Sie in dieser Datei Ihren eigenen JavaScript-Quellcode und Ihre Tests ein.

Listing 2–1
QUnit-Verzeichnisstruktur

```
.
├── TestRunner.html
├── js
│   ├── lib
│   │   └── qunit-1.12.0.js
│   ├── src
│   │   └── source.js
│   └── test
│       └── test.js
└── style
    └── qunit-1.12.0.css
```

Listing 2–1 gibt Ihnen einen Überblick über eine mögliche Verzeichnisstruktur, die Sie für ein Projekt mit QUnit verwenden können. Auf der obersten Verzeichnisebene liegt die Datei `TestRunner.html`, die für die Ausführung der Tests verantwortlich ist. Auf gleicher Ebene liegt das `style`-Verzeichnis, das das QUnit-Stylesheet enthält. Außerdem existiert ein `js`-Verzeichnis, in dem sämtliche JavaScript-Dateien gespeichert werden. Dieses enthält im Unterverzeichnis `lib` den Quellcode von QUnit, im Verzeichnis `src` liegt der eigentliche Quellcode Ihrer Applikation und im `test`-Verzeichnis befinden sich schließlich die zugehörigen Tests.

Im ersten Schritt müssen Sie sich um den Test-Runner kümmern. In Listing 2–2 finden Sie den dafür notwendigen Quellcode.

Listing 2–2
QUnit – TestRunner.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>QUnit Example</title>
        <link rel="stylesheet" href="style/qunit-1.12.0.css">
    </head>
    <body>
        <div id="qunit"></div>
        <div id="qunit-fixture"></div>
        <script src="js/lib/qunit-1.12.0.js"></script>
        <script src="js/src/source.js"></script>
        <script src="js/test/test.js"></script>
    </body>
</html>
```

Die `TestRunner.html`-Datei bindet die einzelnen Komponenten ein, die erforderlich sind, damit Ihre Unit-Tests ablaufen können. Die wichtigsten dieser Komponenten sind Ihr Quellcode und die Tests für diesen. In Listing 2–3 sehen Sie den Inhalt der Datei `source.js` im Verzeichnis `js/src`.

Listing 2–3
QUnit – source.js, Datei mit Quelltext

```
function add (a, b) {
    return a + b;
}
```

Die Datei `test.js` im Verzeichnis `js/test` enthält schließlich Ihre Tests. In Listing 2–4 sehen Sie den entsprechenden Quellcode.

Listing 2–4
QUnit – test.js, Datei mit Tests

```
test('Addition of 1 and 1', function () {
    equal(add(1, 1), 2, '1 and 1 equals 2');
});

test('Addition of 1 and 3', function () {
    equal(add(1, 3), 5, '1 and 3 equals 5');
});
```

Mit diesen Dateien haben Sie alle Voraussetzungen erfüllt, die notwendig sind, um Ihre Applikation zu testen. Sie können nun die `TestRunner.html`-Datei in Ihrem Browser öffnen und damit die Tests ausführen. Abbildung 2–1 zeigt Ihnen das Ergebnis der Tests. Besonders hervorgehoben sind dabei fehlschlagende Tests, da hier akuter Handlungsbedarf zur Fehlerbehebung besteht.

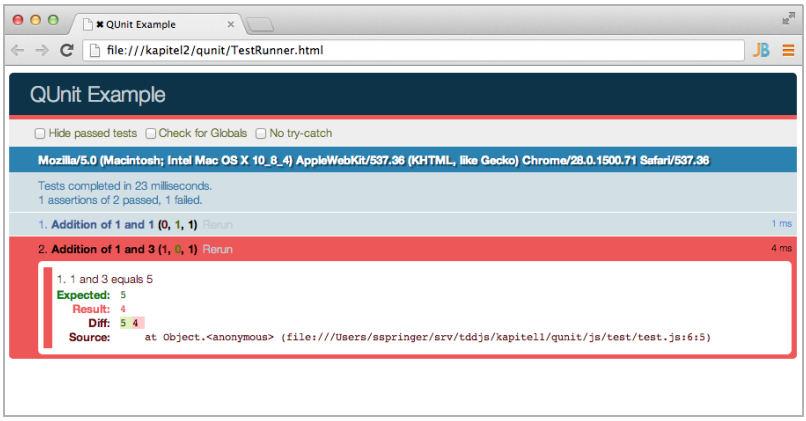

Abb. 2–1
Ausführung der QUnit-Tests

Im folgenden Abschnitt lernen Sie den Aufbau von Tests mit QUnit kennen.

Tests mit QUnit

Wie Sie bereits in Listing 2–4 gesehen haben, beginnt ein Test generell mit einem Aufruf der `test`-Funktion, die Ihnen QUnit zur Verfügung stellt. Das erste Argument ist eine Zeichenkette, die beschreibt, was der Test macht. Diese Zeichenkette wird von QUnit in der Zusammenfassung der Ergebnisse als Repräsentation des Tests angezeigt. Sie sollten hier also einen relativ kurzen und beschreibenden Text verwenden. Das zweite Argument ist eine Callback-Funktion. Sie enthält die eigentliche Logik des Tests. Ein Test besteht in QUnit generell aus einem Aufruf einer sogenannten Assertion. Mit den Assertions prüfen Sie, ob sich Ihr Quellcode so verhält, wie Sie es erwarten. Neben einer Assertion kann ein Test außerdem noch beliebigen Quellcode enthalten, der erforderlich ist, die Umgebung für die Assertion bereitzustellen.

Den Kern der Unit-Tests bilden allerdings die Assertions. In Tabelle 2–1 sehen Sie die verschiedenen Assertions, die Ihnen QUnit zur Verfügung stellt.

Tab. 2–1
QUnit – Assertions

Assertion	Bedeutung
`ok`	Prüfung auf den Boolean-Wert true
`equal`	Nichtstrikte Prüfung auf Gleichheit
`deepEqual`	Rekursive Prüfung auf Gleichheit für beispielsweise Arrays oder Objekte
`strictEqual`	Strikte Prüfung auf Gleichheit
`notEqual`	Negierung der nichtstrikten Prüfung auf Gleichheit
`notDeepEqual`	Negierung der rekursiven Prüfung auf Gleichheit
`notStrictEqual`	Negierung der nichtstrikten Prüfung auf Gleichheit
`Throws`	Prüfung, ob eine Funktion eine Exception wirft

Neben den reinen Tests stehen Ihnen in QUnit zusätzliche Möglichkeiten zur Verfügung. So können Sie Ihre Tests beispielsweise mit der `module`-Funktion logisch gruppieren.

Listing 2–5
QUnit-Gruppierung

```
module('Addition Test', {
    setup: function () {console.log('Setup');},
    teardown:function () {console.log('Teardown');}
});
test('Addition of 1 and 1', function () {
    equal(add(1, 1), 2, '1 and 1 equals 2');
});
```

Die Gruppierung, die Sie in Listing 2–5 mit dem Aufruf der `module`-Funktion erreichen, resultiert in der Ergebnisübersicht der Tests darin, dass die Zeichenkette, die Sie als erstes Argument an die `module`-Funktion übergeben, den entsprechenden Tests vorangestellt wird. Eine Gruppierung reicht dabei von einem Aufruf der `module`-Funktion bis zum nächsten Aufruf dieser Funktion.

Ein weiteres Feature, das Sie im Zuge der Gruppierung erhalten, ist die `setup`- und `teardown`-Funktionalität. Diese beiden Funktionen geben Sie in einem Objektliteral als zweites Argument beim Aufruf der `module`-Funktion an. Dabei wird die `setup`-Funktion vor jedem Test des Moduls ausgeführt. Sie können mit dieser Funktion die Umgebung für den Test vorbereiten und beispielsweise Objekte erstellen oder Ähnliches. Die `teardown`-Funktion wird im Gegensatz zur `setup`-Funktion nach jedem Test des Moduls ausgeführt. Dies ist die Stelle, an der Sie den Quellcode ablegen, der dafür sorgt, dass die Umgebung nach einem Test wieder aufgeräumt wird.

Ein weiteres, weit verbreitetes Testframework ist neben QUnit das unabhängige Framework Jasmine.

2.4 Jasmine

Jasmine wurde, im Gegensatz zu QUnit, nicht im Zuge eines JavaScript-Frameworks entwickelt, sondern gänzlich unabhängig. Eine weitere Besonderheit von Jasmine ist, dass es sich bei diesem Framework nicht um ein traditionelles Unit-Test-Framework handelt, sondern um ein Behaviour Driven Development Framework. Sie können Jasmine allerdings auch für herkömmliche Unit-Tests verwenden. In der Praxis ergibt sich der einzige Unterschied zwischen einem traditionellen Unit-Test-Framework und einem Behaviour Driven Development Framework in der Erstellung der Tests in einer veränderten Syntax.

Jasmine erfreut sich aktuell aufgrund seiner umfangreichen Features und seiner angenehmen Syntax einer großen Verbreitung. Im Verlauf dieses Buchs werden die meisten Beispiele in Jasmine erstellt. Die Unterschiede zu anderen Frameworks sind meist so gering, dass Sie mit einigen Umbauten die Beispiele auch mit anderen Frameworks nachvollziehen können.

Installation

Jasmine ist, wie auch schon QUnit, ein Open-Source-Projekt. Das bedeutet, dass Sie es kostenlos verwenden und in Ihr Projekt einbinden können. Die Entwickler von Jasmine setzen vollständig auf die Infrastruktur von GitHub. Die Seite des Projekts können Sie unter *http://pivotal.github.io/jasmine/* erreichen. Hier finden Sie alle notwendigen Ressourcen für einen schnellen Einstieg in Jasmine. Die Seite enthält sowohl eine Dokumentation zur Verwendung als auch Verweise zum Download von Jasmine.

Einen sehr einfachen Einstieg in Jasmine bietet das Stand-alone-Release von Jasmine. Dieses Release besteht aus einer ZIP-Datei, die alle Komponenten enthält, die Sie zum Start benötigen. Damit Sie Jasmine verwenden können, müssen Sie lediglich die Datei von *https://github.com/jasmine/jasmine/tree/master/dist* herunterladen und in ein Verzeichnis Ihrer Wahl entpacken. Der Name der Datei lautet ähnlich wie bei QUnit schon `jasmine-standalone-<major>.<minor>.<patch>.zip`. Nach dem Entpacken erhalten Sie eine Verzeichnisstruktur, wie Sie sie in Listing 2–6 finden.

Listing 2–6
Jasmine – Verzeichnisstruktur

```
.
├── MIT.LICENSE
├── SpecRunner.html
├── lib
│   └── jasmine-2.1.3
│       ├── boot.js
│       ├── console.js
│       ├── jasmine-html.js
│       ├── jasmine.css
│       ├── jasmine.js
│       └── jasmine_favicon.png
├── spec
│   ├── PlayerSpec.js
│   └── SpecHelper.js
└── src
    ├── Player.js
    └── Song.js
```

Die Struktur ähnelt der, die Sie schon bei QUnit kennengelernt haben. Die Basis bildet die Datei `SpecRunner.html`. Öffnen Sie diese Datei in Ihrem Browser, werden Ihre Tests ausgeführt. Zu diesem Zweck binden Sie in dieser Datei neben Jasmine Ihren eigenen Quellcode und Ihre Tests ein. Das Beispiel, das mit dem Jasmine Stand-alone-Release mitgeliefert wird, enthält mit den Dateien `Player.js` und `Song.js` bereits zwei Dateien mit Quellcode, die im Verzeichnis `src` liegen. Die Tests befinden sich im Verzeichnis `spec` in der Datei `PlayerSpec.js`.

Wahrscheinlich ist Ihnen bereits aufgefallen, dass in Jasmine nicht die Rede von Tests, sondern von Specs ist. Hier macht sich die Herkunft von Jasmine bemerkbar. In vielen Frameworks für Behaviour Driven Development ist von Specs oder Spezifikationen anstatt von Tests die Rede.

Damit Sie die Unterschiede zwischen QUnit und Jasmine deutlicher sehen, passen Sie im Folgenden das Beispiel aus dem Stand-alone-Release von Jasmine so an, dass es den gleichen Zweck erfüllt wie bereits das Beispiel von QUnit. Die Datei `SpecRunner.html` müssen Sie im Gegensatz zu QUnit nur an zwei Stellen anpassen; nämlich dort, wo Sie Ihren Quellcode und Ihre Tests einbinden. In Listing 2–7 sehen Sie den entsprechenden Ausschnitt der Datei.

Listing 2–7
Jasmine – Anpassung der SpecRunner.html-Datei

```
<!-- include source files here… -->
<script type="text/javascript" src="src/source.js"></script>

<!-- include spec files here… -->
<script type="text/javascript" src="spec/test.js"></script>
```

Wenn Sie sich die Datei `SpecRunner.html` genauer ansehen, werden Sie sehen, dass hier noch einiger JavaScript-Quellcode enthalten ist, der

dafür sorgt, dass die Tests ausgeführt werden und ein entsprechender Bericht angezeigt wird. Diesen Quellcode müssen Sie allerdings nicht ändern, da dieser ohne Anpassungen funktioniert.

Im nächsten Schritt müssen Sie die Datei anlegen, die Ihren Quellcode enthält. Diese trägt den Namen `source.js` und liegt im Verzeichnis `src`. Der Inhalt dieser Datei gleicht dem aus dem Beispiel von QUnit und ist in Listing 2–8 zu finden.

Listing 2–8
Jasmine – source.js, Datei mit Quelltext

```
function add (a, b) {
    return a + b;
}
```

Diese Funktion soll nun mit Jasmine getestet werden. Legen Sie hierfür eine neue Datei mit dem Namen `test.js` im Verzeichnis `spec` an. Listing 2–9 zeigt Ihnen den Quellcode der Datei.

Listing 2–9
Jasmine – test.js, Datei mit Tests

```
describe("add", function () {
    it ('Addition of 1 and 1', function () {
        expect(add(1, 1)).toEqual(2);
    });

    it('Addition of 1 and 3', function () {
        expect(add(1, 3)).toEqual(5);
    });
});
```

Mit dieser Infrastruktur können Sie nun Ihre Tests ausführen. In Abbildung 2–2 sehen Sie das Ergebnis des Testlaufs.

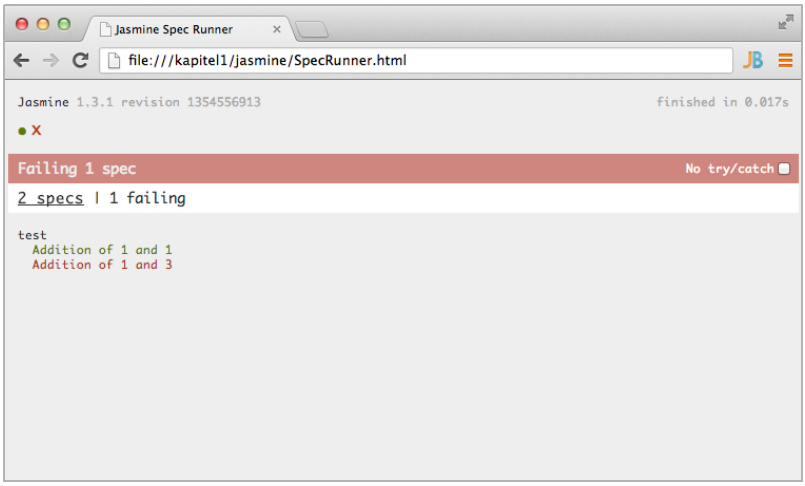

Abb. 2–2
Ausführung von Jasmine-Tests

Der nächste Abschnitt beschreibt, wie Sie Ihre Tests mit Jasmine erstellen.

Tests mit Jasmine

Die Formulierung der Tests in Jasmine unterscheidet sich teilweise erheblich von der in QUnit. Wo Sie in QUnit die Option haben, Ihre Tests zu gruppieren, ist die Gruppierung in Jasmine verpflichtend. Der Aufruf der `describe`-Funktion leitet eine Gruppe von Tests ein. Das erste Argument dieses Funktionsaufrufs ist eine Zeichenkette, die das zu testende Objekt beschreibt. Diese Zeichenkette wird im Bericht über die Ergebnisse der Tests zur Gruppierung verwendet. Das zweite Argument, das Sie der `describe`-Funktion beim Aufruf mitgeben, ist eine Callback-Funktion. Sie enthält die Tests dieser Gruppe.

Innerhalb der Callback-Funktion der `describe`-Funktion definieren Sie die einzelnen Tests mithilfe der `it`-Funktion. Ein Test besteht wiederum aus einem Aufruf der `it`-Funktion. Diesem Aufruf übergeben Sie als erstes Argument einen beschreibenden Text. Dieser Text wird im Bericht als Repräsentation des Tests angezeigt. Das zweite Argument, das Sie an die `it`-Funktion übergeben, ist eine Callback-Funktion. Die Callback-Funktion enthält die eigentliche Testlogik.

Der Kern eines Tests, der mit Jasmine formuliert ist, ist wie bei anderen Testframeworks auch die Assertion; nur dass die Assertions in Jasmine aus mehreren Teilen bestehen. Im ersten Schritt rufen Sie die `expect`-Funktion mit dem zu testenden Wert als Argument auf. Das von der `expect`-Funktion zurückgegebene Objekt stellt Ihnen dann die sogenannten *Matcher* zur Verfügung. Ein Matcher ist mit den Assertions in QUnit vergleichbar. In Tabelle 2–2 sehen Sie eine Auswahl von Matchern.

Tab. 2–2
Jasmine-Matcher

Matcher	Bedeutung
`toBe`	Typsichere Prüfung auf Gleichheit
`toEqual`	Erweiterte Prüfung auf Gleichheit anhand einer internen Funktion
`toMatch`	Prüfung gegen einen regulären Ausdruck
`toBeDefined`	Prüft, ob ein Ausdruck definiert ist
`toBeTruthy`	Prüft, ob ein Ausdruck wahr ist, nicht typsicher
`toBeFalsy`	Prüft, ob ein Ausdruck falsch ist, nicht typsicher

Neben den hier vorgestellten Matchern verfügt Jasmine über zahlreiche weitere Matcher. Sollte innerhalb Ihrer Applikation allerdings ein Spezialfall existieren, der sich mit den bestehenden Matchern nicht oder nur sehr umständlich abdecken lässt, bietet Ihnen Jasmine die Möglichkeit, eigene Matcher zu erstellen und diese einzubinden. Wei-

tere Informationen zu diesem Thema finden Sie in der Online-Dokumentation unter *http://pivotal.github.io/jasmine/*.

Wie Sie in Listing 2–9 bereits sehen konnten, müssen Sie mit Jasmine Ihre Tests strukturieren. Neben der reinen Strukturierung stellt Ihnen die `describe`-Funktion mit den Funktionen `beforeEach` und `afterEach` ein Feature zur Verfügung, das wie die `setup`- und `teardown`-Funktionen von QUnit wirkt. Mit `beforeEach` bereiten Sie die Umgebung für Ihre Tests vor und mit `afterEach` räumen Sie die Umgebung auf. Listing 2–10 zeigt Ihnen den Einsatz dieser beiden Funktionen.

Listing 2–10
Jasmine – beforeEach und afterEach

```
describe("add", function () {
    beforeEach(function () {console.log('Setup');});
    afterEach(function () {console.log('Teardown');});

    it ('Addition of 1 and 1', function () {
        expect(add(1, 1)).toEqual(2);
    });
});
```

Die Aufrufe von `beforeEach` und `afterEach` müssen jeweils innerhalb eines `describe`-Blocks stattfinden. Dabei ist es allerdings nicht relevant, an welcher Stelle Sie beide Funktionen aufrufen. Jasmine sorgt intern dafür, dass die Callback-Funktionen jeweils vor beziehungsweise nach den Tests ausgeführt werden.

Sowohl dem Aufruf von `beforeEach` als auch dem von `afterEach` übergeben Sie eine Callback-Funktion als einziges Argument. Diese wird vor beziehungsweise nach jedem Test im aktuellen `describe`-Block ausgeführt.

Um Ihre Tests noch genauer steuern zu können, erlaubt Ihnen Jasmine, Aufrufe von `describe` zu verschachteln. Das bedeutet, dass Sie innerhalb der Callback-Funktion einer `describe`-Funktion ein weiteres Mal `describe` aufrufen und so eine Gruppe von Tests in einer anderen Gruppe erstellen können. Für die `beforeEach`- und `afterEach`-Funktionen bedeutet das, dass zuerst die äußere `beforeEach`-Funktion und dann die innere Funktion aufgerufen wird. Bei `afterEach` ist der Fall genau umgekehrt. Zuerst wird die innere `afterEach`-Funktion ausgeführt, danach die äußere.

2.5 Nachteile clientseitiger Frameworks

Die Ausführung der Tests in einem Browser, wie Sie es hier am Beispiel von Jasmine und QUnit gesehen haben, birgt einige entscheidende Nachteile für den Entwicklungsprozess.

Wechsel zwischen Entwicklungsumgebung und Browser

Negativ wirkt sich vor allem der Wechsel von der Entwicklungsumgebung in den Browser aus. Sie müssen häufig zwischen verschiedenen Werkzeugen hin und her wechseln. Sie schreiben einen Test in Ihrer Entwicklungsumgebung, führen ihn im Browser aus, stellen ein Problem fest, wechseln zurück zur Entwicklungsumgebung, beheben das Problem und müssen anschließend wieder in den Browser wechseln, um den Test erneut auszuführen. Die testgetriebene Entwicklung lebt von schnellem Feedback und einer sehr häufigen Ausführung der Tests. Dies wird durch den ständigen Wechsel zwischen Entwicklungsumgebung und Browser erheblich erschwert.

Nur ein Browser

Ein weiterer Nachteil der Ausführung der Tests im Browser ist, dass Sie die Tests in genau einem Browser ausführen. Ihr Ziel sollte allerdings sein, die Tests in möglichst vielen Browsern auszuführen.

Für diese beiden Problemstellungen existieren zahlreiche Lösungen, die es Ihnen erlauben, Ihre Tests aus der Entwicklungsumgebung heraus auf mehreren Browsern auszuführen. Der Kern dieser Lösungsansätze besteht aus serverseitigen Testframeworks. In den nachfolgenden Abschnitten werden diese näher behandelt.

2.6 Serverseitige Frameworks

Ähnlich wie schon bei den clientseitigen Frameworks, gibt es auch bei den serverseitigen Lösungen nicht die eine Standardlösung, stattdessen existieren mehrere Implementierungen parallel, jede mit ihren eigenen Vor- und Nachteilen. In den nächsten Abschnitten lernen Sie mit JsTestDriver und Karma zwei Vertreter dieser Kategorie der Testframeworks kennen.

2.7 JsTestDriver

JsTestDriver ist ein serverseitiges Testframework, das schon seit mehreren Jahren existiert und in der Praxis erprobt ist. Ein Seiteneffekt davon ist, dass es zahlreiche Erweiterungen für dieses Framework gibt und die Integration in andere Systeme wie beispielsweise die Entwicklungsumgebung oder Continuous-Integration-Systeme wie Jenkins problemlos möglich ist.

JsTestDriver wird als Open-Source-Projekt im Zuge von *http://code.google.com* entwickelt. Die Seite des Projekts finden Sie unter *https://code.google.com/p/js-test-driver/*. Bevor Sie mit dem Einsatz von JsTestDriver beginnen, sollten Sie einige Details über die Komponenten von JsTestDriver und deren Zusammenspiel wissen.

Funktionsweise

Im Gegensatz zu den clientseitigen Frameworks, die Sie bis jetzt kennengelernt und bei denen Sie Ihre Tests durch einen einfachen Seitenaufruf im Browser ausgeführt haben, bestehen die serverseitigen Testframeworks aus mehreren Komponenten. In Abbildung 2–3 sehen Sie eine schematische Übersicht der einzelnen Teile, die erforderlich sind, damit Sie Ihre Tests mit JsTestDriver ausführen können.

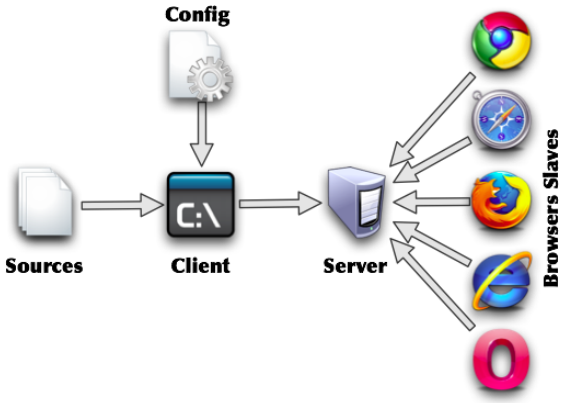

Abb. 2–3
Komponenten von JsTestDriver

Der Kern von JsTestDriver besteht aus der Serverkomponente. Diese ist in Java geschrieben und muss im Hintergrund ausgeführt werden, damit Ihre Tests ablaufen können. Sobald der Server gestartet ist, können Sie die Browser an den Server binden. Diese Browser werden dann dazu verwendet, die Tests auszuführen. Aufbauend auf dem Server müssen Sie für jeden Testlauf einen Clientprozess starten. Dieser basiert wiederum auf Java und enthält die Informationen, wo die Tests liegen und welche davon ausgeführt werden.

Kommunikation zwischen Browser und Server

Die Kommunikation zwischen Browser und Server findet über TCP statt. Das bedeutet, dass Sie nicht darauf beschränkt sind, nur lokale Browser für einen Testlauf zu verwenden, stattdessen können Sie sich von verschiedenen Maschinen mit dem Server verbinden. So ist es beispielsweise möglich, dass Sie den Server auf einem Linux-System ohne grafische Oberfläche ausführen und dann die Browser eines Windows-, eines Linux- und eines Mac-Systems mit dem Server verbinden und dort die Tests ausführen lassen.

Die Möglichkeiten der Testausführung enden jedoch nicht bei den Desktop-Betriebssystemen. Entwickeln Sie beispielsweise eine mobile Webapplikation, können Sie auch die Browser von mobilen Geräten mit dem Server verbinden und dort dann Ihre Tests ausführen.

Der nächste Abschnitt beschäftigt sich nun damit, wie Sie Ihre Infrastruktur aufsetzen können.

Installation

Für den Betrieb von JsTestDriver benötigen Sie Java. Ansonsten müssen Sie keine weiteren Voraussetzungen erfüllen. Damit Sie dieses Framework verwenden können, ist das Java-Archiv, das den JsTestDriver beinhaltet, zwingend erforderlich. Dieses Archiv erhalten Sie auf der Download-Seite des JsTestDriver-Projekts unter *https://code.google.com/p/js-test-driver/downloads/list*. Hier haben Sie die Auswahl aus verschiedenen Paketen. Im Normalfall sollten Sie das Paket mit den enthaltenen Abhängigkeiten herunterladen. Dieses können Sie problemlos ausführen und müssen sich nicht um die Auflösung der Abhängigkeiten kümmern. Der Name der Datei lautet `JsTestDriver-<major>.<minor>.<patch>.jar`. Aktuell ist die Version 1.3.5 verfügbar.

Wie schon bei QUnit und Jasmine, sollten Sie auch bei der Verwendung von JsTestDriver Ihr Projekt strukturieren. In Listing 2–11 finden Sie ein Beispiel für eine solche Struktur in einem Projekt.

Listing 2–11
JsTestDriver – Verzeichnisstruktur

```
.
├── bin
│   └── JsTestDriver-1.3.5.jar
├── jsTestDriver.conf
├── src
│   └── source.js
└── src-test
    └── test.js
```

Im `bin`-Verzeichnis der Verzeichnisstruktur findet sich das Java-Archiv mit dem JsTestDriver. Dieses benötigen Sie, um sowohl den Server als auch den Client zu starten und damit Ihre Tests auszuführen. Die Datei mit dem Namen `jsTestDriver.conf` enthält die Konfiguration für Ihre Tests. Listing 2–12 enthält den Inhalt dieser Konfigurationsdatei.

Listing 2–12
JsTestDriver – jsTestDriver.conf

```
server: http://localhost:9876

load:
  - src/*.js
  - src-test/*.js
```

Das Format der Konfigurationsdatei ist YAML. Über verschiedene Direktiven übergeben Sie dem Test-Runner verschiedene Informationen, die zur Ausführung der Tests notwendig sind. Mit der `server`-Angabe spezifizieren Sie, wo der JsTestDriver-Server läuft. Mit dieser Information weiß der Test-Runner, wo er den Server finden kann. Alternativ zur Konfigurationsdatei können Sie den Server auch über die Kommandozeile angeben. Mithilfe von `load` definieren Sie, welche Dateien für die Tests geladen werden sollen. In diesem Fall werden sämtliche Dateien der Verzeichnisse `src` und `src-test`, die auf `.js` enden, geladen. Das Verzeichnis src beherbergt den Quellcode Ihrer

Applikation. In diesem Beispiel wird der gleiche Quellcode wie auch schon für QUnit und Jasmine verwendet. In Listing 2–13 sehen Sie den entsprechenden Quellcode.

```
function add (a, b) {
    return a + b;
}
```

Listing 2–13
JsTestDriver – source.js, Datei mit Quelltext

Das `src-test`-Verzeichnis enthält schließlich die Tests Ihrer Applikation. JsTestDriver bietet Ihnen neben der bereits vorgestellten Infrastruktur auch ein eigenes Testframework, mit dem Sie Ihre Tests formulieren können. Listing 2–14 stellt Ihnen zwei Tests für das Beispiel der `add`-Funktion vor. Die Tests liegen in der Datei `test.js` im `src-test`-Verzeichnis.

```
AdditionTest = TestCase("AdditionTest");

AdditionTest.prototype.testAdd1 = function() {
    assertEquals(2, add(1, 1));
};

AdditionTest.prototype.testAdd2 = function() {
    assertEquals(2, add(1, 2));
};
```

Listing 2–14
JsTestDriver – test.js, Datei mit Tests

Der nächste Schritt zum Betrieb von JsTestDriver besteht darin, dass Sie den Server auf der Kommandozeile starten. In Listing 2–15 sehen Sie das dafür erforderliche Kommando.

```
$ java -jar JsTestDriver-1.3.5.jar --port 9876
setting runnermode QUIET
```

Listing 2–15
Starten des JsTestDriver-Servers

Als einzige Option müssen Sie dem Aufruf von JsTestDriver die TCP-Portnummer übergeben, auf die Sie den Server binden möchten. Über diesen Port verbinden Sie zu einem späteren Zeitpunkt Ihre Browser mit dem Server. Der Standardwert für die Portnummer ist 9876. Sie können diesen Wert allerdings beliebig variieren. Allerdings sollten Sie beachten, dass Sie für Nummern unter 1024 auf Unix-Systemen Administratorberechtigungen benötigen.

Haben Sie den Server gestartet, können Sie Browser mit diesem Server verbinden. Dies erreichen Sie entweder dadurch, dass Sie im Browser die URL *http://localhost:9876* aufrufen und anschließend auf den Link `Capture This Browser` klicken oder Sie verwenden direkt die URL *http://localhost:9876/capture*. Abbildung 2–4 zeigt Ihnen, wie das Ergebnis einer erfolgreichen Verbindung mit dem Server im Browser aussieht.

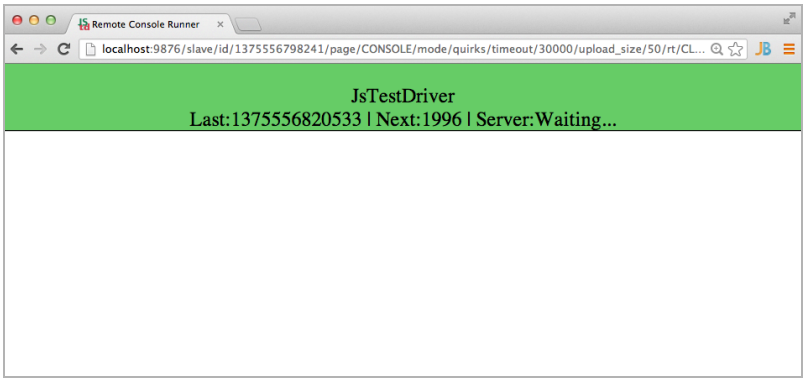

Abb. 2–4 *Verbindung eines Browsers mit dem JsTestDriver-Server*

Haben Sie Ihre Browser mit dem Server verbunden, können Sie Ihre Tests im Anschluss ausführen. Zu diesem Zweck müssen Sie erneut auf die Kommandozeile wechseln und zusätzlich zum bereits laufenden JsTestDriver-Server den JsTestDriver-Client starten. Den Client starten Sie, wie schon den Server, mit einem Aufruf des Java-Archivs. Client und Server unterscheiden sich lediglich in der Art des Aufrufs. Listing 2–16 zeigt Ihnen, wie Sie den JsTestDriver-Client aufrufen müssen, damit Ihre Tests ausgeführt werden. Beim Aufruf müssen Sie außerdem beachten, dass Sie das Kommando im Verzeichnis mit der Konfigurationsdatei absetzen, da der Client ansonsten keine Informationen über den Server hat.

Listing 2–16 *Starten des JsTestDriver-Clients*

```
$ java -jar bin/JsTestDriver-1.3.5.jar --tests all
setting runnermode QUIET
.F
Total 2 tests (Passed: 1; Fails: 1; Errors: 0) (3.00 ms)
  Chrome 28.0.1500.95 Mac OS: Run 2 tests (Passed: 1; Fails: 1;
Errors 0) (3.00 ms)
    AdditionTest.testAdd2 failed (1.00 ms): AssertError: expected
2 but was 3
      Error: expected 2 but was 3
          at AdditionTest.testAdd2
(http://localhost:9876/test/src-test/test.js:8:5)
```

Die Ausgabe der Testergebnisse erhalten Sie auf der Kommandozeile. Sie sehen in diesem Beispiel das Resultat für einen erfolgreichen und einen fehlschlagenden Test. Neben den Informationen, wie viele Tests ausgeführt wurden und wie viele davon erfolgreich waren, sehen Sie außerdem, wie lange die Testausführung in Anspruch nahm und auf welchen Browsern und Systemen die Tests ausgeführt wurden. Im nächsten Abschnitt erhalten Sie einen Einblick in das Testframework von JsTestDriver, mit dem Sie Ihre Tests formulieren.

Tests mit JsTestDriver

JsTestDriver bietet Ihnen, wie schon QUnit und Jasmine, ein eigenständiges Testframework, mit dem Sie Ihre Tests schreiben. Den Quellcode eines typischen JsTestDriver-Tests haben Sie bereits in Listing 2–14 gesehen. Die Grundlage eines Tests bildet ein Aufruf der `TestCase`-Funktion. Dieser Funktion übergeben Sie beim Aufruf den Namen der Testgruppe, die Sie erstellen möchten. Im Falle des Beispiels in Listing 2–14 ist dies die Zeichenkette `AdditionTest`. Diese Zeichenkette können Sie später beim Aufruf des Clients dazu verwenden, nur eine bestimmte Gruppe von Tests aufzurufen. Hierfür geben Sie dann statt der Option `--tests all` die Option `--tests AdditionTest` an und es werden lediglich die Tests dieser Gruppe ausgeführt.

Vom Aufruf der `TestCase`-Funktion erhalten Sie ein Objekt zurück, in dessen `prototype`-Eigenschaft Sie dann Ihre eigenen Tests formulieren. Wichtig ist hier, dass sämtliche Testmethoden mit dem Präfix `test` beginnen müssen, da der JsTestDriver sie ansonsten nicht ausführt.

Ein wichtiger Bestandteil des Testframeworks von JsTestDriver sind die Assertions. Mit diesen formulieren Sie Ihre Erwartung an einen bestimmten Ausdruck. Stimmt der Ausdruck mit Ihrer Erwartung überein, gilt der Test als erfolgreich, ist dies nicht gegeben, schlägt der Test fehl. Tabelle 2–3 enthält eine Auswahl der verfügbaren Assertions.

Assertion	Bedeutung
`assertEquals`	Nicht typsichere Prüfung zweier Werte auf deren Gleichheit
`assertSame`	Typsichere Prüfung zweier Werte auf deren Gleichheit
`assertMatch`	Prüfung gegen einen regulären Ausdruck
`assertUndefined`	Prüft, ob ein Ausdruck undefiniert ist.
`assertTrue`	Prüft, ob ein Ausdruck wahr ist, nicht typsicher
`assertFalse`	Prüft, ob ein Ausdruck falsch ist, nicht typsicher

Tab. 2–3
Assertions in JsTestDriver

Wie in den anderen Testframeworks, die Sie bisher kennengelernt haben, existiert auch in JsTestDriver die Möglichkeit, Setup- und Teardown-Routinen zu definieren, die vor beziehungsweise nach jedem Test einer Gruppe ausgeführt werden. In Listing 2–17 sehen Sie, wie Sie diese beiden Komponenten in Ihre Tests einbauen.

Listing 2–17
JsTestDriver – setUp und tearDown

```
AdditionTest = TestCase("AdditionTest");

AdditionTest.prototype.setUp = function () {
    console.log('Ausführung vor jedem Test');
};

AdditionTest.prototype.tearDown = function () {
    console.log('Ausführung nach jedem Test');
};

AdditionTest.prototype.testAdd1 = function() {
    assertEquals(2, add(1, 1));
};
```

Führen Sie den Test aus Listing 2–17 aus, werden auf der Konsole der Browser, die mit dem Server verbunden sind, die beiden Zeichenketten ausgegeben. Normalerweise setzen Sie die `setUp`-Methode ein, um die Umgebung für einen Test vorzubereiten, und die `tearDown`-Methode, um nach dem Test wieder aufzuräumen.

Die Features von JsTestDriver, die Sie bisher kennengelernt haben, stellen nur einen Bruchteil des gesamten Funktionsumfangs dar. JsTestDriver stellt Ihnen die Infrastruktur allerdings nicht nur für das eigene Testframework zur Verfügung, sondern kann auch mit QUnit oder Jasmine verwendet werden und gleicht dadurch deren Nachteile als clientseitige Testframeworks, die nur in einem Browser eingesetzt werden können, aus.

Adapter

QUnit und Jasmine sind clientseitige Testframeworks und werden als solche direkt im Browser ausgeführt. Es existiert allerdings die Möglichkeit, dass Sie Ihre Tests über die Infrastruktur von JsTestDriver ausführen lassen. Zu diesem Zweck gibt es sowohl für QUnit als auch für Jasmine Adapter, die die jeweiligen Tests in die Syntax von JsTestDriver übersetzen und Sie somit in die Lage versetzen, Ihre Tests auf verschiedenen Browsern von der Kommandozeile aus zu starten. Weiterführende Informationen zu den JsTestDriver-Adaptern finden Sie unter *https://code.google.com/p/js-test-driver/wiki/XUnitCompatibility*.

Ein weiteres serverseitiges Testframework stellt Karma dar. Dieses, im Vergleich zu JsTestDriver, noch relativ junge Framework lernen Sie in den folgenden Abschnitten kennen.

2.8 Karma

Karma ist ein Open-Source-Projekt, das zu einem früheren Zeitpunkt Testacular genannt wurde. Karma baut auf dem Prinzip von JsTestDriver auf. Das Ziel dieses Projekts ist allerdings, sich rein auf die Infrastruktur-Komponente zu konzentrieren und die Tests in anderen Frameworks wie QUnit oder Jasmine zu schreiben. Das bedeutet, dass mit diesem Projekt der Gedanke der JsTestDriver-Adapter konsequent weitergeführt wurde.

Karma ist im Gegensatz zu JsTestDriver nicht in Java, sondern in JavaScript geschrieben und läuft auf Basis von Node.js, was die Installation erheblich vereinfacht.

Installation

Die Voraussetzung zum Betrieb von Karma ist, dass Sie Node.js auf Ihrem System installiert haben. Nähere Informationen hierzu finden Sie unter *http://nodejs.org/*. Haben Sie Node.js erst einmal auf Ihrem System installiert, können Sie Karma über den Node.js-Paketmanager NPM installieren. Dieser sorgt für die Auflösung der Abhängigkeiten. Listing 2–18 enthält das Kommando, das zur Installation von Karma notwendig ist.

```
$ npm install –g karma
```

Listing 2–18
Installation von Karma

Wie auch schon der JsTestDriver, benötigt auch Karma eine Konfigurationsdatei. Karma bietet Ihnen hier einen interaktiven Wizzard, mit dessen Hilfe Sie eine entsprechende Konfiguration erstellen können. In Listing 2–19 sehen Sie diesen Prozess Schritt für Schritt.

```
$ karma init karma.conf.js

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next
question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next
question.
> no

Do you want to capture a browser automatically ?
Press tab to list possible options. Enter empty string to move to
the next question.
> Chrome
>
```

Listing 2–19
Karma – Erstellung der Konfigurationsdatei

```
What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
> spec/*.js
> src/*.js
>

Should any of the files included by the previous patterns be
excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes

Config file generated at "/tddjs/kapitel1/karma/karma.conf.js".
```

Für das Beispiel mit Karma kommt der gleiche Quellcode wie schon im Beispiel mit Jasmine zum Einsatz. Das bedeutet, dass Sie ein `src`-Verzeichnis mit einer Datei `source.js` haben, die den Quellcode Ihrer Applikation enthält, und ein Verzeichnis `spec` mit einer Datei `test.js`, das entsprechend die Tests enthält. Zur Ausführung der Tests wird Jasmine verwendet.

Wizzard zur Erstellung der Konfiguration

Mit dem Kommando `karma init karma.conf.js` erstellen Sie die Konfigurationsdatei. Die erste Frage nach dem verwendeten Framework können Sie mit dem Standardwert `jasmine` beantworten. Require.js kommt bei dieser Applikation nicht zum Einsatz. Entsprechend geben Sie bei der zweiten Frage `no` an.

Wie bei JsTestDriver binden Sie auch bei Karma Ihre Browser an den Server. Karma verfügt über die Möglichkeit, automatisch Browser an den Server zu binden. Sie können bei der Erstellung der Konfiguration die Namen von Browsern angeben, die Sie automatisch starten und an den Karma-Server binden möchten. Der Standardwert lautet hier Chrome.

Im nächsten Schritt müssen Sie angeben, welche Dateien Sie laden möchten. Zum einen sind dies die Testdateien im Verzeichnis `spec` und zum anderen die Quelldateien Ihrer Applikation im Verzeichnis `src`. Neben den eingeschlossenen Dateien können Sie außerdem noch Dateien ausschließen. In diesem Beispiel ist dies allerdings nicht notwendig. Deshalb geben Sie hier keinen Wert an.

Die letzte Frage, die Sie bei der Erstellung der Konfiguration beantworten müssen, ist, ob Sie möchten, dass Karma auf Änderungen an den Dateien lauscht und dann die Tests automatisch ausführt.

Nach der Beantwortung der Fragen können Sie Ihre Tests ausführen. Zu diesem Zweck müssen Sie lediglich das Kommando `karma start` auf der Kommandozeile absetzen. Listing 2–20 zeigt Ihnen das erforderliche Kommando und das Ergebnis der Tests.

Listing 2–20
Karma – Ausführung der Tests

```
$ karma start
WARN [config]: config.configure() is deprecated, please use
config.set() instead.
INFO [karma]: Karma v0.9.4 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 28.0.1500 (Mac OS X 10.8.4)]: Connected on socket id
cZLOYjtnMoZOkhThQQwb
Chrome 28.0.1500 (Mac OS X 10.8.4) test Addition of 1 and 3 FAILED
   Expected 4 to equal 5.
   Error: Expected 4 to equal 5.
       at null.<anonymous>
(/tddjs/kapitel1/karma/spec/test.js:7:27)
Chrome 28.0.1500 (Mac OS X 10.8.4): Executed 2 of 2 (1 FAILED)
(0.054 secs / 0.01 secs)
```

Sobald Sie nun die Dateien editieren, die Sie in der Konfiguration angegeben haben, werden die Tests automatisch ausgeführt. Sie können noch einen Schritt weitergehen und Karma in Ihre Entwicklungsumgebung integrieren. Mit dieser Erweiterung ist kein Wechsel zwischen der Entwicklungsumgebung und der Kommandozeile notwendig.

Integration in die Entwicklungsumgebung

Die Entwicklungsumgebung WebStorm von JetBrains bietet Ihnen ein Plug-in für Karma. Standardmäßig ist dieses Plug-in installiert und aktiviert. Das bedeutet, dass Sie mit einem Rechtsklick auf die Konfigurationsdatei über das Kontextmenü die Tests ausführen können. In Abbildung 2–5 sehen Sie das Ergebnis der Integration Ihrer Jasmine-Tests mit Karma in WebStorm.

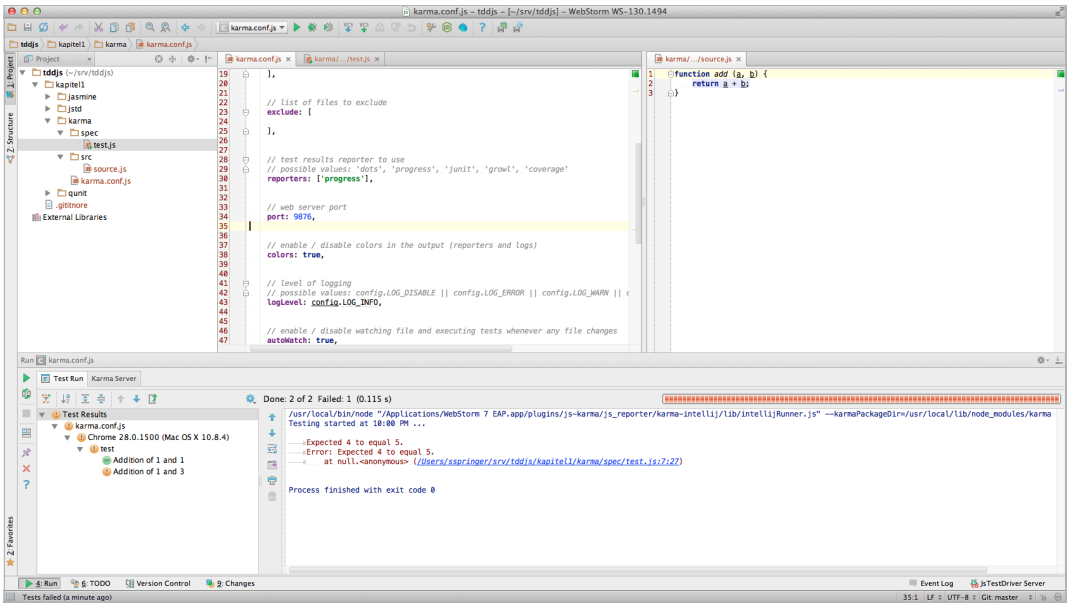

Abb. 2–5
Integration von Karma in die Entwicklungsumgebung

Der Vorteil dieser Integration besteht vor allem darin, dass Sie die Ergebnisse Ihrer Tests direkt in Ihrer Entwicklungsumgebung sehen können und bei einem Fehlschlag direkt an die entsprechende Stelle springen können. Außerdem können Sie die Testausführung steuern. Das bedeutet, dass die Tests nicht unbedingt bei jedem Speichervorgang ausgeführt werden müssen.

Einen Testlauf starten Sie entweder über das Kontextmenü Ihrer Entwicklungsumgebung oder über ein Tastaturkürzel.

2.9 Zusammenfassung

In diesem Kapitel haben Sie verschiedene Testframeworks kennengelernt. Dabei haben Sie vor allem die Unterschiede zwischen clientseitigen und serverseitigen Testframeworks gesehen.

Diese Testframeworks bilden die Basis für die testgetriebene Entwicklung. Im nächsten Kapitel erfahren Sie, wie Sie mit JavaScript und Jasmine Ihre Applikation testgetrieben entwickeln können.