**Buster statt Basta**

# Buster.JS – Nächste Generation des Unit-Testens in JavaScript

**Daniel Wittner**

*Wer auf der Suche nach einem guten Werkzeug zum Unit-Testen von JavaScript ist, findet im Open-Source-Framework Buster.JS ein geeignetes Testwerkzeug, da es das Testen clientseitig im Browser und serverseitig in Node.js unterstützt und alle wichtigen Merkmale anderer Unit-Test-Werkzeuge vereint. Buster.JS unterstützt das Testen mittels statischer HTML-Seiten wie QUnit, das Testen von asynchronem Code wie Mocha und es kann Browser automatisieren wie JsTestDriver. Trotz dieser beeindruckenden Anzahl an unterstützten Funktionen ist es dennoch einfach zu bedienen.*


Initiiert wurde die Entwicklung von *Buster.JS* maßgeblich von Christian Johansen, Autor des Buches „Test-Driven JavaScript Development" (TDDJS, [Joha10]). Von ihm stammt auch das beliebte Stubbing- und Mocking-Framework *Sinon.JS* und das vielseitige JavaScript-Werkzeug *Juicer*. Aktuell befindet sich Buster.JS zwar noch im Beta-Stadium, aber das Release für die stabile Version 1.0 ist noch für dieses Jahr geplant.

## Installation und Konfiguration

Der beste Weg ein Werkzeug kennenzulernen, ist es auszuprobieren. Daher wollen wir uns anhand eines einfachen Beispiels anschauen, wie Unit-Tests für Buster.JS aussehen und wie man sie im Browser und in Node.js ausführen kann. Dazu erstellen wir ein Modul `room`, welches Teil einer Alarmanlage sein soll. Das Modul ermöglicht das Erzeugen von Objekten, welche die zu überwachenden Räume repräsentieren. Ein Raum verwaltet eine Liste von Sensoren und kann diese asynchron auslesen.

Bevor es an die Implementierung geht, muss Buster.JS installiert und konfiguriert werden. Da es sich dabei um ein Node.js-Modul handelt, lässt es sich einfach über den *Node Paket Manager* mittels `npm install -g buster` installieren. Erfreulicherweise klappt die Installation inzwischen auch unter Windows, auch wenn die Verwendung von Buster.JS unter Windows an der einen oder anderen Stelle noch ein wenig hakt. Eine vollständige Unterstützung für Windows ist erst für die Version 1.0 geplant.

Nach erfolgreicher Installation muss Buster.JS mitgeteilt werden, wo sich die Quell- und Testdateien befinden und in welcher Umgebung die Tests ausgeführt werden sollen – ob im Browser oder in Node.js. Die Konfiguration erfolgt in Form eines *CommonJS*-Moduls, also in JavaScript. Bei Buster.JS unterscheidet man zwischen einer Konfigurationsdatei und den eigentlichen Konfigurationen, Konfigurationsgruppen genannt.

Eine Konfigurationsdatei kann dabei mehrere Konfigurationen enthalten. Das Modul exportiert deshalb ein Array von Konfigurationsobjekten. Diese können sich gegenseitig sogar erweitern. Wir werden später davon noch Gebrauch machen.

## Testen im Browser – Browser-Automatisierung

Unsere Konfiguration für das Testen im Browser sieht wie folgt aus:

```
var config = module.exports;

config["browser tests"] = {
  environment: "browser",
  libs: [
    "lib/**/*.js"
  ],
  sources: [
    "src/as.js"
    "src/**/*.js"
  ],
  tests: [
    "test/**/*-test.js"
  ]
};
```

Alle Pfade in der Konfigurationsdatei sind relativ und werden standardmäßig ausgehend vom Verzeichnis der Konfigurationsdatei aufgelöst. Es ist aber auch möglich, ein Verzeichnis anzugeben, von dem aus die Pfade aufgelöst werden sollen.

Damit Buster.JS die Konfigurationsdatei automatisch findet, muss sie `buster.js` benannt und im Wurzelverzeichnis des Projekts oder im Unterverzeichnis `test` oder `spec` gespeichert werden. Alternativ kann auch das Kommandozeilenargument `--config` eingesetzt werden, um die zu verwendende Konfigurationsdatei explizit anzugeben.

In der obigen Konfigurationsdatei sehen wir, dass eine JavaScript-Datei namens `as.js` vor allen anderen Quelldateien geladen wird. In dieser erzeugen wir den Namensraum `as` (alarm system) für die Anwendung, über den dann auf die Module der Anwendung zugegriffen wird.

Entsprechend der Konfiguration werden alle Dateien, die sich unterhalb des Verzeichnisses `test` befinden und auf `-test.js` enden, als Tests ausgeführt. Zum Testen des Moduls `room` erstellen wir die Datei `room-test.js` und legen sie direkt im Verzeichnis `test` ab. Als Erstes wollen wir prüfen, ob das Hinzufügen eines Sensors zu einem Raum funktioniert:

```
var assert = buster.assert;

buster.testCase("room module", {
  setUp : function () {
    this.room = new as.Room();
  },

  "add a sensor to the room" : function () {
    var sensor = {};
    this.room.addSensor(sensor);
    assert.equals(this.room.getSensors().length, 1);
  }
});
```

Die Syntax der Testdatei entspricht der von *xUnit*, das heißt, die Tests beziehungsweise die Test-Methoden sind in Testfällen zusammengefasst. Einem Testfall können die Methoden **setUp** und **tearDown** hinzugefügt werden, die vor beziehungsweise nach jeder Test-Methode ausgeführt werden.

Wie bereits erwähnt, ist Buster.JS in der Lage, Browser zu automatisieren. Die Art und Weise, wie das geschieht, ist von *JsTestDriver* übernommen. Das Besondere an der Automatisierung ist, dass die Tests in beliebig vielen und insbesondere verschiedenen Browsern gleichzeitig ausgeführt werden können. Folgende Schritte sind dazu notwendig:

### 1. Schritt: Starten des Servers

```
buster@ubuntu:~$ buster-server
buster-server running on http://localhost:1111
```

Wird kein Port angegeben, verwendet Buster.JS standardmäßig den Port *1111*.

### 2. Schritt: Starten des Browsers

Sobald der Server erfolgreich gestartet wurde, kann der erste Browser geöffnet und in die Adresszeile die URL des Servers eingegeben werden. In unserem Beispiel ist das **http://localhost:1111**. Daraufhin erscheint die in Abbildung 1 gezeigte Seite. Auf dieser muss die Schaltfläche *Capture browser* betätigt werden. Anschließend kann der Browser minimiert und der Schritt für beliebig viele Browser wiederholt werden.

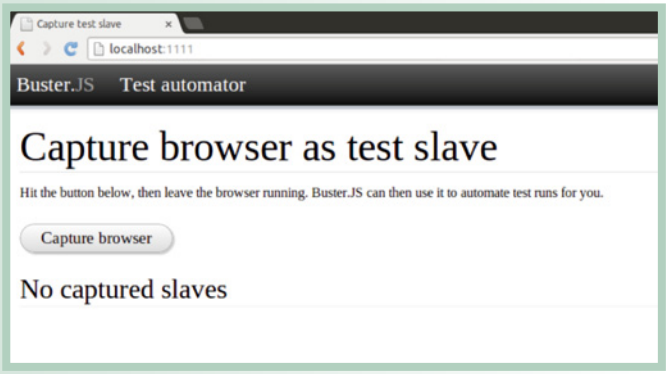


Abb. 1: Startseite

### 3. Schritt: Ausführen der Tests

```
buster@ubuntu:~$ buster-test
Chrome 24.0.1312.56, Ubuntu Chromium: .
Firefox 19.0, Ubuntu:                 .
2 test cases, 2 tests, 2 assertions, 0 failures, 0 errors, 0 timeouts
Finished in 0.015s
buster@ubuntu:~$
```

Beim Ausführen der Tests wird zunächst ein Client gestartet. Dieser liest die Konfigurationsdatei ein, lädt die dort angegebenen Quell- und Testdateien und schickt diese zum Server. Dieser überträgt die Dateien zu den Browsern, wo die Tests anschließend ausgeführt werden. Den Status und das Ergebnis der Tests melden die Browser zurück an den Server und dieser zurück an den Client. In der Beispiel-Ausgabe sehen wir das Ergebnis des Tests, nachdem die entsprechende Funktionalität implementiert wurde.

Nach Änderungen an der Implementierung braucht nur Schritt 3 wiederholt zu werden. Den Server und die Browser lässt man einfach im Hintergrund weiterlaufen. Das vereinfacht die testgetriebene Entwicklung erheblich.

## Testen in Node.js

Da wir Buster.JS durch den Schalter **-g** global installiert haben, müssen wir für unser Projekt, um auch für Node.js testen zu können, einen Link darauf anlegen. Das erreichen wir durch den Aufruf von **npm link buster** im Wurzelverzeichnis des Projekts.

Damit unser Test nun auch für Node.js ausgeführt wird, müssen wir eine zusätzliche Konfiguration erstellen. Dazu erweitern wir die vorhandene Konfigurationsdatei um folgenden Code:

```
config["node.js tests"] = {
  extends: "browser tests",
  environment: "node"
};
```

Aufgrund des Modulsystems von Node.js muss der Test um eine Anweisung zum Laden des Moduls **buster** und eine zum Laden des Moduls für den Namensraum der Anwendung erweitert werden. Durch eine zusätzliche Bedingung wird sichergestellt, dass die Anweisungen nur für Node.js ausgeführt werden:

```
if (typeof require === "function" && typeof module === "object") {
  var buster = require("buster");
  var as = require("../src/as");
}
```

Ähnliche Anpassungen müssen auch an den Quelldateien vorgenommen werden. Anschließend können die Tests wieder mit dem Kommando **buster-test** ausgeführt werden:

```
buster@ubuntu:~$ buster-test
Chrome 24.0.1312.56, Ubuntu Chromium: .
Firefox 19.0, Ubuntu:                 .
2 test cases, 2 tests, 2 assertions, 0 failures, 0 errors, 0 timeouts
Finished in 0.016s
room module: .
1 test case, 1 test, 1 assertion, 0 failures, 0 errors, 0 timeouts
Finished in 0.006s
```

## Asynchrone Tests

Eine elementare Anforderung an ein Testwerkzeug für JavaScript ist das Testen asynchroner Funktionalität. Buster.JS bietet dafür drei Möglichkeiten an:

### 1. Möglichkeit: mit Sinon.JS

Buster.JS enthält das Stubbing- und Mocking-Framework Sinon.JS. Dieses stellt das **clock**-Objekt zur Manipulation der Zeit bereit. Darüber kann zum Beispiel getestet werden, ob Funktionen, die mittels **setTimeout** asynchron ausgeführt werden sollen, auch tatsächlich ausgeführt wurden.

Da Sinon.JS aber recht einfach in jedes Test-Werkzeug eingebunden werden kann, soll dieser Ansatz hier nicht näher betrachtet werden.

**2. Möglichkeit: mit done-Parameter**
Man kann einen asynchronen Test erstellen, indem man einen Parameter `done` für die Test-Methode deklariert:

```
"read sensors" : function (done) {
  ...
  room.readSensors(done(function (err, results) {
    assert.equals(room.getSensors()[0].state, 0);
    assert.equals(room.getSensors()[1].state, 1);
  }));
}
```

Das markiert den Test als asynchron und es wird bei Ausführung des Tests für `done` eine Callback-Funktion übergeben. Diese muss innerhalb des Tests aufgerufen werden, um das Ende des Tests zu markieren. Wird sie nicht oder nicht rechtzeitig aufgerufen, läuft der Test in einen Timeout und schlägt schließlich fehl.

**3. Möglichkeit: mit Promise**
Alternativ zum Parameter `done` kann der Test auch ein *Promise* zurückliefern. Buster.JS wertet jedes Objekt als *Promise*, das eine Methode `then` besitzt. Unter Verwendung von *when.js* könnte der Test so aussehen:

```
"read sensors": function () {
  var deferred = when.defer();

  room.readSensors(function (err, results) {
    assert.equals(room.getSensors()[0].state, 0);
    assert.equals(room.getSensors()[1].state, 1);
    deferred.resolver.resolve();
  });
  return deferred.promise;
}
```

## Assertions

Anders als andere Unit-Test-Werkzeuge verwendet Buster.JS nicht das Präfix *„not“*, um eine Assertion zu negieren, sondern besitzt dafür ein passendes Gegenstück, die Refutation. Statt `assert.notEquals(foo, bar)` schreibt man in Buster.JS `refute.equals(foo, bar)`.

Buster.JS besitzt eine Menge vordefinierter Assertions. Es erlaubt aber auch das Hinzufügen eigener Assertions. Beim Testen der Alarmanlagensoftware kann es zum Beispiel sein, dass der Entwickler hin und wieder prüfen muss, ob es in einem Raum einen Alarm gibt oder nicht. Sofern die Anwendung selbst keine Funktion bereitstellt, mit welcher dies abgefragt werden kann, könnte man sich für den Test eine entsprechende Helferfunktion schreiben. Die Verwendung einer eigenen Assertion ist allerdings noch eleganter. Sie ist nicht nur komfortabel, sondern auch integriert. Das bedeutet, dass sie automatisch im Reporting mit eingebunden wird:

```
buster.assertions.add("isSafe", {
  assert: function (room) {
    for (var i = 0; i < room.sensors.length; i++) {
      if (room.sensors[i].state !== 1) {
        return false;
      }
    }
    return true;
  },
  assertMessage: "Expected the room ${0} to be safe!",
  refuteMessage: "Expected the room ${0} to be not safe!",
  expectation: "toBeSafe"
});
```

Verwendet werden kann die Assertion dann durch `assert.isSafe(room)` und `refute.isSafe(room)`.

## Ausführen einer Teilmenge von Tests

Oft ist es hilfreich, nur eine Teilmenge an Tests auszuführen, um sich zum Beispiel auf die Komponente zu konzentrieren, an der man gerade arbeitet, oder um nach der Ursache für einen Fehler zu suchen. Buster.JS bietet verschiedene Filter an, um die auszuführenden Tests zu beschränken. Mit der Option `--environment` oder `-e` kann die Testausführung auf Browser (`browser`) oder auf Node.js (`node`) beschränkt werden. Mit der Option `--config-group` oder `-g` kann eine Liste von Konfigurationsgruppen angegeben werden, deren Tests ausgeführt werden sollen. Die einzelnen Elemente müssen mit einem Komma getrennt werden. Buster.JS interpretiert dabei die Angabe eines Elements als regulären Ausdruck. Das heißt, es können mit einer Angabe die Tests mehrerer Konfigurationsgruppen ausgeführt werden, sofern deren Name zum regulären Ausdruck passt.

Ohne Angabe einer Option kann eine Liste mit den Namen von Tests oder Testfällen angegeben werden, die ausgeführt werden sollen. Auch hier müssen die einzelnen Elemente durch ein Komma getrennt werden, und jedes Element wird als regulärer Ausdruck interpretiert. Da all diese Angaben als Filter wirken, können sie beliebig kombiniert werden.

Soll genau ein Test ausgeführt werden, kann man auch die sogenannte Fokusrakete verwenden. Dabei handelt es sich um die Zeichenfolge `=>`, die dem Namen des Tests vorangestellt wird, der ausgeführt werden soll:

```
"=>read sensors" : function (done) {
  ...
}
```

## Aussetzen von Tests

Manchmal möchte der Entwickler einen Test temporär deaktivieren, zum Beispiel wenn er bereits einen Test für eine Funktionalität geschrieben hat, die er erst in naher Zukunft implementiert. Auch hierfür bietet Buster.JS eine Lösung. Durch Voranstellen von `//` zum Namen eines Tests oder Testfalles wird dieser nicht ausgeführt. Als Erinnerung gibt Buster.JS eine Liste aller ausgesetzten Tests im Report aus. So können diese nicht vergessen werden.

## Strukturieren mit geschachtelten Testfällen

Bislang hält sich die Menge an Tests in unserem Beispiel in Grenzen und ist noch gut zu überblicken. Mit steigender Anzahl wird dies aber zunehmend schwieriger. Jeder Entwickler sollte sich rechtzeitig überlegen, wie er seine Tests strukturiert, um den Überblick nicht zu verlieren. Buster.JS bietet dafür ein einfaches, aber wirksames Instrument an: geschachtelte Testfälle. Jeder Testfall kann beliebig viele andere Testfälle enthalten und die Schachtelung kann beliebig tief erfolgen. Auf diese Weise können die Tests einfach gruppiert und strukturiert werden. Vor der Ausführung eines Tests werden alle `setUp`-Metho-

den der umgebenden Testfälle von außen nach innen aufgerufen und abschließend alle `tearDown`-Methoden von innen nach außen.

Im Fehlerfall fügt Buster.JS die Namen der geschachtelten Testfälle und den Namen des fehlgeschlagenen Tests zusammen. So ließen sich beispielsweise die beiden Tests aus dem Abschnitt „Asynchrone Tests“ wie folgt in einen geschachtelten Testfall gruppieren:

```
buster.testCase("room module", {
  ...
  "read sensors with": {
    "done callback": function (done) {
      ...
    },
    "promise": function () {
      ...
    }
  }
});
```

## Weitere nützliche Funktionen

Die Liste der Funktionen, die Buster.JS bereitstellt, ist zu lang, um hier alle detailliert zu beschreiben. Die folgende Liste gibt einen Überblick weiterer nützlicher Funktionen mit einer kurzen Beschreibung.

- Testen mit statischen HTML-Seiten: Wie die meisten Unit-Test-Werkzeuge für JavaScript unterstützt auch Buster.JS das Testen mittels statischer HTML-Seiten im Stil von QUnit.
- Integriertes Sinon.JS: Wie im Abschnitt „Asynchrone Tests“ erwähnt, ist Sinon.JS, ein Framework zum Erzeugen von Spies, Stubs und Mocks, bereits in Buster.JS integriert. Somit kann man von Anfang an gut isolierte Unit-Tests schreiben.
- Feature-Detection: Mit dem Objekt `requiresSupportFor` können für einen Testfall Bedingungen angegeben werden, die erfüllt sein müssen, damit der Testfall ausgeführt wird. Das können grundsätzlich beliebige Bedingungen sein. Gedacht ist diese Funktion aber zur Feature-Detection.
- Expectations: Statt Assertions und Refutations können auch Expectations verwendet werden, was oftmals von der Verhaltensgetriebenen Softwareentwicklung (*BDD*) bevorzugt wird. Die Expectation für unsere hinzugefügte Assertion „`isSafe`“ haben wir „`toBeSafe`“ genannt.
- *BDD*-Syntax: Neben der xUnit-Syntax, die wir für unser Beispiel verwendet haben, unterstützt Buster.JS standardmäßig auch die BDD-Syntax.
- Reporter: Buster.JS enthält verschiedene Reporter, das heißt, es kann den Verlauf und das Ergebnis eines Testlaufs in verschiedenen Formaten bereitstellen: XML (xUnit), dots (Matrix aus Punkten für grüne Tests, „F“ für fehlerhafte usw.), Specification (nodeunit), TAP (Test Anything Protocol), *TeamCity* und mehr.
- Buster AMD: Erweiterung zum Testen von Anwendungen, die AMD (Asynchronous Module Definition) und einen Programmlader nutzen.
- Benutzerdefinierte Testumgebung: Es kann eine HTLM-Datei angegeben werden, in der die Tests ausgeführt werden. Damit lassen sich die Tests beispielsweise in einem HTML4 Strict-Dokument ausführen, statt in einem HTML5-Dokument, was der Standard bei Buster.JS ist. Diese Funktion ist allerdings noch nicht in der Beta-Version implementiert.
- *Headless browser testing:* Wenn man den Buster.JS-Server nicht startet, werden die Browsertests headless in *PhantomJS* ausgeführt. Auch diese Funktion ist noch nicht in der Beta-Version implementiert.

## Fazit

Aufgrund der großen Anzahl an Funktionen, die Buster.JS bereitstellt, deckt es fast alle Aspekte des Unit-Testens von JavaScript-Anwendungen ab. Benötigt man dennoch weitere Funktionalitäten, oder eine vorhandene Funktionalität entspricht nicht ganz den eigenen Bedürfnissen, kann Buster.JS sehr einfach angepasst werden. Es gibt für fast alles eine öffentliche Programmierschnittstelle (API). Es lassen sich Reporter für weitere Ausgabeformate ergänzen, Wrapper erstellen, um Tests anderer Test-Frameworks in Buster.JS auszuführen, weitere Test-Syntaxen aufnehmen und vieles mehr. Ich nutze Buster.JS jetzt seit einiger Zeit und bin äußerst zufrieden. Das alles übergreifende Konzept hat mich überzeugt.

## Links

**[BusterJS]** `https://github.com/busterjs`
**[BusterJSHome]** `http://busterjs.org`
**[BusterJSHT]** Hybrid testing, Buster.JS 0.7.0 documentation, `http://docs.busterjs.org/en/latest/hybrid-testing/`
**[BusterJSOver]** `http://docs.busterjs.org/en/latest/overview/`
**[Joha10]** Ch. Johansen, Test-Driven JavaScript Development, Addison-Wesley, 2010, `http://cjohansen.no/`
**[Juicer]** `https://github.com/cjohansen/juicer`
**[SinonJS]** Versatile standalone test spies, stubs and mocks for JavaScript, `http://sinonjs.org/`

**Daniel Wittner** ist Senior Software-Engineer in dem Software- und Beratungshaus PPI AG, das sich auf Finanzunternehmen spezialisiert hat. Nach acht Jahren Entwicklung und Qualitätssicherung von Java-EE-Anwendungen beschäftigt er sich seit zwei Jahren mit JavaScript und dessen Integration in Entwicklungs- und Qualitätssicherungsprozesse. Sein Anspruch ist dabei, die aus der Java-EE-Welt gewohnten Maßstäbe auf JavaScript zu übertragen.
E-Mail: daniel.wittner@ppi.de