Unit-Tests für JavaScript und AJAX

Peter Krenn

10. Januar 2011

Zusammenfassung

Zusammenfassung

Inhaltsverzeichnis

1 Einleitung

2 QUnit

QUnit[13] wird von Mitgliedern des jQuery Teams[5] entwickelt und ist auch die offizielle *test suite* des jQuery Projektes. Es zählt zur Gruppe der xUnit Test Frameworks[2], die einem Design folgen, das erstmals mit SUnit[1] für Smalltalk formuliert wurde und als JUnit für Java große Verbreitung fand.

2.1 Assertions

Der zentrale Bestandteil eines *unit tests* ist die *assertion*. Eine *assertion* ist eine Aussage über den Zustand einer bestimmten Stelle in einem Programm[9]. Der Entwickler macht eine Zusicherung, dass eine bestimmte Bedingung unabhänging von den Laufzeitumständen immer wahr ist.

QUnit stellt eine Reihe von allgemeinen *assertions* zur Verfügung, die JUnit nachempfunden sind, und auch solche, die speziell an JavaScript und asynchrone Entwicklung angepasst sind[13]:

`ok(state, message)`

`ok()` ist eine boolsche *assertion*, die positiv terminiert, wenn der erste Parameter wahr ist. Den zweiten, optionalen Parameter `message` haben alle *assertion*-Funktionen gemein: dieser wird gemeinsam mit dem Ergebnis ausgegeben.

```
ok(true, "terminiert immer positiv");
ok("", "Leerstring entspricht false, die assertion schlägt fehl");
```

`equal(actual, expecteded, message)`

Diese *assertion* ist `ok()` sehr ähnlich, allerdings werden die `actual` und `expecteded` Werte mit dem Ergebnis ausgegeben. Dadurch werden die Tests aussagekräftiger und *debugging* einfacher. `equal` terminiert positiv, wenn `actual == expecteded`.

```
var actual = 1;
equal(actual + 1, 2, "terminiert positiv");
equal(actual, 2, "schlägt fehl");
```

strictEqual(actual, expected, message)

Im Gegensatz zu `equal()` werden bei dieser *assertion* die Werte mit `actual === expecteded` verglichen. Dieser Operator vergleicht zusätzlich die Typengleichheit der beiden Parameter.

```
var actual = 0;

// terminiert positiv, da == den Datentyp ignoriert
equal(actual, false);

// schlägt fehl, da 0 vom Typ Number und false Boolean ist
strictEqual(actual, false);
```

deepEqual(actual, expected, message)

`deepEqual()` überprüft primitve Datentypen, *arrays* und Objekte rekursiv auf Gleichheit. Sie ist dabei auch strikter als `equal()`, da der === Operator verwendet wird.

```
var actual = {a: 1};

// equal schlägt mit verschiedenen Objekten fehl
equal(actual, {a: 1});

// schlägt fehl, da die Datentypen verschieden sind;
deepEqual(actual, {a: "1"});

// terminiert positiv, da die Inhalte der Objekte gleich sind
deepEqual(actual, {a: 1}
```

notEqual(actual, expected, message)

Diese *assertion* ist die invertierte Version von `equal()`. Entsprechende Funktionen existieren auch für `strictEqual()` und `deepEqual()`: `notStrictEqual()` und `notDeepEqual`.

```
var actual = 0;
notEqual(actual, false, "schlägt fehl");
notStrictEqual(actual, false, "terminiert positiv");

var actual = {a: 1};
notDeepEqual(actual, {a: "1"}, "terminiert positiv");
```

`raises(state, message)`

`raises()` überprüft, ob die übergebene *callback* Funktion eine *exception* wirft. Die Funktion wird ohne Parameter im *default scope* aufgerufen.

```
raises(function() {
  throw new Error("Fehlerfall");
}, "terminiert positiv, falls der gewünschte Fehler auftritt");
```

2.2 Testfunktionen

Werden die vorgestellten *assertions* direkt ausgeführt, unterbrechen sie im Falle eines Fehlers den Testdurchlauf. Deswegen werden sie in Testfunktionen eingebettet.

Es ist auch üblich, dass jede Testfunktion nur eine spezifische Eigenschaft testet. Dies geschieht oft mit mehreren *assertions* und die Testfälle bleiben dabei überschaubar und einfach zu verstehen.

QUnit macht das mit der Funktion `test(name, expected, test)`. `name` ist eine Bezeichnung des Tests und `test` ist eine Funktion, die die *assertions* enthällt. `expect` ist optional und erlaubt es die Anzahl der zu erwartenden *assertions* festzulegen. Dies ist bei asynchronen Tests von Bedeutung.

```
Beispiel
```

2.3 Test Suites

Mit der `module(name, lifecycle)` Anweisung ist es möglich, die Testfälle weiter in Module aufzuteilen. Dies dient einerseits der Strukturierung der Tests und andererseits wird es auch möglich, für mehrere Testfälle relevante Daten vor deren Ausführung zu definieren.

```
Beispiel
```

Im Parameter `lifecycle` kann man optional die Funktionen `setup()` und `tearDown()` übergeben, die vor beziehungsweise nach jedem Test ausgeführt werden. Sie teilen den `this`-*scope* der Testfunktionen, wodurch sogenannte *fixtures* erstellt werden können. Die Daten werden nach jedem Test zurückgesetzt, was in `tearDown()` auch manuell geschehen kann.

2.4 Asynchrones Testen

Asynchrone Funktionen, wie sie bei *AJAX requests* und Aufrufen von `setTimeout()` und `setInterval()` vorkommen, können mit den bisher vorgestellten Methoden nicht getestet werden.

Im folgenden Beispiel[3] wird die *assertion* nicht ausgeführt, da das Ergebnis des Testfalles zum Zeitpunkt des Aufrufens bereits feststeht.

```
test("Asynchroner Test", function() {
  setTimeout(function() {
    ok(true);
  }, 100)
});
```

QUnit stellt aus diesem Grund zwei Funktionen zur Verfügung, mit der sich der Testablauf pausieren und fortsetzen lässt.

```
test("Asynchroner Test", function() {
  stop();

  setTimeout(function() {
    ok(true);

    start();
  }, 100)
})
```

`stop()` hält den Test an und nach Aufrufen der *assertion*, wird mit `start()` wieder fortgesetzt. Da der Fall, dass `stop()` am Anfang eines Tests ausgeführt wird, häufig vorkommt, gibt es dafür auch ein spezielle Funktion names `asyncTest()`.

Wird nicht ein `setTimeout()` Aufruf sondern ein AJAX request getestet, bei dem man sich nicht sicher sein kann, wann und ob die *callback* Funktion ausgeführt wird, ist es möglich der `stop()` Funktion ein Intervall als Paramter mitzugeben, nachdem der Testablauf automatisch fortgesetzt wird. Der betreffende Test schlägt dabei fehl.

Werden in einem Testfall mehrere zeitverzögerte Aufrufe getestet, ist es notwendig, `start()` manuell mit `setTimout()` ausreichend zu verzögern, so dass alle *callbacks* ausreichend Zeit zum Terminieren haben.

Um sicher zu stellen, dass alle *callbacks* mit den enthaltenen *assertions* ausgeführt werden, gibt es eine `expect()` Funktion, der man als Parameter die zu erwartende Anzahl der *assertions* übergeben kann.

```
// Ein AJAX request mit der zu testenden callback Funktion
function ajaxCall(successCallback) {
  jQuery.ajax({
    url: "http://server.com/",
    success: successCallback
  });
}

asyncTest("Asynchroner Test", function() {
  // asyncTest unterbricht den Testablauf

  // 3 assertions sollten ausgeführt werden
  expect(3);

  ajaxCall(function() {
    ok(true);
  });

  ajax(function() {
    ok(true);
    ok(true);
  })

  // Nach 3000 ms wird der Testablauf fortgesetzt
  setTimeout(function() {
    start();
  }, 3000);
});
```

Die Kombination von `expect()` mit `asyncTest()` ist wiederum so verbreitet, dass die Anzahl der zu erwartenden *assertions* als zweiter Parameter direkt in den `asyncTest()` Aufruf verlegt werden kann.

JQuery stellt dafür die Methoden `stop()` und `start()`

2.5 Testablauf

Die Ausgabe von QUnit erfolgt ausschließlich im Web-Browser. Die Dateien `qunit.js` und `qunit.css` müssen gemeinsam mit dem zu testenden JavaScript Code und den Tests in eine HTML-Datei eingebunden werden.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script type="text/javascript" src="qunit.js"></script>
    <link rel="stylesheet" href="qunit.css" type="text/css"
          media="screen" />

    <!-- project file -->
    <script type="text/javascript" src="my_project.js"></script>

    <!-- tests file -->
    <script type="text/javascript" src="my_tests.js"></script>
  </head>
  <body>
    <h1 id="qunit-header">QUnit example</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
    <div id="qunit-fixture">test markup, will be hidden</div>
  </body>
</html>
```

Beim Ausführen der Datei im Web-Browser befüllt QUnit die Elemente im `body` mit den Ergebnissen der Tests.

```
REENSHOT
```

```
hr schreiben
```

2.6 Headless Testing

2.7 Continuouse Integration

3 Jasmine

Jasmine[4] gehört zur Gruppe der Behavior Driven Development[10] oder BDD Frameworks. Im Gegensatz zu traditionellen xUnit Test Frameworks versuchen diese die Sprache, in der die Tests verfasst sind, einfacher und damit auch für nicht-technische Projektteilnehmer verständlich zu machen. BDD soll dabei helfen, die gewünschte *software behavior* in Diskussion mit den Auftraggebern klar zu definieren.

```
hr schreiben?
```

Im Gegensatz zu seinem Vorgänger Screw.Unit[8] und QUnit ist Jasmine ein Headless Testing Framework, das heißt es benötigt keinen Browser und ist nicht von anderen JavaScript Libraries abhängig. Dies vereinfacht den automatisierten Aufruf der Tests und ermöglicht es Server-side Anwendungen zu testen. Wie QUnit unterstützt es asynchrones Testen und verfügt außerdem über Mocking und Stubbing Funktionalitäten.

3.1 Specs

Specs entsprechen den Testfunktionen von klassischen xUnit Test Frameworks. Sie beschreiben das Verhalten einer Funktion oder eines Objektes mit einem Aufruf der Funktion `it()` in Form einer natürlichen Sprache. Die Beschreibung sollte aussagekräftig sein, da sie im Test Bericht ausgegeben wird.

```
it("should increment a variable", function() {
  var value = 0;
  value++;
});
```

3.2 Expectations

Expectations drücken die Erwartungen an das Verhalten eines bestimmten Teils des Quellcodes mittels der `expect()` Funktion aus. Sie sind vergleichbar mit den *assertions* der xUnit Frameworks. Das Verhalten wird durch Matchers spezifiziert.

```
it("should increment a variable", function() {
  // Definition der Testdaten
  var value = 0;

  // Ausführen des zu testenden codes
  value++;

  // Testen der Erwartung
  expect(value).toEqual(1);
});
```

3.3 Matchers

Jasmine verfügt über eine Reihe von eingebauten Matchers und deren Inversionen, die man durch Voranstellen von `.not` erhält:

`toEqual()`

Dieser Matcher entspricht der QUnit *assertion* `deepEqual()` und kann außerdem Datumswerte und DOM-Nodes vergleichen.

```
// Alle expectations terminieren positiv
expect(true).toEqual(true);
expect(false).not.toEqual(true);

expect("5").not.toEqual(5);

expect({a: 1}).not.toEqual({a: "1"});

expect(new Number("5")).toEqual(5);

expect(new Date("January 11, 2011, 12:00:00"))
  .not.toEqual(new Date("January 12, 2011, 12:30:00"));

var nodeA = document.createElement("div");
var nodeB = document.createElement("div");
expect(nodeA).toEqual(nodeA);
expect(nodeA).not.toEqual(nodeB);
```

`toBe()`

`toBe()` entspricht `strictEqual()`.

```
var a = {};
var b = {};
var c = b;

expect(a).not.toBe(b);
expect(c).toBe(b);
```

4 Regression Testing

Regression Testing[12] versucht Probleme und Nebenwirkungen von Modifikationen im Quellcode durch Wiederholung der Testfälle aufzuzeigen. Ein verwandter Begriff ist Continuous Integration[11], die kontinuierliche Qualitätskontrolle für ein Projekt erreichen will. Automatisiertes Testing ist eines ihrer Ziele.

Da es keine einheitliche JavaScript Implementierung gibt, sondern diese sich je nach Browser, Browserversion und Betriebssystem unterscheiden, ist die Durchführung der Testläufe problematisch.

Ein Ansatz der sich derzeit in Entwicklung befindet ist TestSwarm[6] von John Resig, dem Autor von jQuery und QUnit. Es handelt sich dabei um ein System, das den Code eines Projektes bei jedem *commit* im Versionierungssystem automatisch auf verschiedenen Plattformen und Browsern testet.

Das System wird von einem zentralen Server aus gesteuert, der Testaufträge an mehrere Clients, auf denen verschiedene Browserversionen laufen, weiterleitet. Das Ergebnis der Prozesses ist eine tabellarische Ansicht die für jeden *commit* darstellt, auf welchen Plattformen und Browsern die Test Suite fehlerfrei angewendet worden ist.

Im Gegensatz zu vergleichbaren Systemen, wie Selenium[7], ist TestSwarm vom verwendeten Test Framework unabhängig: es werden alle im Moment gebräuchlichen Frameworks unterstützt.

Literatur

[1] Kent Beck. Simple smalltalk testing: With patterns. *Smalltalk Report*, 4(2):16–18, 1994.

[2] Martin Fowler. xUnit. http://www.martinfowler.com/bliki/Xunit.html, 2010.

[3] Gaving Huang. How to test your JavaScript code with QUnit | nettuts+. http://net.tutsplus.com/tutorials/javascript-ajax/how-to-test-your-javascript-code-with-qunit/, 2010.

[4] Jasmine. Jasmine - BDD for javascript. http://pivotal.github.com/jasmine/, 2011.

[5] John Resig. jQuery. http://jquery.com/, 2011.

[6] John Resig. TestSwarm. https://github.com/jeresig/testswarm/wiki, 2011.

[7] Selenium. Selenium web application testing system. http://seleniumhq.org/, 2011.

[8] Nathan Sobo. Screw.Unit. https://github.com/nathansobo/screw-unit, 2011.

[9] Wikipedia. Assertion (computing). http://en.wikipedia.org/wiki/Assertion_(computing), 2011.

[10] Wikipedia. Behavior driven development. http://en.wikipedia.org/wiki/Behavior_Driven_Development, 2011.

[11] Wikipedia. Continuous integration. http://en.wikipedia.org/wiki/Continuous_integration, 2011.

[12] Wikipedia. Regression testing. http://en.wikipedia.org/wiki/Regression_testing, 2011.

[13] Jörg Zaefferer and John Resig. QUnit. http://docs.jquery.com/Qunit, 2011.