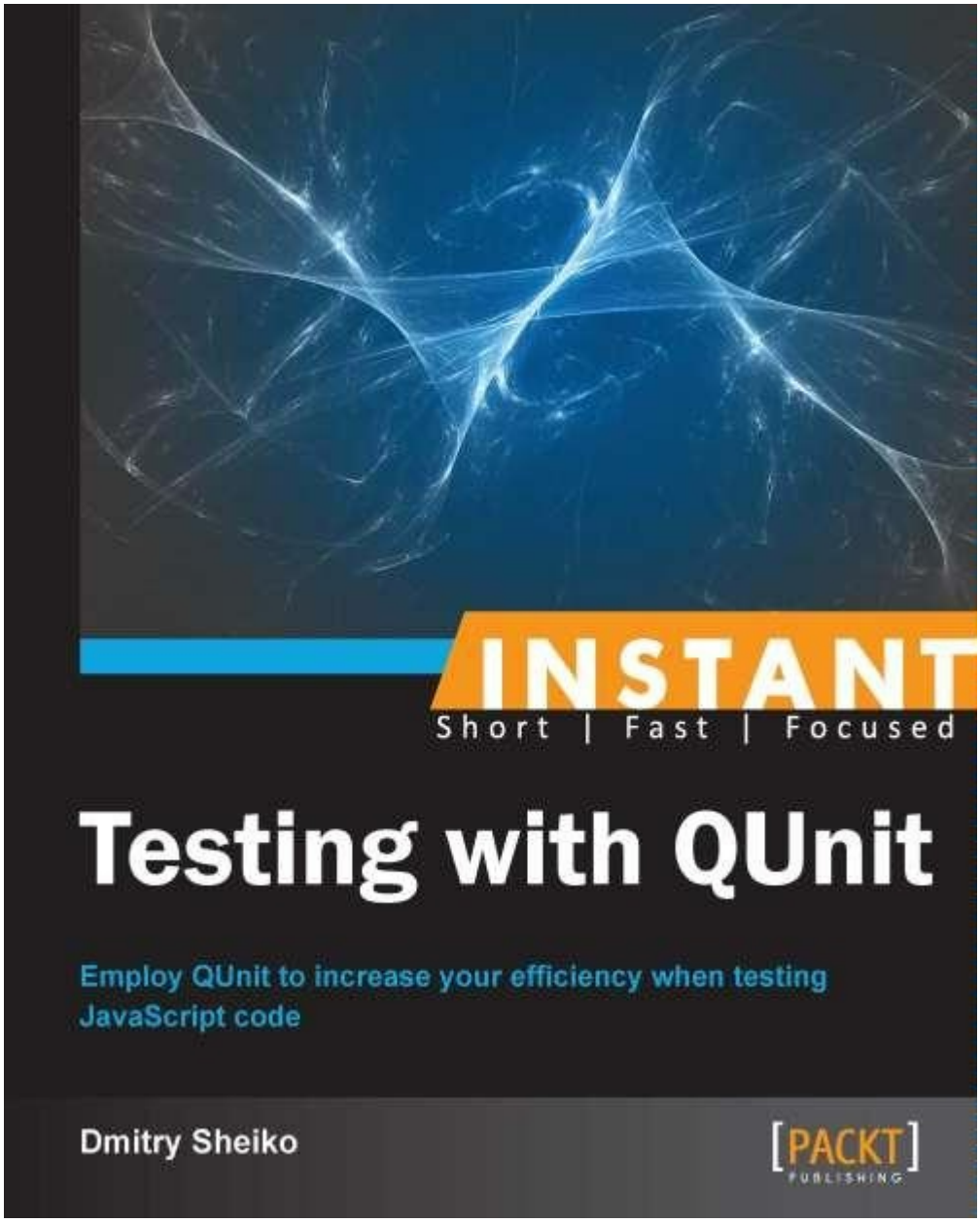

INSTANT
Short | Fast | Focused
Testing with QUnit
Employ QUnit to increase your efficiency when testing JavaScript code
Dmitry Sheiko
[PACKT]
PUBLISHING

Table of Contents

Instant Testing with QUnit
Credits
About the Author
About the Reviewer
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Instant Testing with QUnit
    Setting up QUnit (Simple)
        Getting ready
        How to do it
        How it works
        There's more
    Testing assertions (Simple)
        Getting ready
        How to do it
        There's more
    Writing a custom assertion plugin (Advanced)
        Getting ready
        How to do it
    Testing exceptions (Medium)
        Getting ready
        How to do it
    Testing asynchronous calls (Medium)
        Getting ready
        How to do it
        There's more
    Organizing test cases (Simple)
        Getting ready
        How to do it
    Using a shared setup (Medium)
        Getting ready
        How to do it

Testing user actions (Medium)
    Getting ready
    How to do it
Running Qnit in the console (Advanced)
    Getting ready
    How to do it
    There's more
Cross-browser-distributed testing (Advanced)
    Getting ready
    How to do it
Building a web project (Advanced)
    Getting ready
    How to do it
QUnit and CI – setting up Jenkins (Advanced)
    Getting ready
    How to do it
    There's more

Instant Testing with QUnit

Instant Testing with QUnit

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1230813

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-217-3

www.packtpub.com

Credits

Author

Dmitry Sheiko

Reviewer

Sachin G. Kulkarni

Acquisition Editor

Mary Nadar

Commissioning Editor

Govindan K.

Technical Editor

Dylan Fernandes

Sonali S. Vernekar

Copy Editor

Brandt D'Mello

Mradula Hegde

Project Coordinator

Amey Sawant

Proofreader

Stephen Copestake

Graphics

Abhinash Sahu

Production Coordinator

Zahid Shaikh

Cover Work

Zahid Shaikh

Cover Image

Valentina D'silva

About the Author

Dmitry Sheiko is a web developer, blogger, and open source contributor, living and working in the lovely city of Frankfurt am Main, Germany.

He started learning computer programming in the late eighties. Since the late nineties he has been developing web applications and tools. Currently, he works as a senior web developer for the leading game developer company Crytek GmbH.

This is Dmitry's first book, but he has been blogging for a decade. You can find his thoughts and solutions on such subjects as code quality, software architecture, JavaScript, Node.js, TypeScript, HTML5 API, CSS3, and PHP at www.dsheiko.com.

Dmitry's very first open source contribution was BCWB, an XSLT-based CMS in 2004. Since that time, he has been contributing quite actively. His latest works are available at https://github.com/dsheiko.

I would like to thank my beautiful wife, Olga, for her support and patience throughout this project.

About the Reviewer

Sachin G. Kulkarni is a freelancer and consultant who helps in building software for businesses to automate business activities. He has extensively worked on PHP/MySql and HTML5/CSS3 to bring out the best UI design and implement business needs. His latest passion is leveraging mobile development using JavaScript with PHP backend to produce feature-rich applications for his clients. Sachin uses development patterns and best practices to enjoy application development to its fullest extent. For many years, he has been involved in open source projects and gives open source advice to people. Please visit his blog at `sachinkulkarni.info` to know more about him.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browsers

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

JavaScript was first released about 20 years ago. It had not been taken seriously by a wide audience for many years. Today, it is a mature (and the most popular) programming language. Whatever is done for the Web, one way or another, relies on JavaScript. Furthermore, with the advent of HTML5-related technologies, the development focus is shifting even more towards rich internet application architecture. That means further expansion of sophisticated scalable solutions based on JavaScript. The server-side JavaScript is rapidly gaining its momentum, thereby giving automated testing of JavaScript.

QUnit is a freakishly easy-to-use JavaScript testing framework at the same time, it is quite powerful and flexible to be a good choice for unit and functional testing of any scale projects.

This book provides a step-by-step guide to mastering QUnit. It shows how to set up and use QUnit, how to automate cross-browser testing with QUnit, and how to benefit from QUnit in cooperation with continous integration tools.

The goal of this book is to help the reader make the most of QUnit in a short amount of time.

What this book covers

Setting up QUnit (Simple) explains setting up a testing environment and getting acquainted with the QUnit framework.

Testing assertions (Simple) describes a tutorial on QUnit assertion methods and popular QUnit assertion plugins.

Writing a custom assertion plugin (Advanced) explains leveraging QUnit callbacks for a custom assertion plugin.

Testing exceptions (Medium) describes custom exceptions in JavaScript and asserting exceptional behavior.

Testing asynchronous calls (Medium) describes asserting asynchronous callbacks and mocking XHR for in-unit testing.

Organizing test cases (Simple) examines QUnit facilities to keep tests logically organized.

Using shared setup (Medium) describes a way to share tasks among the tests of a group and keeping the testing environment clean between test executions.

Testing user actions (Medium) explains exercising functional testing on a simple calculator application by simulating end-user actions and asserting intended behavior.

Running QUnit in the console (Advanced) explains running QUnit tests in the command line by the PhantomJS headless browser. It also explains assigning QUnit testing tasks to Grunt and automating testing with Travis CI.

Cross-browser-distributed testing (Advanced) explains automating client-side cross-browser testing using the command-line tool Bunyip.

Building a web project (Advanced) explains adding QUnit testing tasks into the project build script.

QUnit and CI – setting up Jenkins (Advanced) explains setting up Jenkins continuous integration server to automate QUnit testing.

Who this book is for

This book would be handy for anyone working with JavaScript who is looking for a powerful, but easy-to-use testing framework.

The reader doesn't need to know any particular framework but at least basic principles of JavaScript and HTML.

The book will be most profitable if the reader has any background in automated testing.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Setup the build script as `build.xml` in the root of the project working directory."

A block of code is set as follows:

```
<?xml version="1.0"?>
<!DOCTYPE project>
<project name="tree"  basedir="." default="build">
<target name="build" description="runs QUnit
tests using PhantomJS">
  <!-- Clean up output directory -->
  <delete dir="./build/qunit/"/>
  <mkdir dir="./build/qunit/"/>
  <!-- QUnit Javascript Unit Tests -->
  <echo message="Executing QUnit Javascript Unit
Tests..."/>

</target>
</project>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
utils.subscribe( global, "load", function() {
             var calc =
document.getElementById("calc"),
                 num1 =
document.getElementById("num1"),
                 num2 =
document.getElementById("num2"),
                 sum =
document.getElementById("sum");

            QUnit.test( "Test that calc sums up typed in
numbers by click", function( assert ){
                num1.value = "5";
                num2.value = "7";
                utils.trigger( calc, "click" );
                assert.strictEqual( sum.value, "12" );
            });
  });
```

Any command-line input or output is written as follows:

```
phantomjs runner.js test-suite.html
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Enter Jenkins Dashboard page and click on **Add new job** link".

> ## Note
>
> Warnings or important notes appear in a box like this.

> ## Tip
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/support, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Instant Testing with QUnit

Welcome to Instant testing with QUnit!

For a long time, JavaScript wasn't taken seriously. However, with the advent of AJAX, it turned out that using JavaScript web pages can be brought to speed with the desktop software RIA Rich Internet application (http://en.wikipedia.org/wiki/Rich_Internet_application). The forthcoming server-side JavaScript proved that application can be written entirely in JavaScript. JavaScript-based applications started growing up and getting complex. Even if good developers work on them, any sophisticated software will still have defects. Test engineers do their best to catch these before the product is released, but it is hardly possible to detect all the flaws manually. Automated testing is the best way to increase the efficiency and coverage of software testing. Besides, it reveals any problem present in the early development stages, which cuts down the cost of fixing it drastically.

Regardless of the technologies used, the best development practices include automation of unit tests, integration tests, system tests, and functional tests. Unit tests verify if the smallest functional parts of the application work as intended in isolation. Integration tests check if the components of the project collaborate properly. System tests examine the whole application—usually for criteria such as usability, performance/load/stress, scalability, compatibility, and security—and functional tests validate that the application UI is fine from the user's perspective.

Currently, there are plenty of tools meant to achieve automated testing in JavaScript. One of the most popular testing frameworks is QUnit. It's sufficiently powerful, extremely flexible and, at the same time, surprisingly easy to start with.

This is a practical book showing how to achieve a beneficial automated testing of your JavaScript applications with QUnit and its plugins, how to engage QUnit for automated cross-browser testing, and how to utilize QUnit in conjunction with developer automation and continuous integration tools.

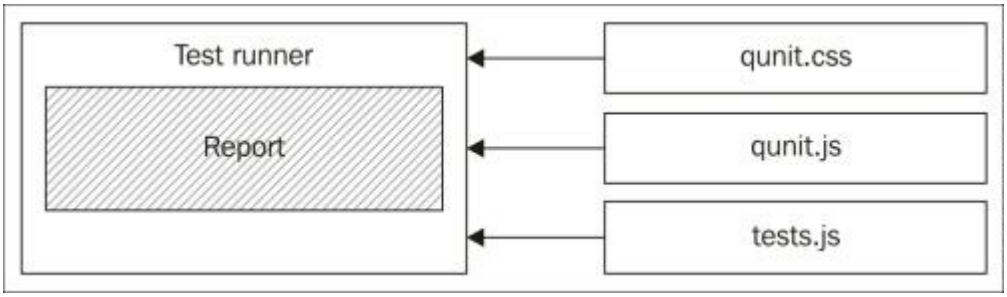
The book provides framework-agnostic Vanilla JS examples on all the contributed tasks that one can find easy to follow. The source code is categorized by the tasks and available for download at http://www.packtpub.com/support. Thus, whatever your previous experience or whatever libraries you work with, you can master the tasks in no time.

Setting up QUnit (Simple)

In this recipe, we will download the QUnit testing framework components and create a test runner HTML. We will examine how the QUnit test runner reports on cases of both succeeded and failed tests. We will also consider some of the best practices of the framework fi le structure.

Getting ready

The framework consists of the QUnit JavaScript module, the test runner HTML, and the style sheet. Test runner is a test suite executer. It loads other QUnit components and generates the report page. We include our tests either directly in the test runner HTML or in an external file. The component model is illustrated as follows:



We can simply copy-paste the test runner code from www.qunitjs.com as well as download the CSS and JavaScript components. In fact, it can work without downloading by using the following links to jQuery CDN in the test runner:

- http://code.jquery.com/qunit/qunit-git.css
- http://code.jquery.com/qunit/qunit-git.js

How to do it

Well, how does it work?

1. Create an `index.html` file with the following content:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Test Suite</title>
    <link rel="stylesheet"
href="http://code.jquery.com/qunit/qunit-
git.css" type="text/css" media="screen">
</head>
<body>
```

```
    <h1 id="qunit-header">Test Suite</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</div>

    <div id="sandbox">
        <!--Put here sandbox HTML if needed
-->
    </div>

    <script type="text/javascript"
src="http://code.jquery.com/qunit/qunit-
git.js"></script>
    <script type="text/javascript">
        QUnit.test( "Test example",
function( assert ){
            assert.ok( true );
        });
    </script>
</body>
</html>
```

2. Load HTML in your browser. You will see something similar to the following screenshot



So, what is on the page? First the title that we assigned in the HTML. Underneath follows a bar that is in calming green if all the tests succeeded and alerting red if any failed. In case the tests fail, we will see something similar to the following screenshot:



Further, we can see a toolbox. It contains the following options:

- The **Hide passed tests** checkbox, useful when you have plenty of tests running and want to focus only on those that failed.
- The **Check for Globals** checkbox, which allows you to test if the global object was modified during testing.
- The **No try-catch** checkbox, which triggers the running of the tests outside the try-catch block revealing native exceptions. This handy while debugging on legacy browsers.

Right below the toolbox, QUnit displays the user agent details. After that, we have a report area with the overall statistics and a list of per test results.

How it works

You may like to know what really happens behind the scenes. While the QUnit JavaScript module is being processed, it subscribes a handler on the `window.onload` event. The handler initializes the `QUnit` object and populates the placeholders of the test runner layout with the elements required to build the report page. Towards the end, it calls the `QUnit.start` method, which actually runs tests available at that time. Thus, by default, QUnit tries to run tests as soon as the page DOM is ready. If you work with modular JavaScript such as AMD and CommonJS, all the modules, including tests, are loaded asynchronously. The case requires the test runner to be manually started, which can be performed by using a configuration option as described in the following code:

```
QUnit.config.autostart = false;
require(
  [ "test-module1", "test-module2" ],
  function() {
    QUnit.start();
  }
);
```

There's more

In the previous example, we placed the dummy test code straight into the test runner HTML. fine for a short example, but doesn't suit a real project with a lot of tests. You will have to come up with a decent file structure. I would recommend creating a dedicated subdirectory (tests) in your project's `wwwroot`. There you can add subdirectories per project module under test. These subdirectories contain the code for module tests and test-related files, if required.

Usually, test runner is simply named `index.html` as it aggregates all the QUnit tests of the project. I would encapsulate different kinds of tests into respective runners such as `unit-tests.html`, `functional-tests.html`, and `acceptance-tests.html`.

Thus, we can run fast unit tests with every commit (for example, by a Continuous Integration server) and run functional and acceptance tests before any deployment. It also improves test readability when you keep functional tests out of acceptance tests.

```
app-wwwroot/
  ├── js/
  │   └── moduleA.js
  └── tests/
      ├── moduleA/
      │    ├── dummies.js
      │    ├── stubs.js
      │    ├── mocks.js
      │    ├── fixtures.js
      │    └── unit-tests.js
      └── unit-tests.html
```

Testing assertions (Simple)

Assertion is the main method used in testing. In this recipe, we will examine all the assertion methods that QUnit has. Besides, we will take a look at a few assertion methods provided by popular QUnit plugins.

Getting ready

It would be much more useful if while learning assertions, we test the real code. The following two helper functions are quite suitable to show discrete cases of the use of assertions:

```
var utils = (function() {
  "use strict";
  return {
    /**
     * Port of PHP trim function which differ
from EcmaScript 5 String.prototype.trim
     * Strip whitespace (or other characters)
from the beginning and end of a string
     * Without the second parameter, trim() will
strip these characters:
     * " " (ASCII 32 (0x20)), an ordinary space.
     * «\t» (ASCII 9 (0x09)), a tab.
     * "\n" (ASCII 10 (0x0A)), a new line (line
feed).
     * "\r" (ASCII 13 (0x0D)), a carriage return.
     * "\0" (ASCII 0 (0x00)), the NUL-byte.
     * "\x0B" (ASCII 11 (0x0B)), a vertical tab.
     * @param {string} str
     * @param {string} charlist
     * @return {string}
     */
    trim: function( str, charlist ) {
      charlist = charlist || " \t\n\r\0\x0B";
      return str.replace( new RegExp( "^[" +
charlist + "]+|[" + charlist + "]+$", "g" ), '' );
    },
    /**
     * Emulate class-based inheritance.
     * <example>
     * ClassA = function() {}
     * ClassB = utils.extendClass(ClassA, {
     *   prop: true
     * });
     * </example>
     */
    extendClass: function( SuperType, subType ) {
      var prop,
          F = function(){};
      F.prototype = new SuperType();
```

```
        F.prototype.constructor = SuperType;
        for ( prop in subType ) {
          if ( subType.hasOwnProperty( prop ) ) {
            F.prototype[ prop ] = subType[ prop ];
          }
        }
        return F;
      }
    };
  }( this ));
```

The first one is an analog of the PHP `trim` function. It removes any white spaces from the beginning and end of a supplied string. The second function implements class-based inheritance. So, you can create, for this function, a new class (`constructor`) by extending a given one (`SuperType`) with a specified object (`subType`).

How to do it

1. Define the test scope with the required assertions as follows:

```
QUnit.test( "Test title", function( assert ) {
  assert.<assertion-method>( comparison of
actual and expected values, "assertion
title" );
});
```

   Here, the scope of the test is defined comprises one or more assertions. QUnit provides assertion methods such as `equal`, `notEqual`, `strictEqual`, `notStrictEqual`, `ok`, `deepEqual`, and `notDeepEqual`.

> ## Note
>
> You may find plenty of examples made with the simplified interface on the Internet. QUnit methods are of course namespaced, but they are also exposed as shortcut functions on the global object. So, you can put the code in the following way:
>
> ```
> test( "Test title", function() {
>  <assertion-method>( comparison of actual
> and expected values, "assertion title" );
> });
> ```
>
> However, I would rather recommend against it. As you see, it's hardly short of code, but it may interfere with the global properties

added by other scripts and you won't be able to access plugin assertions this way.

2. Test `utils.trim` with equality assertions. If we pass a string as an argument to the `utils.trim` function from the preceding example, we expect this string with all leading and trailing whitespaces removed in the output. If the whitespaces still exist, the function doesn't work as intended. This can be tested by applying the common pattern as follows

```
QUnit.test( "Test utils.trim function",
function( assert ) {
  assert.eq ual( utils.trim(" .. "), "..",
"trims spaces" );
  assert.notEqual( utils.trim(" .. "), " ..
", "does not ignore trimming" );
});
```

3. Test `utils.trim` with strict equality assertions. Surely, we can't test properly this function with only one or two assertions, but nevertheless the case shows you can use the `equal` and `notEqual` methods. The methods `strictEqual` and `notStrictEqual` act in the same way as the last two except that they check for equality more strictly (let's say in a proper way). That's all about the difference between `==` and `===` JavaScript comparison operators. The first one is known as bad part JavaScript (See *JavaScript: The Good Parts, Douglas Crockford, O'Reilly Media*) and should be avoided unless one really means /fals comparison. Thus, we rather rewrite the `utils.trim` test the following code:

```
QUnit.test( "Test utils.trim function",
function( assert ) {
  assert.strictEqual( utils.trim(" .. "),
"..", "trims spaces" );
  assert.notStrictEqual( utils.trim(" .. "),
" .. ", "does not ignore trimming" );
});
```

4. Examine the risks of using weak equality assertions:

```
assert.notEqual( " \n\r\t ", false ); // fails
assert.notStrictEqual( " \n\r\t ",
false ); // passes
```

5. Test `utils.extendClass` with boolean assertions. First, we have to write a client code for the method. We define a pseudo class (constructor with some members added to the prototype) and pass it to `utils.extendClass`. We need to test it if the derived pseudo class makes an instance of what we extended. We cannot use equality assertion methods here, but QUnit provides the `ok` assertion method, which verifies if a supplied expression is true, and can likely be used.

```
QUnit.test( "Test utils.extendClass
function", function( assert ) {
  var ClassA = function() {
    this.propA = true;
  },
  ClassB = utils.extendClass( ClassA, {
    propB: true
  }),
  ClassC = utils.extendClass( ClassB, {
    propC: true
  }),
  obj = new ClassC();

  assert.ok( obj instanceof ClassA, "it is an
instance of ClassA" );
  assert.ok( obj instanceof ClassB, "it is an
instance of ClassB" );
  assert.ok( obj instanceof ClassC, "it is an
instance of ClassC" );

});
```

6. Test the two identical objects with deep equality assertions. At this point, we have tried the entire core QUnit assertion methods. However, QUnit API also includes helper methods `deepEqual` and `notDeepEqual` that allow you to assert on the equality of complex types such as arrays, objects, regular expressions, dates, and functions. This is useful when you need to compare two identical objects. Equality methods (`equal`/`strictEqual`) will never confirm the identical objects unless these are not references to the same object. On the contrary, `deepEqual` recursively compares members of both the supplied types. The following code is an example of the assertion that two arrays are identical:

```
QUnit.test( "Examine deepEqual",
function( assert ) {
  var dummy1 = {
    propA: true,
    propB: true
  },
  dummy2 = {
    propA: true,
    propB: true
  };

  assert.notStrictEqual( dummy1, dummy2,
"that is not refernces on the same object" );
  assert.deepEqual( dummy1, dummy2, "but they
are identical" );
});
```

There's more

Besides QUnit core assertion methods, you can also use the methods provided by numerous QUnit plugins. We will examine the most popular ones here.

Most xUnit frameworks have facilities for so-called **Parameterized Unit Tests** (**PUTs**). These tests are provided with a set of any number of test input values. During test exercising, these values are passed repeatedly as parameters to the test. Thus, the unit is tested with all the necessary input cases. The **Parameterize** plugin (https://github.com/AStepaniuk/qunit-parameterize) brings these facilities to QUnit.

Do you remember we were testing the `utils.trim` function? To make sure the string is trimmed correctly, in many cases we would need to write a lot of assertions repeating almost the same code all over the test scope. It can be simplified notably by using the Parameterize plugin. See the following example:

```
QUnit
        .cases([
            { str: ' string ', charlist: null,
expected : 'string' },
            { str: '  string  ', charlist: null,
expected : 'string' },
            { str: '\t\n\rstring  ', charlist:
null, expected : 'string' },
            { str: '||string|', charlist: "|",
expected : 'string' },

        ])
        .test("Test trim", function( params ) {
            var expected =
utils.trim( params.str, params.charlist );
            strictEqual( expected,
params.expected );
        });
```

If you need to assert that two float numbers are equal to some degree, it would be not so easy to achieve this in JavaScript. That is another "bad" part of the language. Consider the following code snippet:

```
var a = 1.1234,
    b = 1.1230;

console.log( ( Math.round( parseFloat( a ) *
1000 ) / 1000 ) ===
    ( Math.round( parseFloat( b ) * 1000 ) /
1000 ) );
```

On the other hand, the **Close** plugin (https://github.com/JamesMGreene/qunit-assert-close) provides a much more convenient way to do this as shown in the following code:

```
assert.close(actual, expected, maxDifference,
message);
assert.notClose(actual, expected, minDifference,
message);
```

Here `actual` and `expected` are the float numbers we compare, `maxDifference` / `minDifference` are the degree to which we compare them, and `message` is an optional description of the assertion. So, the test may look like the following code:

```
(function(){
    'use strict';
    var closeToPi = 3.141,
        notCloseToPi = 3.1;

    test('Test closeToPi is close to PI till
1/1000', function( assert ) {
        assert.close( closeToPi, Math.PI, 0.001 );
    });
    test('Test notCloseToPi isn\'t close to PI
till 1/1000', function( assert ) {
        assert.notClose( 3.1, Math.PI, 0.001 );
    });

}( this ));
```

Sometimes, we have to make sure that functions are being invoked in a defined order for example, the AMD-compatible module loader works asynchronously but must resolve dependencies when they are sequenced. The **Step** plugin (https://github.com/JamesMGreene/qunit-assert-step) allows us to assert the proper sequence in which the code should perform. The following example verifies that the `bar` function is called prior to the `foo` function:

```
(function(){
    'use strict';
    test('Test "foo" is invoked after "bar"',
function( assert ) {
         function foo() {
            assert.step( 2 );
         }
         function bar() {
            assert.step( 1, "bar is invoked
first" );
         }
         bar();
```

```
            foo();
    });
}( this ));
```

Nowadays, the canvas element is widely used to render diagrams, simple animation, and browser-based games. Obviously, it means a lot of code testing. However, any assertion on a canvas region doesn't seem an easy task. The **Canvas** plugin (https://github.com/JamesMGreene/qunit-assert-canvas) provides the assertion method to test individual pixel values in a canvas. In the following example, we define a canvas area on the page and draw a black rectangle. Then, we assert a dot on the rectangle that has black color.

```
<div id="sandbox"><canvas width="5"
height="5"></canvas></div>
<script type="text/javascript"
src="http://code.jquery.com/qunit/qunit-
git.js"></script>
<script type="text/javascript"
src="../vendors/Canvas/qunit-assert-
canvas.js"></script>
<script type="text/javascript">
(function( window ){
    'use strict';
    var document = window.document,
        canvas =
document.getElementById('sandbox').firstChild,
        context = canvas.getContext('2d');

    test('Test the dot at 0,0 is black',
function( assert ) {
      context.fillStyle = 'rgba(0, 0, 0, 0)';
      context.fillRect( 0, 0, 10, 10 );
      assert.pixelEqual( canvas, 0, 0, 0, 0, 0,
0 );
    });

}( this ));
</script>
```

HTML5 tolerates HTML-like looseness such as non-closed elements and non-escaped attribute values. This makes it difficult to compare two different HTML strings that are treated equally by a browser. The HTML plugin (https://github.com/JamesMGreene/qunit-assert-html) solves the issue in the following way:

```
(function( global ){
        'use strict';

        test('Test HTML strings are equal',
function(assert) {
```

```
            assert.htmlEqual('<B
TITLE=test>test</B>', '<b title="test">test</b>');
        });


}( this ));
```

Writing a custom assertion plugin (Advanced)

This recipe will guide you on the basis of creating QUnit plugins. We will create a custom assertion method and add it to the common QUnit assertion service.

Getting ready

QUnit is easy to start with as it provides only a few necessary assertion methods. Nevertheless, you can have as many methods as you wish by using QUnit plugins. Plenty of plugins are already available on the Internet. Besides, you may need your own custom plugin. Luckily, QUnit has a Callback API, which makes it easy. You can simply register your own handler on the entry and exit points of every testing scope (test suite, module, and test) or subscribe a handler function to the event that happens every time assertion performed in a test; for instance, PhantomJS (www.phantomjs.org) is widely used to run QUnit tests in the console. However, the QUnit output test results as an HTML report in DOM, which doesn't work with the command line. The PhantomJS runner JavaScript (NodeJS) utility solves this issue by translating the QUnit output into a format compatible with the command-line interface. It registers a callback with the `QUnit.log` method to collect and store results on every assertion. It also subscribes a handler on the test suite complete event with `QUnit.done`, which outputs the accumulated results in the required way.

However, if you write just an assertion method, you may do without the callbacks API in most cases. You can simply add your method in the `QUnit.assert` object.

How to do it

1. Write an assertion method. Imagine that we need a convenient method for asserting that a supplied element is visible. The method is very simple—it only checks if the element is hidden via style properties visibility and display.

```
function isVisible( node ) {
  return node.style.visibility !== "hidden" &&
         node.style.display !== "none";
}
```

2. Extend the `QUnit.assert` object with this new method. As `QUnit` already provides the `extend` helper, we don't need to use a third party library or write our own helper for extending an object to do it.

```
(function( global ){
    "strict mode";
    var QUnit = global.QUnit;
    QUnit.extend( QUnit.assert, {
        /**
         * Check if a supplied element is
visible
         *
         * @example
assert.isVisible(document.getElementById("#id"));
         * @param {object} HTMLElement
         */
        isVisible: function( node ) {

 this.ok( node.style.visibility !
== "hidden" &&
              node.style.display !== "none" )
        }
    });
}( this ));
```

3. Save this code in the `qunit-assert-is-visible.js` file.
4. Provide the test runner HTML as follows:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Plugin Usage Example</title>
    <link rel="stylesheet"
href="http://code.jquery.com/qunit/qunit-
git.css" type="text/css" media="screen">
</head>
<body>
    <h1 id="qunit-header">Plugin Usage
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</div>

    <div id="sandbox">
      <div id="myTarget"><!-- --></div>
    </div>

    <script type="text/javascript"
src="http://code.jquery.com/qunit/qunit-
git.js"></script>
    <script type="text/javascript"
src="../js/qunit-assert-is-
visible.js"></script>
    <script type="text/javascript">
        "strict mode";
```

```
        QUnit.test( "Test myTarget is
visible", function( assert ){
          var node =
document.getElementById("myTarget");
            assert.isVisible( node );
        });
    </script>
</body>
</html>
```

5. Load the test runner in a browser, and examine the results shown in the following screenshot:



Testing exceptions (Medium)

In this recipe, we will learn how to handle the function's exceptional behavior, how to define custom exceptions, and how to test exceptions are really thrown for invalid input and these are exceptions of the intended types.

Getting ready

We've learnt already how to test whether or not a function performs the intended business logic. But what about its exceptional behavior? Let's say we supply invalid input parameters. We expect the function to warn us about them. The best practice would be to check the parameters have been passed onto the function entry point and thrown exceptions if any of the parameters are invalid. We need a way to test this logic as well.

Do you remember the `utils.trim` function example? We have to modify it now for checking the validity of the input parameters:

```
var trim = function( str, charlist ) {

  if ( typeof str !== "string" ) {
    throw new this.InvalidTypeException("str
argument must be a string");
  }
  if ( charlist && typeof charlist !== "string" )
{
    throw new this.InvalidTypeException("charlist
argument must be a string");
  }
  if ( !str.length ) {
    throw new this.InvalidReferenceException("str
argument must be empty");
  }

  charlist = charlist || " \t\n\r\0\x0B";
  return str.replace( new RegExp( "^[" + charlist
+ "]+|[" + charlist + "]+$", "g" ), '' );
}
```

Well, the function throws exception whenever it detects an invalid input but these are custom exceptions that must be defined. This can be done by declaring new error object constructors that inherit from standard JavaScript errors. Please find the implementation for this in the following example:

```
"strict mode";
var utils = (function( global ) {
  "use strict";
  return {
```

```
    /**
     * Port of PHP trim function
     * @param {string} str
     * @param {string} charlist
     * @return {string}
     */
    trim: function( str, charlist ) {
    // function body from the example above
   },
    /**
    * @constructor
    */
    InvalidReferenceException:
function( message ) {
       this.name = "InvalidReferenceException";
       this.message = message ||
"InvalidReferenceException thrown";
    },
    /**
    * @constructor
    */
    InvalidTypeException: function( message ) {
       this.name = "InvalidTypeException";
       this.message = message ||
"InvalidTypeException thrown";
    }
  };
}());

 // Inherit from ReferenceError
utils.InvalidReferenceException.prototype = new
ReferenceError();
utils.InvalidReferenceException.prototype.constructor
= utils.InvalidReferenceException;
// Inherit from TypeError
utils.InvalidTypeException.prototype = new
TypeError();
utils.InvalidTypeException.prototype.constructor
= utils.InvalidTypeException;
```

How to do it

1. Define the test scope and the assert expected exception is thrown. In this case, QUnit provides the `throws` method, which can be described with the following code:

```
QUnit.test( "Test title", function( assert ) {
  assert.throws( "callback throwing
exception, expected exception", "assertion
title" );
});
```

2. Test whether `utils.trim` validates the input parameters and throws the intended exceptions:

```
QUnit.test( "Test utils.trim contract
violation", function( assert ){
  assert.throws( function() {
      utils.trim("");
    }, utils.InvalidReferenceException,
    "str agrument must not be empty"
  );
  assert.throws( function() {
      utils.trim( 1 );
    }, utils.InvalidTypeException,
    "str agrument must be a string"
  );
  assert.throws( function() {
      utils.trim( "string", 1 );
    }, utils.InvalidTypeException,
    "charlist agrument must be a string"
  );
});
```

3. Load the test runner in a browser and examine the results shown in the following screenshot:



Testing asynchronous calls (Medium)

In this recipe, we will see how QUnit deals with asynchronous callbacks. We will write an assertion test for an asynchronously called function. We will also go through an example of testing the XHR-associated function by mocking the transfer method.

Getting ready

One of the greatest features of JavaScript is its ability to call functions asynchronously. You can easily make multiple functions that execute simultaneously as follows:

```
window.setTimeout( function() {
    yourFunc1( param1, paramN );
}, 0);
```

As you understand, these functions cannot be tested the way we did before. We cannot assert on the function straight after it is launched. Otherwise, we risk asserting on the call before it is completed. We will encounter the same problem while testing handlers subscribed to events and even while using JavaScript objects associated with events;—for instance, XHR.

How to do it

1. Define the test scope. For testing asynchronously called functions, QUnit provides the `asyncTest` method, which in contrast to the test method receives an additional parameter with a number of expected assertions. Thus, QUnit will set the test as complete only when the supplied number of assertions is performed.

```
  /**
   * Test async. called function
   * @param {string} test title
   * @param {number} number of expected
assertions
   * @param {function) test scope
   */
QUnit.asyncTest( "Test title", 1,
function( assert ){
});
```

2. Assert within the asynchronous callback. Since we assert asynchronously, we have to call the `QUnit.start` method as soon as

the last assertion is done. This will let QUnit know when it can jump to the next test.

```
    /**
     * Test async. called function
     * @param {string} test title
     * @param {number} number of expected
assertions
     * @param {function) test scope
     */
    QUnit.asyncTest( "Test async control", 1,
function( assert ){
        window.setTimeout( function() {
          assert.ok( true, "captured" );
          QUnit.start();
        }, 0);
    });
```

3. Load the test runner in a browser, and examine the results shown in the following screenshot:



There's more

The described approach works fine with the test-friendly code. I mean the code where the asynchronous call accepts test callback injection or fires an event on which a handler is subscribed. When you are working with black-box code, you have no way to assert on asynchronous calls except mock the transfer utility whose interface you are familiar with; for example, while unit-testing a function that relies on XHR, we have to mock the helper performing the request. We do it anyway as it includes an across network communication and gets out of the unit scope. Let's say the function under test uses the jQuery.get method. We know its interface and can override it with our own method, which does nothing but report to QUnit. You can see this in the following example:

```
var
  /**
   * Object representing an abstract RSS Feed
   * @constructor
   */
  RssFeed = function() {
```

```
    var xml;
    return {
      /**
       * Load data from a remote source
       * @param {string} url
       */
      loadData: function( url ) {
          $.get( url, function( rsp ){
            xml = rsp;
          });
      },
      /**
       * Get last loaded data
       */
      getData: function() {
        return xml;
      }
    };
  };


  QUnit.asyncTest( "Test RssFeed.loadData
requests remote source", 1, function( assert ){
      var feed = new RssFeed();

       /**
        * Mock XHR helper just for the case
        * @param {string} url
        * @param {function} callback
        */
       $.get = function( url, callback ) {
         window.setTimeout( function(){
               callback("data");
               assert.ok( true, "data
requested" );
               QUnit.start();
           }, 0 );
       };
      feed.loadData("http://google.com");
  });
```

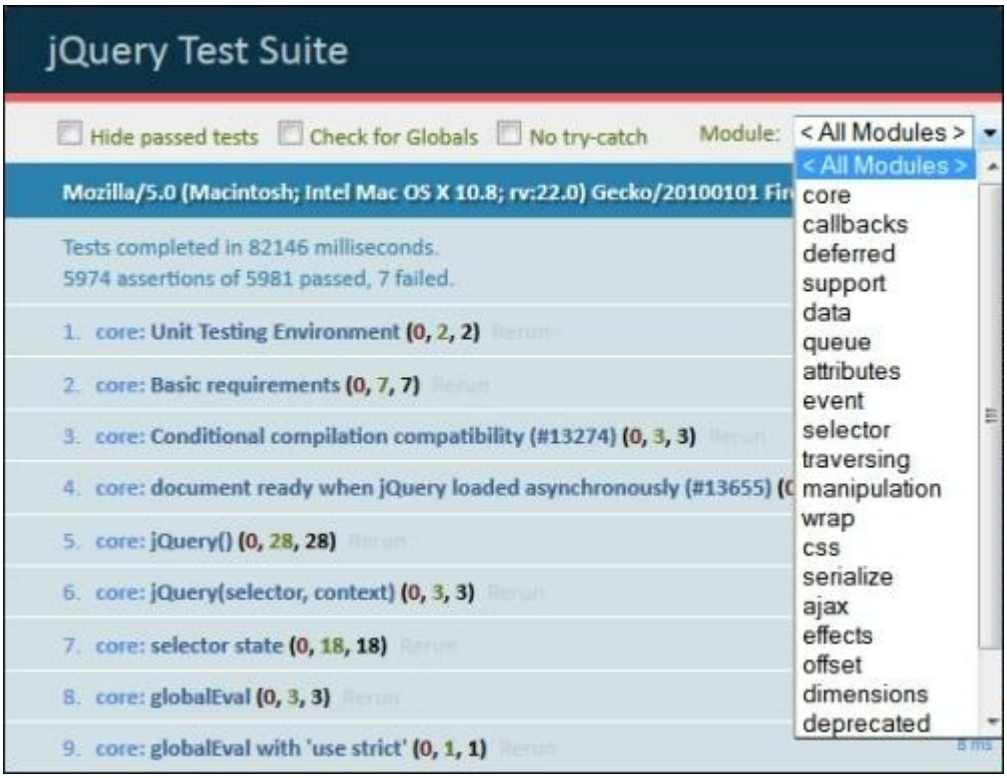
The `RSSFeed` constructor produces an object that can load data from a remote source. Data is being loaded asynchronously using `$.get`. To test the method, we override `$.get` with our own method. Our implementation invokes a supplied callback as expected from `jQuery.get` and does it asynchronously by using `setTimeout`. Towards the end, it makes the assertion and notifies QUnit that the test is complete.

Organizing test cases (Simple)

In this recipe, we will learn how to keep the tests logically organized by using the `QUnit.module` method.

Getting ready

While working on a real project, the number of tests can be pretty huge; for example, the jQuery test suite contains about 6,000 tests shown in the following screenshot:



Navigation on such lists without any grouping abilities would be a tough task. Luckily, QUnit provides the module method, which allows us to organize the tests and run a specific group of tests individually.

Imagine we have an application handles user interaction with the Sign In and Password Reset forms. We don't need any implementation details for now, but a high-level concept as follows:

```
var ResetPasswordForm = function() {
    return {
      /** Render and synchronize UI */
      init: function() {
      },
      /** Show panel */
      show: function() {},
```

```
    /** Hide panel */
    hide: function() {},
    /** Validate form input */
    validateInput: function() {}
  };
},
SignInForm = function() {
  return {
    /** Render and synchronize UI */
    init: function() {
    },
    /** Show panel */
    show: function() {},
    /** Hide panel */
    hide: function() {},
    /** Validate form input */
    validateInput: function() {}
  };
}
```

Thus, we have two object constructors and three methods each that must be tested. Let's split up the test into two groups, one for the SignIn form and one for Rest PasswordForm.

How to do it

1. Define a test group (`module`) for the `SignInForm` constructor.

```
QUnit.module("SignInForm");
```

2. Put underneath all the tests relevant to the group.

```
QUnit.test( "show method", function( assert ){
    // Stub assertion
    assert.ok( true );
  });
  QUnit.test( "hide method",
function( assert ){
    // Stub assertion
    assert.ok( true );
  });
  QUnit.test( "validateInput method",
function( assert ){
    // Stub assertion
    assert.ok( true );
  });
Repeat the flow for the ResetPasswordForm
constructor.
// Define the group
  QUnit.module("ResetPasswordForm");
  QUnit.test( "show method",
function( assert ){
    // Stub assertion
    assert.ok( true );
```

```
  });
  QUnit.test( "hide method",
function( assert ){
    // Stub assertion
    assert.ok( true );
  });
  QUnit.test( "validateInput method",
function( assert ){
    // Stub assertion
    assert.ok( true );
  });
```

3. Load the test runner in a browser and examine the results shown in the following screenshot:



Using a shared setup (Medium)

In this recipe, we will guide you on test group operations such as shared preliminary tasks and how to keep the testing environment clean between test executions.

Getting ready

While writing more and more tests, you will realize that to get clear results, you need to repeatedly set up the environment in a specific state before running a test and restore its original state afterwards. This is a time-consuming task that involves code duplication. Most testing frameworks, including QUnit, support the so-called shared setup code. Thus, you can define two callbacks for a group of tests. The first one (`setup`) takes responsibility for recreating the intended environment state prior to every test of the exercising group. The second (`teardown`) cleans up the changes made to the environment after every test is done.

To consider the advantages of the approach, let's take a sample module for testing:

```
/**
   * @constructor
   * @returns {object}
   */
  var SignInForm = function( boundingBox ) {
    return {
      /** Show panel */
      show: function() {
        boundingBox.style.display =
"inline-block";
      },
      /** Hide panel */
      hide: function() {
        boundingBox.style.display = "none";
      }
    };
  };
```

A constructor producing `SignInForm` can hide or show up. During initialization, the object (`SignInForm`) binds to the elements. An instance of `SignInForm` expects of HTML layout an element to bind to. The `SignInForm` methods are permitted to modify DOM within the bounding box element. Theoretically, every second test starts on a diverse DOM that may affect test results. We can prevent the problem by adding a clean bounding box to the DOM tree on the test setup and removing it with teardown.

How to do it

1. Define a test group (`module`) for the `SignInForm` constructor as follows:

```
QUnit.module("SignInForm");
```

2. The second parameter `QUnit.module` of the module method expects an object called shared setup confi guration.

```
  QUnit.module("SignInForm", {
    setup: function() {
    },
    teardown: function() {
    }
});
```

3. Add the code appending the bounding box element to the DOM tree on setup and the code removing it on teardown as follows:

```
QUnit.module("SignInForm", {
    // set up testing environment
    setup: function() {
      this.fixture =
document.createElement("div");

 document.getElementsByTagName('body')
[ 0 ].appendChild( this.fixture );
    },
    // restore the initial state
    teardown: function() {

 document.getElementsByTagName('body')
[ 0 ].removeChild( this.fixture );
    }
});
```

4. Move the `SignInForm` instance creation code to the shared setup as it is repeatedly required by every test of the group.

```
..
    setup: function() {
      this.fixture =
document.createElement("div");

 document.getElementsByTagName('body')
[ 0 ].appendChild( this.fixture );
      this.form = new
SignInForm( this.fixture );
    },
..
```

5. Write tests to assert on the `SignInForm` methods.

```
// Test hide method
  QUnit.test( "hide method",
function( assert ){
    this.form.hide();

 assert.strictEqual( this.fixture.style.display,
"none" );
  });
  // Test show method
  QUnit.test( "show method",
function( assert ){
    this.form.show();

 assert.notStrictEqual( this.fixture.style.display,
"none" );
});
```

6. Load the test runner in a browser and examine the results shown in the following screenshot:

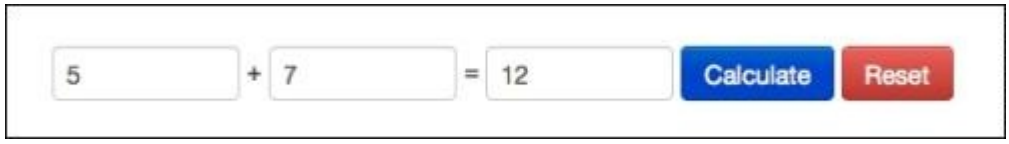


Testing user actions (Medium)

In this recipe, we will write a simple calculator widget. By testing, we will simulate end-user actions and assert the intended application behavior.

Getting ready

Before jumping on the test, we definitely need an application expecting user actions. Let it be a simple calculator that cannot do complex operations but will sum up two of the supplied numbers, as seen in the following screenshot:



To make it work, we subscribe a handler to the `click` event on the **Calculate** button. The handler takes the values of the first and second fields, sums them up, and puts the result in the third field. The implementation of the code is as follows:

```
"strict mode";

this.utils = {
    /**
     * Fire a suplied event on a given element
     * @param {object} el instance of HTMLElement
     * @param {string} eventName
     */
    trigger: function( el, eventName ) {
        var e = document.createEvent("Event");
        e.initEvent( eventName, true, true );
        el.dispatchEvent( e );
    },
    /**
     * Subscribe handler for event on a supplied
element
     * @param {object} el instance of HTMLElement
     * @param {string} eventName
     * @param {function} handlerCb
     */
    subscribe: function( el, eventName,
handlerCb ) {
        el.addEventListener( eventName,
function( e ) {
            handlerCb( e );
        }, false );
    }
};
```

```
(function( global ){
  var document = global.document,
      utils = global.utils;

  utils.subscribe( global, "load", function() {

 utils.subscribe( document.getElementById("calc"),
"click", function( e ) {
      var num1 = document.getElementById("num1"),
          num2 = document.getElementById("num2"),
          sum = document.getElementById("sum");

      e.preventDefault();
      sum.value = parseInt(num1.value, 10) +
        parseInt(num2.value, 10);
    });
  });
}( this ));
```

Let's save the source code into the `calc.js` file and load it from the HTML layout:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Calc</title>
        <link
href="//netdna.bootstrapcdn.com/twitter-
bootstrap/2.3.1/css/bootstrap-
combined.min.css" rel="stylesheet">
        <style>
          .container {
            margin: 32px;
          }
        </style>
    </head>
    <body>
    <div class="container">
      <form class="form-inline">
        <input id="num1" type="text"
class="input-small" placeholder="Summand">
        <span>+</span>
        <input id="num2" type="text"
class="input-small" placeholder="Summand">
        <span>=</span>
        <input id="sum" type="text"
class="input-small" placeholder="Sum">
        <button  id="calc" type="button"
class="btn btn-primary">Calculate</button>
        <button type="reset" class="btn
btn-danger">Reset</button>
      </form>
```

```
    </div>
    <script src="./js/calc.js"></script>
    </body>
</html>
```

Now, we run this HTML in a browser and test it manually. With numbers typed in, the button clicked, and the result received, we are now ready to write an automated test for the actions we just performed.

How to do it

1. Subscribe a handler to the DOM-ready event and get references on the involved elements. To mimic a user typing numbers and clicking the button, we need to refer to the inputs and the button. However, elements will be available only when the DOM tree is built. So, we obtain references on the elements in the handler, which is called on the DOM-ready event. For that, we use the helper introduced in the sample application as follows:

```
utils.subscribe( global, "load", function() {
            var calc =
document.getElementById("calc"),
                num1 =
document.getElementById("num1"),
                num2 =
document.getElementById("num2"),
                sum =
document.getElementById("sum");
});
```

2. Define a test scope and simulate the user behavior. Within the scope, we assign test values to inputs and trigger the click event. Since the application responds to the click event immediately, we can synchronously assert that the value in the result field equals the sum of numbers we supplied.

```
utils.subscribe( global, "load", function() {
            var calc =
document.getElementById("calc"),
                num1 =
document.getElementById("num1"),
                num2 =
document.getElementById("num2"),
                sum =
document.getElementById("sum");

            QUnit.test( "Test that calc sums
up typed in numbers by click",
function( assert ){
                num1.value = "5";
                num2.value = "7";
```

```
                utils.trigger( calc,
"click" );

 assert.strictEqual( sum.value, "12" );
            });
  });
```

3. Load the test runner in a browser and examine the results shown in the following screenshot:



Running Qnit in the console (Advanced)

In this recipe, we will run QUnit tests from the command line by using the PhantomJS headless browser.

Getting ready

We've learnt how to run QUnit in a browser and it's suitable in the beginning. However, in an advanced development cycle, the tests are being executed by a tool such as a piece of build automation software or a **VCS** (**version control system**). So, we need to run our tests in the console. That can be done with QUnit tests using a headless browser.

Let's take **PhantomJS** (https://github.com/ariya/phantomjs). First of all, we have to install the package. That's quite easy to do with a package manager in Mac OS or in Unix/Linux. For an instance using Homebrew on Mac OS, you can do it like this:

```
brew install phantomjs
```

On Windows, you can simply download the executable file from http://phantomjs.org/download.html and add its location to the Windows path as follows:

1. Hold *Win* and press *Pause*.
2. Click on **Advanced System Settings**.
3. Click on **Environment Variables**.
4. Append `C:\phantomjs` to the *Path* variable.

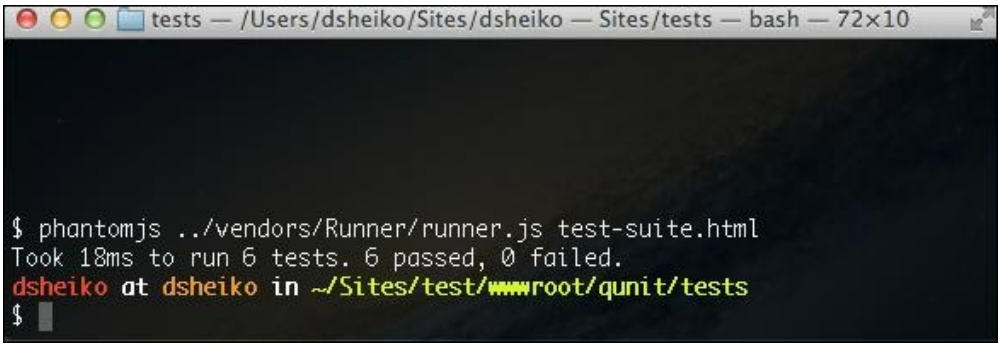
We will also need the PhantomJS Runner script (https://github.com/ariya/phantomjs), which plugs in to QUnit to translate the report output into command-line-compatible format.

How to do it

1. Install PhantomJS.
2. Download PhantomJS Runner and save it as `runner.js`.
3. Run the test suite with the following command:

```
phantomjs runner.js test-suite.html
```

4. Examine the results shown in the following screenshot:



There's more

You may find plenty of projects on GitHub (https://github.com) using the **Travis CI** (https://travis-ci.org) continuous integration server. Travis CI reports to the project members when somebody breaks the build. It means if you make some changes in the project code and use the push command to commit to GitHub, Travis CI can automatically check if the changed code still adheres to the style guide and linter requirements, if the unit test still passes, and so on. How does it work? With every code commit, Travis CI pulls down the project's working directory from GitHub and runs the **Grunt** task runner (www.gruntjs.com). Grunt performs the tasks supplied in the confi guration fi le (`Gruntfile.js`). Among other tasks, you can assign QUnit tests. Travis CI stores the test results in the build report and notifi es the commit if the build fails.

Let's see how we can take advantage of this approach:

1. Install Node.js. For Mac OS X, we can again recourse to Homebrew:

   ```
   brew install node
   ```

   Installation instructions for other platforms can be found on this page:

   https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

2. Install Grunt. It can be easily done now that we have the `Node.js` package manager:

   ```
   npm install -g grunt-cli
   ```

3. Move to the project working directory.
4. Configure the project with `package.json`.

   ```
   {
     "name": "project-name",
   ```

```
  "description": "Project description",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-qunit": "~0.2.1",
    "qunitjs": "1.x"
  }
}
```

5. Install the required dependencies:

```
npm install
```

6. Set up the `Grunt` task to fire up the QUnit test runner in `test-suite.html`:

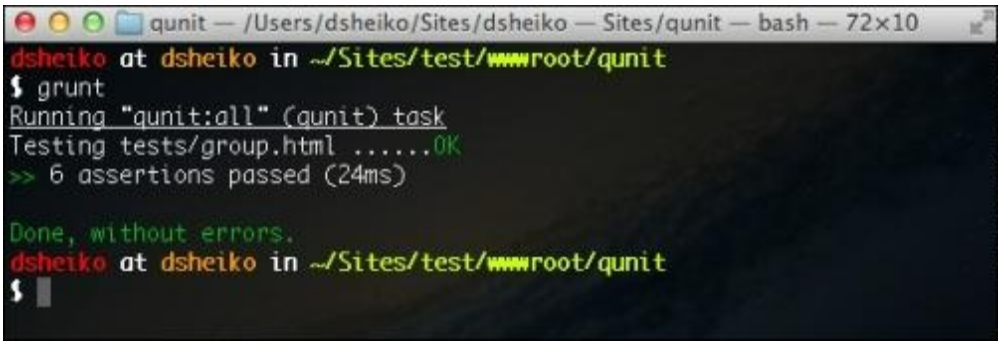
```
module.exports = function(grunt) {
  grunt.loadNpmTasks("grunt-contrib-qunit");

  grunt.initConfig({
    qunit: {
      all: ["tests/test-suite.html"]
    }
  });

  grunt.registerTask("test", ["qunit"]);
  grunt.registerTask("default", ["test"]);

};
```

7. Examine the results shown in the following screenshot:



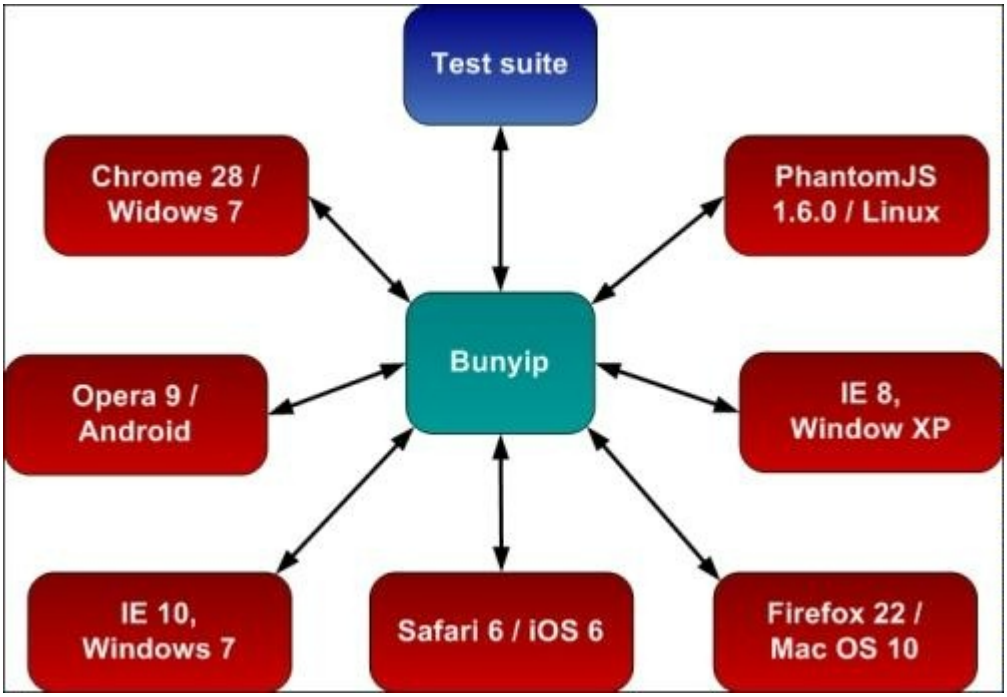
Cross-browser-distributed testing (Advanced)

This recipe will guide you on automating client-side cross-platform/browser testing using the command-line tool **Bunyip**.

Getting ready

In contrast to the server-side software, JavaScript applications are being executed on the client side and therefore depend on the user browser. Normally, project specification includes the list of the browsers and platforms that the application must support. The longer the list, the harder is cross-browser-compatibility testing. For example, jQuery supports 13 browsers on different platforms. The project is fully tested in every declared environment with every single commit. That is possible thanks to the distributed testing tool TestSwarm (swarm.jquery.org). You may also hear of other tools such as Js TestDriver (code.google.com/p/js-test-driver) or Karma (karma-runner.github.io). We will take Bunyip (https://github.com/ryanseddon/bunyip) as it has swiftly been gaining popularity recently.

How does it work? You launch the tool for a test runner HTML and it provides the connect end-point (`IP:port`) and launches a locally installed browser if configured. As soon as you fire up the address in a browser, the client is captured by Bunyip and the connection is established. With your confirmation, Bunyip runs the tests in every connected browser to collect and report results. See the following figure:



Bunyip is built on top of the Yeti tool (www.yeti.cx) works with **YUI Test**, QUnit, **Mocha**, **Jasmine**, or DOH. Bunyip can be used in conjunction with BrowserStack. So, with a paid account at BrowserStack (www.browserstack.com), you can make Bunyip run your tests on hundreds of remotely hosted browsers.

To install the tool type in the console as follows:

```
npm install -g bunyip
```

Here, we recourse to the `Node.js` package manager is part of `Node.js`. So if you don't have `Node.js` installed, find the installation instructions on the following page:

https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

Now, we are ready to start using Bunyip.

How to do it

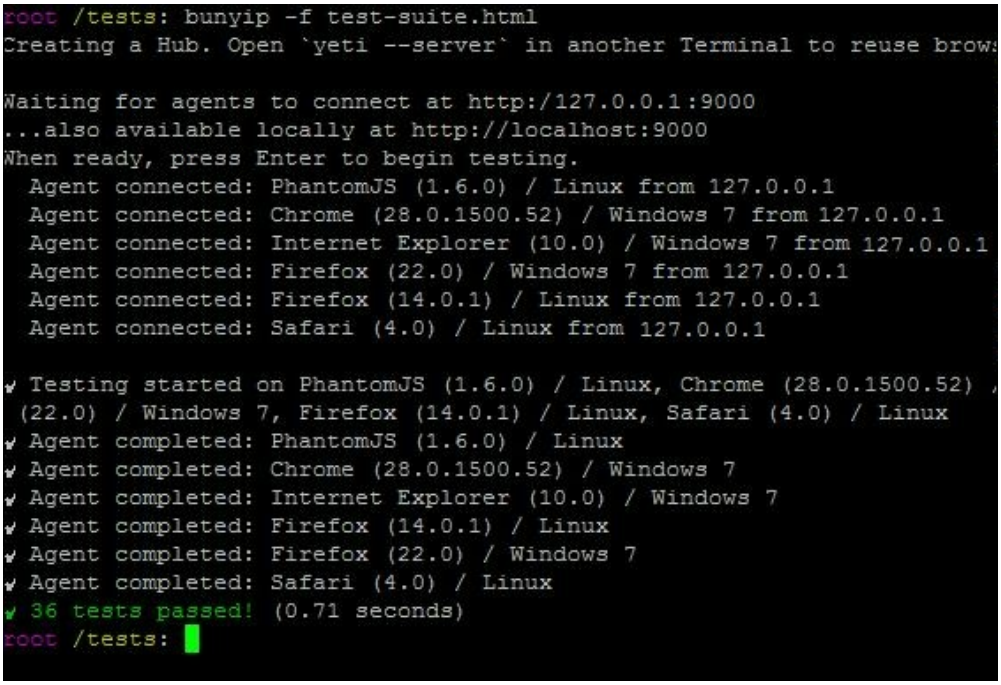
1. Add to the QUnit test suite (`test-suite.html`) the following configuration option to prevent it from auto-starting before the plugin callback is set up:

```
if (QUnit && QUnit.config) {
    QUnit.config.autostart = false;
}
```

2. Launch a Yeti hub on port 9000 (default configuration) and use `test-suite.html`.

```
bunyip -f test-suite.html
```

3. Copy the connector address (for example, `http://127.0.0.1:9000`) from the output and fire it up in diverse browsers. You can use Oracle VirtualBox (www.virtualbox.org) to launch browsers in virtual machines set up on every platform you need.
4. Examine the results shown in the following screenshot:



Building a web project (Advanced)

While working with a JavaScript project build may sound odd, it's quite relevant here. Using build , we can perform a bunch of project-related tasks with a single command. It's common practice nowadays to run tasks such as JavaScript linting, code standard validation, CSS preprocessing, and API documentation update with a build script, manually or using continuous integration tools. This recipe shows how the QUnit testing task can be added to the project build script.

Getting ready

For this task, we will use **Apache Ant (** http://ant.apache.org/ **)** for build automating. On Mac OS X, Ant is available by default. On Unix/Linux, it is available in ports and you can use a package manager to install Ant. As for Windows you can find the installation instructions in the official manual at ant.apache.org/manual/index.html.

Since Ant runs QUnit tests on the command line, we will use PhantomJS as it is discussed in . However, if we are to satisfy third-party software executing the build script such as continuous integration tools, we need to make the QUnit report in a specific format. Most tools accept **JUnit XML**. We can translate the QUnit output to JUnit format with a plugin called **JUnit Logger** (https://github.com/jquery/qunit-reporter-junit). So, we add to the test runner HTML the plugin JavaScipt module and the following configuration option:

```
QUnit.jUnitReport = function(report) {
console.log(report.xml);
};
```

But here pops up another problem. The PhantomJS Runner script now breaks JUnit XML with its own report version. You can simply comment out `console.log` occurrences in the `QUnit.done` and `QUnit.testDone` callback functions at the end of `runner.js` and save it, say as `runner-muted.js`.

How to do it

1. Set up the build script as `build.xml` in the root of the project working directory:

```
<?xml version="1.0"?>
<!DOCTYPE project>
<project name="tree"  basedir="."
default="build">
```

```
<target name="build" description="runs QUnit
tests using PhantomJS">
  <!-- Clean up output directory -->
  <delete dir="./build/qunit/"/>
  <mkdir dir="./build/qunit/"/>
  <!-- QUnit Javascript Unit Tests -->
  <echo message="Executing QUnit Javascript
Unit Tests..."/>

</target>
</project>
```

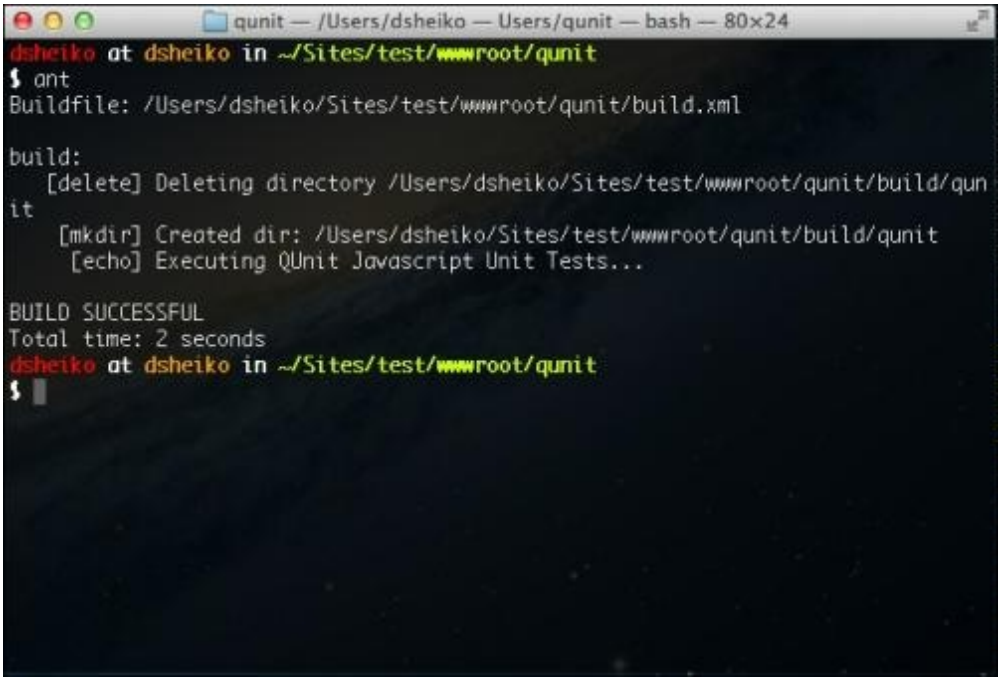
2. Extend it with the build target to run QUnit tests:

```
<?xml version="1.0"?>
<!DOCTYPE project>
<project name="tree"  basedir="."
default="build">
<target name="build" description="runs QUnit
tests using PhantomJS">
  <!-- Clean up output directory -->
  <delete dir="./build/qunit/"/>
  <mkdir dir="./build/qunit/"/>
  <!-- QUnit Javascript Unit Tests -->
  <echo message="Executing QUnit Javascript
Unit Tests..."/>
  <exec executable="/usr/local/bin/phantomjs"
output="./build/qunit/qunit-results.xml">
    <arg
value="./vendors/Runner/runner-muted.js" />
    <arg
value="http://test.dev/qunit/tests/test11/index.html" />
  </exec>
</target>
</project>
```

3. Run Ant in the same directory where `build.xml` is located:

```
ant
```

4. Examine the results shown in the following screenshot:


qunit — /Users/dsheiko — Users/qunit — bash — 80×24
dsheiko at dsheiko in ~/Sites/test/wwwroot/qunit
$ ant
Buildfile: /Users/dsheiko/Sites/test/wwwroot/qunit/build.xml
build:
[delete] Deleting directory /Users/dsheiko/Sites/test/wwwroot/qunit/build/qun
it
[mkdir] Created dir: /Users/dsheiko/Sites/test/wwwroot/qunit/build/qunit
[echo] Executing QUnit Javascript Unit Tests...
BUILD SUCCESSFUL
Total time: 2 seconds
dsheiko at dsheiko in ~/Sites/test/wwwroot/qunit
$

QUnit and CI – setting up Jenkins (Advanced)

Continuous integration is the practice of automating predeployment-related tasks that allows team members to integrate their work into the project frequently.. As an approach, CI doesn't require any particular tool but is most profitable when used with a Continuous Integration Server. You can find many open source servers available nowadays. Yet Jenkins (known as Hudson till 2011; www.jenkins-ci.org) seems to be the most popular solution. It became known as the better alternative to **CruiseControl** (the first original implementation of the CI automation concept) and won many open source awards. Jenkins is quite easy to set up and has a huge repository of plugins that integrate the server with a wide range of third-party tools

We will now install a Jenkins CI server and set it up for a QUnit test runner job.

Getting ready

The best way to explain CI is to show it in an example. Let's assume you are one of the developers working on a project. You're using a version control system so your collaboration on the code goes fine. Yet whenever a stable release is tagged, you inevitably run into problems (integration hell). You attempt to build the release and get surprised with the results. While working with the front end, "build" as a term may sound odd it is nonetheless quite relevant here. During release, a number of tasks must be performed: code quality analysis, code standard violation check, tests, CSS preprocessing, CoffeScript or TypeScript compilation, API documentation generation, image optimization, JS/CSS concatenation, and compression (see the article *H5bp Ant Build Script* at https://github.com/h5bp/ant-build-script). It gets even better on the server-side sources: project size is measured, software metrics are calculated, mess detection is performed, and code duplicates are identified (see *Template for Jenkins Jobs for PHP Projects* at http://jenkins-php.org/index.html). That is what we call a build. We automate the build processing using special software, such as Apache Ant, Maven, Grunt, Phing, and Bash.

CI server listens to the updates on the version control repository. Whenever you commit, the server pulls the working version of the project and runs the **build script**. It collects the reports of the tools performing the build tasks and lets you know by e-mail whether the build was successful or not. Thus, you know immediately if the changes you made cause any trouble during integration.

To get a better grip on CI, we install Jenkins server and define a job running QUnit tests.

How to do it

1. Install Jenkins. If you work under MacOS X, just run:

```
sudo brew install jenkins
```

   To install Jenkins on Unix/Linux, please find instructions at https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins and for Windows at https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+as+a+Windows+service.

2. Create a dedicated user `Jenkins` for the server daemon.
3. Fire up `http://localhost:8080` (by default configuration) in your browser.
4. Examine the Jenkins dashboard page shown in the following screenshot:



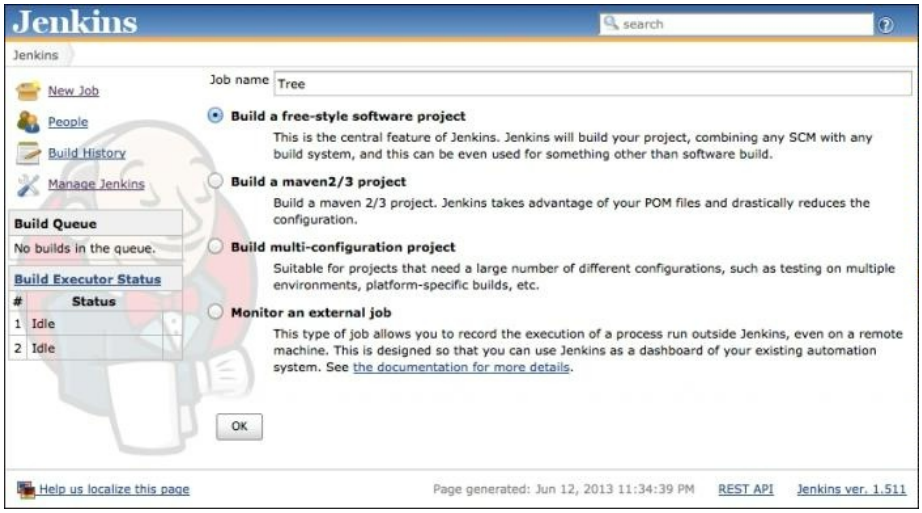
5. Enable the Git plugin.

   1. Navigate to **Jenkins** | **Manage Jenkins** | **Manage Plugins** | **Available** and select **Git plugin**.
   2. Confirm your choice by clicking on the **OK** button at the bottom of the page.

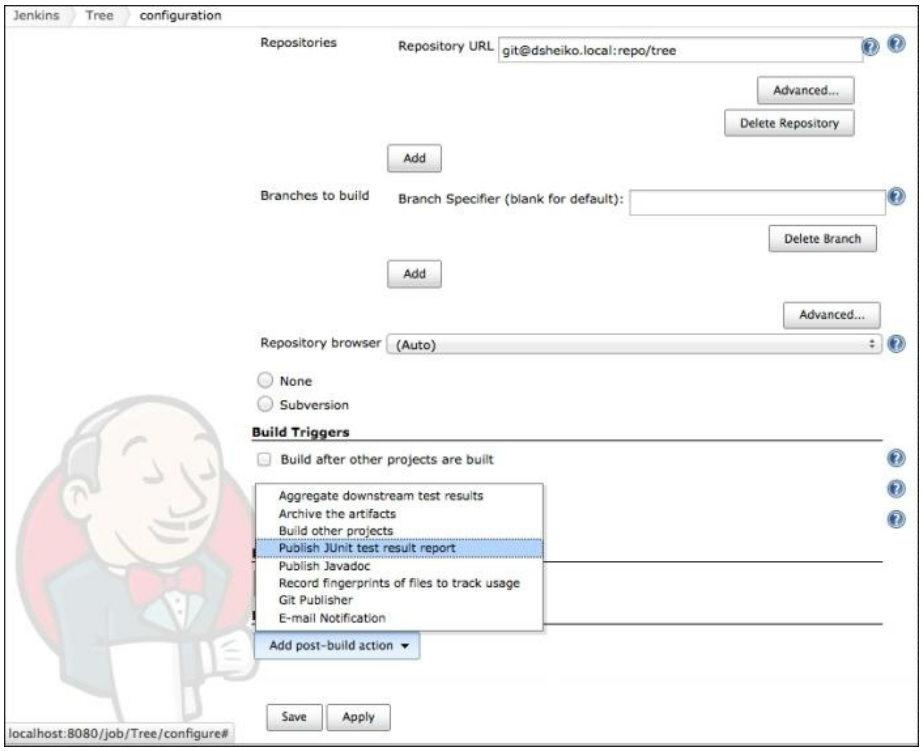
6. Add a new job to Jenkins.

   1. Enter the **Jenkins Dashboard** page and click on the **Add new job** link.
   2. Specify the project (`job`) name in the displayed form.

3. Choose **Build a free-style software project** as shown in the following screenshot:



4. Examine the next screen as shown in the following screenshot:



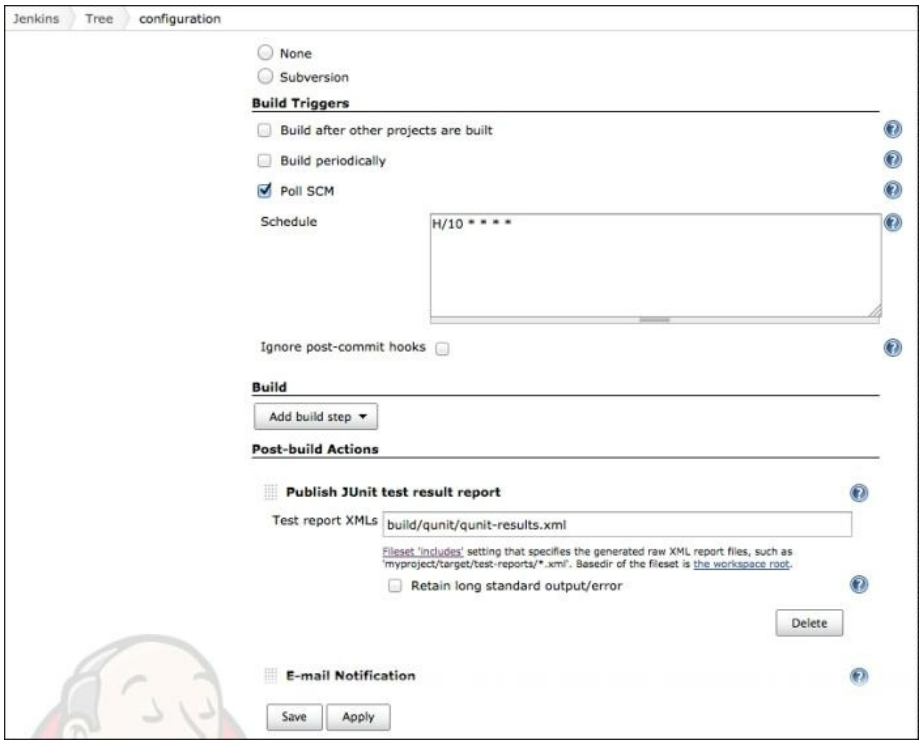
5. Choose the **Source Code Management** / **Git** section and type in your Git remote repository (for example, `git@dsheiko.local:repo/tree`)
6. Select **Publish JUnit test result report** in the **Post-build Actions** combo-box, and specify the location of our build-generated report file (`build/qunit/qunit-results.xml`)
7. Confirm changes by clicking on the **Save** button

7. Come back to the **Jenkins Dashboard** page and click on the **Build Now** link.
8. Find a new record added in the **Build History** block of the Jenkins sidebar menu.
9. Open the link and examine the **build details** page.
10. Follow the **Test Result** link at the bottom to get report details.
11. Examine the rest of the results shown in the following screenshot:



12. Configure the job for continuous integration

    1. Open the project page and click on **Configure**.
    2. Choose the build-required notification method in **Build Triggers** and specify a run schedule in the **Schedule** field as shown in the following screenshot:



3. Pick up **E-mail notification** from the **Post-build Actions** combo-box and type in your email address.

13. Commit a change in your project and push it to the Git server and get automatically notified if the QUnit tests fail.

There's more

If you are an authorized `jenkins` user at the Git server, you won't be able to make a build using the Git plugin. You need a pair of SSH keys to exchange between the `git` and `jenkins` users as described in the *Git Documentation* section (http://git-scm.com/book/ch4-4.html). If you are on a Mac, the article *Setting up a Git Server on Mac OSX* (http://blog.smitec.net/posts/setting-up-a-git-server-on-osx/) will be of great help.

In order to use the Jenkins Git plugin, under Unix/Linux, we login as `jenkins` and create a pair of RSA keys:

```
cd ~/.ssh
ssh-
keygen -f jenkins_rsa -C 'Some comment' -N '' -t
rsa -q
```

Then, we copy derived the `jenkins_rsa.pub` file to the Git server.

```
scp jenkins_rsa.pub
git@our-git-server.com:/home/git/.ssh
```

We also have to edit `~/.ssh/config` to configure the SSH client:

```
Host git@our-git-server.com
  IdentityFile ~/.ssh/jenkins_rsa
```

Now, Jenkins is supposed to be automatically authorized on the Git server.