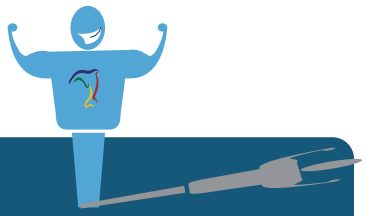
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

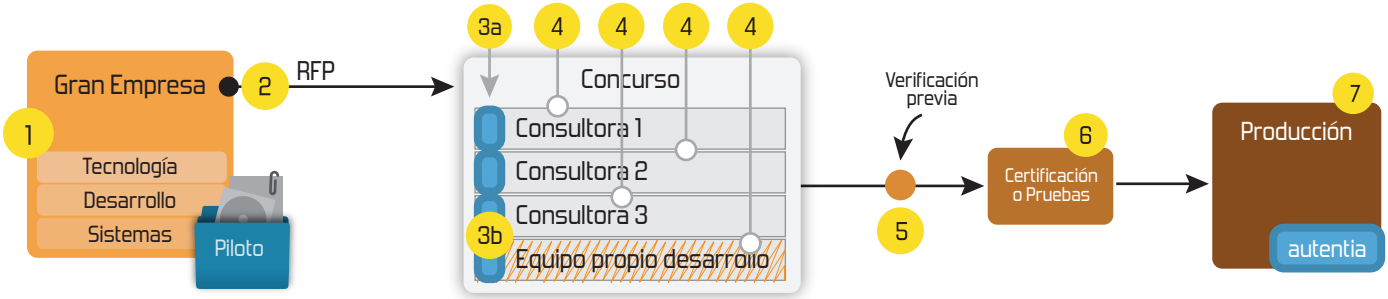
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

AdictosAlTrabajo
Terrakas 1x04
¡¡Ya está en la web!! ;-)
terrakas.com
autentia
Soporte a desarrollo informático
Hosting patrocinado por enredados
Entra en Adictos a través de
E-mail
Contraseña
Entrar
Deseo registrarme
Olvidé mi contraseña
Inicio Quiénes somos Formación Comparador de salarios Nuestro libro Más
» Estás en: Inicio Tutoriales Hello Jasmine! Primeros pasos para hacer BDD/TDD con JavaScript

Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en J2EE, Linux y optimización de aplicaciones empresariales.

Ingeniero en Informática y Certified ScrumMaster

Seguir a @alejandropgarci 767 seguidores

Si te gusta lo que ves, puedes contratarle para darte ayuda con soporte experto, impartir **cursos presenciales** en tu empresa o para que **realicemos tus proyectos como factoría** (Madrid).
Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.

Ver todos los tutoriales del autor

Dibuje sus procesos → Conéctese a sus sistemas IT → Ejecute su aplicación → Descarga Gratis

Fecha de publicación del tutorial: 2012-12-10 Tutorial visitado 1 veces Descargar en PDF

Hello Jasmine! Primeros pasos para hacer BDD/TDD con JavaScript

0. Índice de contenidos.

- 1. Introducción
- 2. Entorno
- 3. Jasmine "a pelo"
- 3.1. Instalación
- 3.2. Escribiendo nuestras especificaciones
- 3.3. Escribiendo los tests y las expectativas
- 3.4. Escribiendo el código de producción
- 4. Jasmine con Maven
- 4.1. Configurando el proyecto Maven
- 4.2. Haciendo BDD/TDD
- 4.3. Integración con el ciclo de construcción de Maven
- 5. Jasmine con Node.js
- 5.1. Configurando el proyecto Node.js
- 5.2. Ejecutando los test
- 5.3. Ejecutando los tests automáticamente
- 6. Conclusiones
- 7. Sobre el autor

1. Introducción

Según los propios autores, Jasmine es un framework de desarrollo dirigido por comportamiento para código JavaScript. No depende de ninguna otra librería JavaScript. No requiere un DOM. Y tiene una sintaxis obvia y limpia para que puedas escribir tus tests fácilmente.

En resumen, podríamos decir que desde que los señores creadores del conocido PivotalTracker sacaron a la luz este framework de test, prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.

En este tutorial vamos ha ver como empezar a trabajar con Jasmine en distintas plataformas, tanto usándolo de forma independiente, como integrado con Maven para Java, o Node para JavaScript puro.

En este enlace os podéis descargar todo el código de los ejemplos.

2. Entorno

Este tutorial está escrito usando el siguiente entorno:

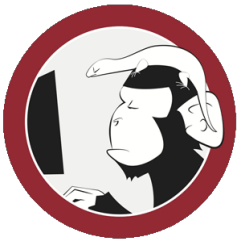
- Hardware: Portátil MacBook Pro 15' (2.5 GHz Intel i7, 8GB 1333 Mhz DDR3, 256GB Solid State Drive).
- AMD Radeon HD 6770M 1024 MB
- Sistema Operativo: Mac OS X Lion 10.8.2
- Jasmine 1.3.1

3. Jasmine "a pelo"

3.1 Instalación

Nuestra primera opción va a ser utilizar Jasmine "a pelo" o como dicen ellos "standalone". Es decir un fichero HTML y nuestro JavaScript sin ningún otro artificio. Para ello nos descargamos la última versión de Jasmine en: https://github.com/pivotal/jasmine/downloads (en nuestro caso el zip de la 1.3.1)

Catálogo de servicios Autentia



Síguenos a través de:

Últimas Noticias

» ESTRENO de Terrakas 1x05: "Un lugar en el mundo"

» Mi retrospectiva sobre la CAS 2012

» Autentia estuvo en el Duatlón de Torrejón de Ardoz

» Participamos en la Carrera de las Empresas 2012

» ¡¡¡Terrakas 1x04 recién salido del horno!!!

Histórico de noticias

Últimos Tutoriales

» ZK 6.5 Consejos para dispositivos móviles

» Inserción de gráficas JFreeChart en un documento Excel mediante la librería POI de Apache

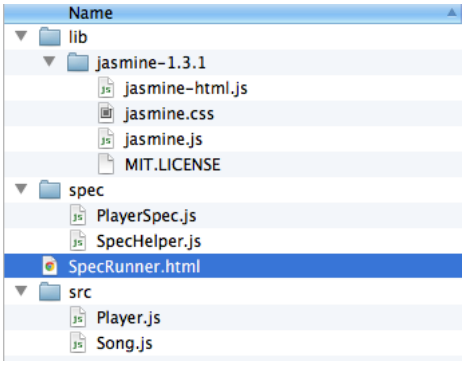
» ZK 6.5 Empezando con el Responsive Design

» Cálculo de una nómina

» ZK 6.5 Responsive design

Últimos Tutoriales del Autor

Si descomprimimos el zip nos encontraremos con la siguiente estructura:



Vemos como tenemos una carpeta **lib** con las librerías de Jasmine, un carpeta **src** con nuestro código de producción (tal cual lo bajamos un pequeño ejemplo de canción y reproductor), y en la carpeta **spec** nuestros ficheros donde escribir las especificaciones de nuestro código en base a tests de Jasmine. Y por último, pero no por ello menos importante, el fichero **SpecRunner.html** (lo podemos ver remarcado en azul); este fichero es un HTML normal y corriente (puede tener cualquier otro nombre si lo preferimos) que al abrirlo con nuestro navegador se encargará de ejecutar los tests para verificar las especificaciones.

Echemos un vistazo a este fichero:

view plain print ?

```
01. <!DOCTYPE HTML PUBLIC "-
    //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
02. <html>
03. <head>
04.   <title>Jasmine Spec Runner</title>
05.
06.   <link rel="shortcut icon" type="image/png" href="lib/jasmine-1.3.1/jasmine_favicon.png">
07.   <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/jasmine.css">
08.   <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.js"></script>
09.   <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-html.js"></script>
10.
11.   <!-- include source files here... -->
12.   <script type="text/javascript" src="src/Player.js"></script>
13.   <script type="text/javascript" src="src/Song.js"></script>
14.
15.   <!-- include spec files here... -->
16.   <script type="text/javascript" src="spec/SpecHelper.js"></script>
17.   <script type="text/javascript" src="spec/PlayerSpec.js"></script>
18.
19.   <script type="text/javascript">
20.     (function() {
21.       var jasminejasmineEnv = jasmine.getEnv();
22.       jasmineEnv.updateInterval = 1000;
23.
24.       var htmlReporter = new jasmine.HtmlReporter();
25.
26.       jasmineEnv.addReporter(htmlReporter);
27.
28.       jasmineEnv.specFilter = function(spec) {
29.         return htmlReporter.specFilter(spec);
30.       };
31.
32.       var currentWindowOnload = window.onload;
33.
34.       window.onload = function() {
35.         if (currentWindowOnload) {
36.           currentWindowOnload();
37.         }
38.         execJasmine();
39.       };
40.
41.       function execJasmine() {
42.         jasmineEnv.execute();
43.       }
44.
45.     })();
46.   </script>
47.
48. </head>
49.
50. <body>
51. </body>
52. </html>
```

Donde podemos destacar:

- Líneas 8 y 9: Las librerías de Jasmine.
- Líneas 12 y 13: Referencias a los ficheros con el código de producción (el código de nuestra aplicación).
- Líneas 16 y 17: Referencia a los test donde tentemos las especificaciones escritas con Jasmine.
- Líneas de la 19 a la 46: El runner de Jasmine que se encarga de ejecutar los tests con las especificaciones.

Si abrimos este fichero con cualquier navegador veremos algo como esto:

» Cómo instalar Java7 en Mac OS X

» Cómo usar el GPS en nuestras aplicaciones iOS 5 (iPhone, iPad, ...)

» Como internacionalizar nuestras aplicaciones iOS 5 (iPhone, iPad, ...)

» TDD con PHP gracias a PHPUnit

» Galerías de imágenes con WOW Slider

Últimas ofertas de empleo

2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.

Alejandro Pérez
alejandropgarci

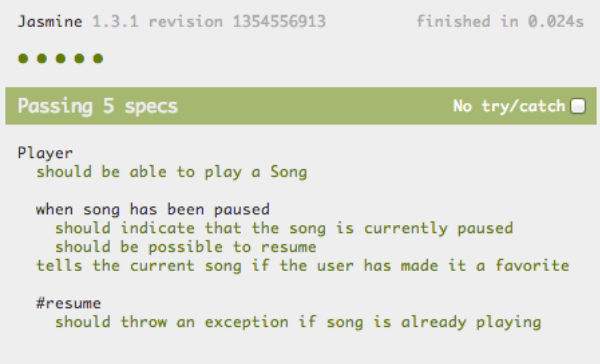
aledoo No hay vuelta atrás: las imágenes de Instagram ya dejan de verse en Twitter – kcy.me/d0xp
4 minutes ago · reply · retweet · favorite

chemaalonso El lado del mal – "Descuido" gordo en la web del Senado de los 500.000 € elladodelmal.com/2012/12/desc #in #dataleak #egoverment }:O
42 minutes ago · reply · retweet · favorite

alejandropgarci Jajaja grandiosa 'Abraham Lincoln: Cazador de vampiros'. Aunque no se a cuento de que viene lo de Abraham Lincoln :P
10 hours ago · reply · retweet · favorite

MaxVico Tubes.io una plataforma para transformar

Join the conversation



3.2 Escribiendo nuestras especificaciones

En el punto anterior hemos visto como poner en marcha Jasmine, ahora vamos a ver como empezar a escribir nuestros propios tests. Para ello vamos a modificar los ficheros actuales para añadir las especificaciones de lo que podría ser la típica lista de cosas pendientes. Esta lista permitirá añadir tareas, dar por terminadas tareas o borrar tareas.

Para editar los ficheros podemos usar el editor que más nos guste en mi caso voy a usar IntelliJ IDEA, pero otra opción más sencilla puede ser Sublime Text. Así empezaremos a escribir nuestro fichero TodoListSpec.js

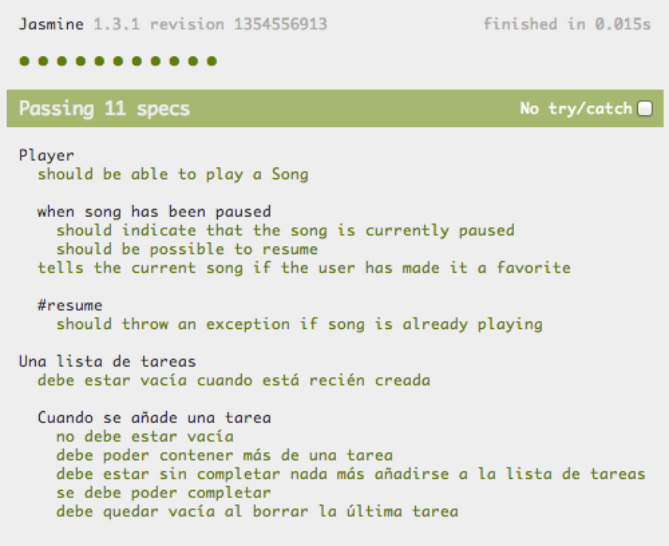
view plain print ?

```
01. describe("Una lista de tareas", function() {
02.
03.     it("debe estar vacía cuando está recién creada", function() {
04.
05.     });
06.
07.     describe("Cuando se añade una tarea", function() {
08.
09.         it("no debe estar vacía", function() {
10.
11.         });
12.
13.         it("debe poder contener más de una tarea", function() {
14.
15.         });
16.
17.         it("debe estar sin completar nada más añadirse a la lista de tareas", function() {
18.
19.         });
20.
21.         it("se debe poder completar", function() {
22.
23.         });
24.
25.         it("debe quedar vacía al borrar la última tarea", function() {
26.
27.         });
28.     });
29. });
```

Por supuesto en un ciclo de TDD o BDD lo primero es escribir el test y luego escribir el código de producción que hace que pase el tests. Ya sabéis, recordar siempre el ciclo Rojo (falla) --> Verde (funciona) --> Refactor (mejora estructura del código sin cambiar funcionalidad). Así que lo que hemos hecho no es demasiado ortodoxo ya que deberíamos haber hecho pasar el primer test antes de escribir el siguiente, pero lo que quería con este ejemplo es que vierais el aspecto que tendrán nuestras especificaciones.

- Línea 1: Nos encontramos con la función **describe**. Esta tiene como primer parámetro un texto que da nombre a la suite o conjunto de tests que vamos a ejecutar. Este nombre, normalmente hará referencia al SUT (Subject Under Test), es decir, la pieza que estamos probando o para la que estamos definiendo las especificaciones, en nuestro caso la lista de tareas. El segundo parámetro es una función donde pondremos la suite de tests (conjunto de especificaciones).
- Línea 3: Nos encontramos la función **it**. Esta define una especificación, tomando un primer parámetro que es el título y un segundo parámetro que es una función que tendrá el código del test y donde pondremos nuestras expectativas. La idea es que si concatenamos el nombre de la suite (el describe) con el título de la especificación (el it) formamos una frase que **describe** que es lo que debe cumplir el SUT.
- Línea 7: Simplemente destacar que podemos tener describe anidados a cualquier nivel. Esto no es más que una forma de organizar nuestros tests y la gracia que tiene es que como leemos las especificaciones, por ejemplo en el caso del segundo it (el primero que está en el describe anidado) seria "Una lista de tareas Cuando se añade una tarea no debe estar vacía".

Si damos de alta este fichero en SpecRunner.html y lo ejecutamos, encontraremos algo de este estilo:



Si os fijáis todo está en verde y esto es terrible!!! pero es natural porque nos hemos saltado el ciclo de Rojo/Verde/Refactor, y como no hemos definido ni test ni expectativas todo sale en verde. Vamos a arreglar esto en el siguiente punto.

3.3 Escribiendo los tests y las expectativas

Por ahora lo que hemos hecho es escribir la descripción de qué esperamos de nuestro SUT, pero no estamos ejecutando nada. Vamos a añadir un poco de código para darle vida al asunto (ojo, insisto en que en la "vida real" estoy hay que hacerlo paso a paso rojo/verde/refactor).

view plain print ?

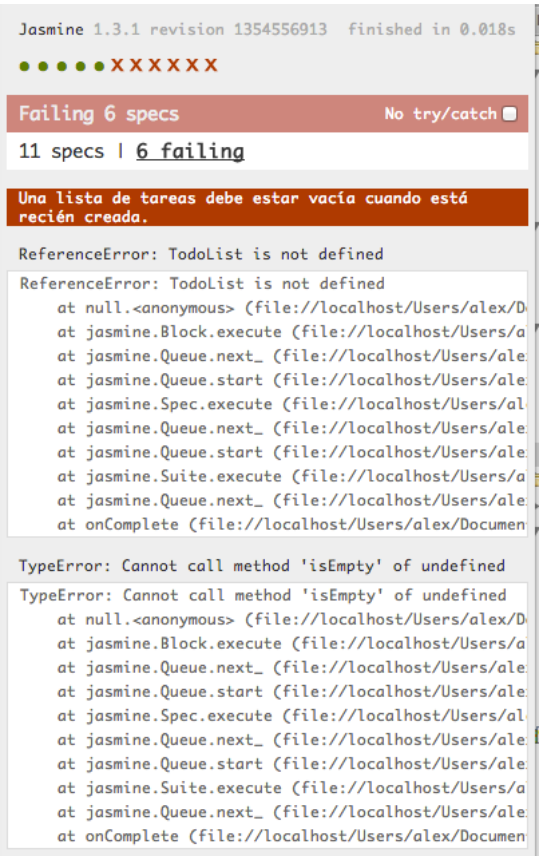
```
01. describe("Una lista de tareas", function() {
02.
03.     var todoList;
04.
05.     beforeEach(function() {
06.         todoList = new TodoList();
07.     });
08.
09.     it("debe estar vacía cuando está recién creada", function() {
10.         expect(todoList.isEmpty()).toBe(true);
11.     });
12.
13.     describe("Cuando se añade una tarea", function() {
14.
15.         beforeEach(function() {
16.             todoList.addTask("Hacer tutorial para adictosaltrabajo.com");
17.         });
18.
19.         it("no debe estar vacía", function() {
20.             expect(todoList.isEmpty()).toBe(false);
21.         });
22.
23.         it("debe poder contener más de una tarea", function() {
24.             todoList.addTask("Ir a la comida de navidad de Autentia");
25.             expect(todoList.size()).toBe(2);
26.         });
27.
28.         it("debe estar sin completar nada más añadirse a la lista de tareas", function() {
29.             expect(todoList.task(0).isDone()).toBe(false);
30.         });
31.
32.         it("se debe poder completar", function() {
33.             todoList.task(0).done();
34.             expect(todoList.task(0).isDone()).toBe(true);
35.         });
36.
37.         it("debe quedar vacía al borrar la última tarea", function() {
38.             todoList.removeTask(0);
39.             expect(todoList.isEmpty()).toBe(true);
40.         });
41.     });
42. });
```

Destacamos:

- Líneas 5 a 7 y 15 a 17: Vemos como usamos la función beforeEach para inicializar el contexto antes de ejecutar cada test (cada it). La idea es que cada test (cada it) se pueda ejecutar de forma independiente, y para ello necesitamos inicializar un contexto o fixture sobre la que se ejecutará el test. También existe el método afterEach que nos permite limpiar el entorno una vez termina cada test (por ejemplo si quisiéramos hacer un rollback de base de datos).
- Línea 10, 20, 25, â€¦: Usamos la función expect para describir las expectativas que debe cumplir nuestro código. A esta función le concatenamos los Matchers, en el ejemplo el toBe para hacer una comparación ===, pero exiten muchos más como: toEqual, toMatch, toBeDefined, â€¦ incluso podemos escribir los nuestros propios. Podemos ver la lista completa en al documentación sobre los Matchers de Jasmine.

3.4 Escribiendo el código de producción

Si ahora volvemos a ejecutar los test (es tan fácil como volver al navegador y recargar la página SpecRunner.html) veremos algo de este estilo:



Así es como reporta Jasmine los errores. Nos va diciendo que especificaciones han fallado, el por qué, y la pila de llamadas hasta el fallo. Ahora mismo nos fallan todas las especificaciones parque no tenemos código de producción, así que vamos a añadirlo (vuelvo a insistir en que nos estamos saltando los pasos de TDD, que luego no se os olvide hacerlo bien).

view plain print ?

```
01. /**
02.  * Representa una tarea de la lista de tareas.
03.  * @param title el título de la tarea
04.  * @constructor
05.  */
06. function Task(title) {
07.     this.title = title;
08.     this.alreadyDone = false;
09. }
10.
11. Task.prototype.isDone = function() {
12.     return this.alreadyDone;
13. };
14.
15. Task.prototype.done = function() {
16.     this.alreadyDone = true;
17. };
18.
19.
20. /**
21.  * Representa una lista de tareas.
22.  * Puede tener varias tareas, cada una de las cuales estará terminada o pendiente.
23.  * @constructor
24.  */
25. function TodoList() {
26.     this.tasks = [];
27. }
28.
29. TodoList.prototype.isEmpty = function() {
30.     return this.tasks.length === 0;
31. };
32.
33. TodoList.prototype.addTask = function(title) {
34.     this.tasks.push(new Task(title));
35. };
36.
37. TodoList.prototype.size = function() {
38.     return this.tasks.length;
39. };
40.
41. TodoList.prototype.task = function(taskIndex) {
42.     return this.tasks[taskIndex];
43. };
44.
45. TodoList.prototype.removeTask = function(taskIndex) {
46.     return this.tasks.splice(taskIndex, 1);
47. };
```

Con este fichero conseguimos volver a tener en verde todos los tests. Eso si, que no se os olvide añadirlo al fichero **SpecRunner.html!!!**

Vamos a dejar ya de profundizar en Jasmine, pero hay mucho más, por ejemplo como usar el resto de Matchers o crear los nuestros, como hacer Spies, como mockear el reloj de JavaScript, como probar operaciones asíncronas, â€¦ para todo ello os recomiendo leer la documentación oficial de Jasmine y practicar, practicar y practicar.

4. Jasmine con Maven

4.1 Configurando el proyecto Maven

Si tenemos un proyecto Java donde también tenemos código JavaScript, podemos integrar en la build de Maven los test de Jasmine gracias al plugin: jasmine-maven-plugin.

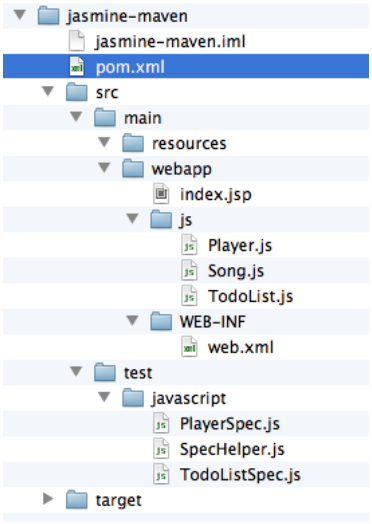
Supongamos que tenemos un proyecto web sencillo:

```
view plain print ?
01. mvn archetype:generate \
02.     -DgroupId=com.autentia.tutorial \
03.     -DartifactId=jasmine-maven \
04.     -DarchetypeArtifactId=maven-archetype-webapp \
05.     -DinteractiveMode=false
```

Ahora debemos editar el **pom.xml** para añadir el plugin de jasmine:

```
view plain print ?
01. <build>
02.    ---
03.     <plugins>
04.        ---
05.         <plugin>
06.             <groupid>com.github.searls</groupid>
07.             <artifactid>jasmine-maven-plugin</artifactid>
08.             <version>1.2.0.0</version>
09.             <extensions>true</extensions>
10.             <executions>
11.                 <execution>
12.                     <goals>
13.                         <goal>test</goal>
14.                     </goals>
15.                 </execution>
16.             </executions>
17.             <configuration>
18.                 <jssrcdir>src/main/webapp/js</jssrcdir>
19.             </configuration>
20.         </plugin>
21.         ---
22.     </plugins>
23.     ---
24. </build>
```

La estructura de directorios y fichero debería ser la que se muestra en la siguiente figura. Nótese que hemos copiado los ficheros js del ejemplo anterior en los directorios correspondientes, y que los ficheros de producción y las especificaciones/test están claramente separados tal como sugiere la estructura estándar de directorios de Maven:



4.2 Haciendo BDD/TDD

Ahora para hacer BDD/TDD podemos ejecutar en la línea de comandos:

```
view plain print ?
01. mvn jasmine:bdd
```

Y abrir con nuestro navegador la dirección: http://localhost:8234/

Veremos la misma pantalla de resultados de Jasmine que veíamos en los ejemplos anteriores. La gran ventaja de este método es que Maven está vigilando tanto el directorio de ficheros de producción como el de test, de forma que en cuento modificamos o añadimos ficheros estos estarán disponibles sin necesidad de tener que añadirlos manualmente a SpecRunner.html, como nos pasaba anteriormente, y bastará con recargar la página en el navegador para ver los resultados de los cambios introducidos.

Además nos permite trabajar con CoffeScript directamente.

4.3 Integración con el ciclo de construcción de Maven

Para esto no tenemos que hacer nada, viene de serie! Basta con ejecutar el típico

```
view plain print ?
01. mvn clean install
```

de Maven y veremos como ademas de todas las fases típicas de Maven aparece una más que lanza los test de Jasmine. En la consola deberíamos ver algo como:

```
JASMINE SPECS
[INFO]
Player
  should be able to play a Song
  when song has been paused
    should indicate that the song is currently paused
    should be possible to resume
  tells the current song if the user has made it a favorite
  #resume
    should throw an exception if song is already playing

Una lista de tareas
  debe estar vacía cuando está recién creada
  Cuando se añade una tarea
    no debe estar vacía
    debe poder contener más de una tarea
    debe estar sin completar nada más añadirse a la lista de tareas
    se debe poder completar
    debe quedar vacía al borrar la última tarea

Results: 11 specs, 0 failures
```

Como veis, la integración con Maven es fácil, sencilla y muuuy practica. Incluso nos puede merecer la pena aunque no trabajemos con Java (bastaría configurar el plugin para que los directorios de código y test se adaptaran mejor a nuestra estructura de proyecto).

5. Jasmine con Node.js

5.1 Configurando el proyecto Node.js

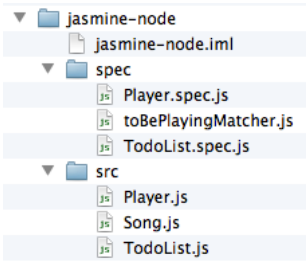
Por último vamos a ver como integrar Jasmine con Node.js. Teniendo en cuenta que estamos hablando de desarrollo con JavaScript, estas opción no podía faltar ;)

Para hacer la instalación ejecutaremos en la línea de comandos:

```
view plain print ?
01. sudo npm install jasmine-node -g
```

(en mi caso hago sudo porque estoy en Mac y quiero hacer una instalación global '-g' para que esté disponible para todos los usuarios)

Además hay que tener en cuenta que node funciona con un sistema de módulos de forma que todo lo que declaramos en un fichero (variables, funciones, ...) son privadas a este. Esto nos obliga a modificar ligeramente el código que habíamos visto hasta ahora. Os recomiendo abrir el código para ver los cambios.



En la estructura de ficheros también podemos apreciar que hemos cambiado el nombre de los ficheros de expectativas ya que ahora tienen que tener la extensión **.spec.js** o **.spec.coffee** (sí, habéis visto bien, igual que con Maven, con Node también podemos escribir nuestras expectativas en CoffeScript).

5.2 Ejecutando los test

Basta con ejecutar en la línea de comandos:

```
view plain print ?
01. jasmine-node spec
```

Siendo spec el nombre del directorio donde están los ficheros de expectativas de Jasmine. Este comando ya se encarga de ejecutar todos los tests que encuentre en este directorio o subdirectorios. De forma que, en nuestro caso, obtendremos una salida de este estilo:

```
...........

Finished in 0.009 seconds
11 tests, 14 assertions, 0 failures
```

5.3 Ejecutando los test automáticamente

jasmine-node tiene un modo muy interesante de trabajo que consiste en ejecutar los tests automáticamente en cuanto modificamos un fichero. De esta forma ni siquiera tenemos que recargar el navegador, como nos pasaba con Maven. Bastará con guardar el fichero y veremos como cambia la consola.

Para activar este modo basta con ejecutar en la línea de comandos:

```
view plain print ?
01. jasmine-node --autotest .
```

Nótese que hemos especificado como directorio donde están los tests . (el directorio actual que engloba tanto spec/ como src/). Esto es porque aunque los tests realmente sólo están el directorio spec/ queremos que Node esté vigilando los cambios tanto de estos ficheros como de los ficheros que están en el directorio de código de producción, de forma que en cuanto cambiemos un test o un fichero de producción, se ejecuten los tests automáticamente.

Una vez activado este modo, veremos en la consola lo siguiente:

```
..........

Finished in 0.01 seconds
11 tests, 14 assertions, 0 failures


{ '0': 0, '1': null }
```

Se ve como se han ejecutado todos los tests y han ido bien. Se ve una última línea con { '0': 0, '1': null }, esto es algo interno y lo podemos ignorar (de hecho seguramente acaben quitándolo de la salida). También vemos como la consola se queda bloqueada en este punto. Vamos a cambiar algo en los fuentes para que fallen los tests, a ver que pasa en cuento salvemos los cambios:

```
....F.

Failures:

  1) Una lista de tareas Cuando se añade una tarea se debe pod
er completar
   Message:
     Expected false to be true.
   Stacktrace:
     Error: Expected false to be true.
    at new jasmine.ExpectationResult (/usr/local/lib/node_modu
les/jasmine-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:102:32)
    at null.toBe (/usr/local/lib/node_modules/jasmine-node/lib
/jasmine-node/jasmine-2.0.0.rc1.js:1171:29)
    at null.<anonymous> (/Users/alex/Documents/tutorial/jasmin
e-hello-world/jasmine-hello-world/jasmine-node/spec/TodoList.s
pec.js:36:47)
    at jasmine.Block.execute (/usr/local/lib/node_modules/jasm
ine-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:1001:15)
    at jasmine.Queue.next_ (/usr/local/lib/node_modules/jasmin
e-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:1790:31)
    at jasmine.Queue.start (/usr/local/lib/node_modules/jasmin
e-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:1743:8)
    at jasmine.Spec.execute (/usr/local/lib/node_modules/jasmi
ne-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:2070:14)
    at jasmine.Queue.next_ (/usr/local/lib/node_modules/jasmin
e-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:1790:31)
    at jasmine.Queue.start (/usr/local/lib/node_modules/jasmin
e-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:1743:8)
    at jasmine.Suite.execute (/usr/local/lib/node_modules/jasm
ine-node/lib/jasmine-node/jasmine-2.0.0.rc1.js:2215:14)

Finished in 0.011 seconds
6 tests, 6 assertions, 1 failure
```

Podemos ver el test que ha fallado con su pila correspondiente. Si ahora volvemos a cambiar el código y volvemos a grabar:

```
......

Finished in 0.013 seconds
6 tests, 6 assertions, 0 failures


..........

Finished in 0.009 seconds
{ '0': 0, '1': null }
11 tests, 14 assertions, 0 failures
```

Aquí es interesante destacar como primero ha ejecutado la suite que había fallado (6 tests, 6 assertions, 0 failures), y como después a ejecutado todos los tests (11 tests, 14 assertions, 0 failures). Este es muy útil ya que primero nos centramos en poner en verde lo que está rojo, y luego comprobamos que no hemos “roto” nada más ;)

Con esta forma de trabajar hay que destacar que según como funcione nuestro editor podemos tener algún problema porque no se detecten correctamente los cambios al guardar los ficheros. Esto por ejemplo pasa con el Vim o el IntelliJ ya que utilizan un modo de guardado seguro que lo que hace es guardar los cambios en un fichero nuevo, luego borrar el original y renombrar el nuevo. Para desactivar esto en IntelliJ podemos irnos a la configuración --> General --> Use "safe write" (safe changes to temporary file first).

6. Conclusiones

Da igual el lenguaje o plataforma en la que desarrollemos, si queremos hacer las cosas bien es fundamental utilizar técnicas como BDD y o TDD. Recordar que este tipo de herramientas nos ayudan a guiar nuestro diseño para conseguir piezas menos acopladas y más reutilizables, y nos permiten tejer una red seguridad que nos permitirá evolucionar nuestro sistema sin miedo al cambio, ya que los tests nos indicarán de inmediato si algo ha dejado de funcionar.

En este tutorial hemos visto como hacerlo para JavaScript, gracias a Jasmine, el framework que posiblemente se ha convertido en el estándar de facto. Y ojo, porque aquí sólo hemos visto como usarlo en solitario, con Maven o con Node, pero lo podemos usar combinado con muchas más plataformas, como Ruby o .NET.

7. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software) y Certified ScrumMaster
Socio fundador de Autentia (Desarrollo de software, Consultoría, Formación) mailto:alejandropg@autentia.com Autentia Real Business Solutions S.L. - "Soporte a Desarrollo" http://www.autentia.com

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Share | 0

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» Registrate y accede a esta y otras ventajas **«**

Esta obra está licenciada bajo licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5

IMPULSA Impulsores Comunidad ¿Ayuda?

---- sin clicks

0 personas han traído clicks a esta página

+ + + + + + + +

powered by karmacracy

Copyright 2003-2012 © All Rights Reserved | Texto legal y condiciones de uso | Banners | Powered by Autentia | Contacto

XHTML 1.0 CSS XML RSS XML ATOM