ceviche.js

Capítulo 8: Mejorando el flujo de trabajo

> *Trabajar como desarrollador web frontend no solo implica saber HTML, CSS y JavaScript. También debemos conocer herramientas que aligeran el flujo de trabajo y reducen tiempo en tareas que llegan a ser repetitivas.*

Grunt

En un flujo de trabajo común vamos a verificar que el código no tenga errores de sintáxis, realizar pruebas unitarias automatizadas, y minificar el código para reducir espacio, entre otras acciones. Realizar cada una de estas tareas puede tomar tiempo, y las vamos a realizar siempre cada cierto tiempo, sobre todo después de realizar un cambio fuerte en el código, así que es vital tener una herramienta que le delegue a la computadora este trabajo tedioso y aburrido. Grunt es un *task runner*, una herramienta que permite definir y realizar este tipo de tareas automatizadas.

Para utilizar Grunt necesitamos Node.js, una plataforma que permite ejecutar JavaScript fuera del navegador, el cual también instalará la herramienta de comandos `npm`. Luego de esto, es necesario instalar la herramienta `grunt-cli` utilizando el comando `npm`:

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

Con esto ya tenemos instalado el comando `grunt` en nuestra consola, el cual es necesario para ejecutar las tareas.

El siguiente paso es crear un archivo `package.json` en la raíz de la carpeta del proyecto. Este archivo es utilizado para definir los paquetes de NPM a utilizar en

el proyecto, pero en este caso lo usaremos para trabajar con Grunt:

```
{
  "name": "buena-espina",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5"
  }
}
```

Un paquete de NPM es solo una biblioteca de JavaScript que, por lo general, funciona dentro de Node.js, y pueden ser encontradas en el sitio web de NPM (cualquiera puede subir sus bibliotecas a NPM, haciéndolas disponibles al público). NPM se encarga de manejar cada paquete y las dependencias de cada paquete, y las descarga de ser necesarias.

El archivo `package.json` contiene 3 propiedades principales, pudiendo tener más: el nombre del proyecto, la versión y las dependencias del proyecto (en este caso, los plugins para Grunt).

Lo siguiente que tenemos que hacer es instalar el paquete `grunt` desde NPM (lo que instalamos líneas arriba solo era el comando para consola, pero también necesitamos la biblioteca que permita utilizar los plugins de Grunt):

```
npm install
```

`npm install` es un comando que descargará cualquier paquete definida como parámetro, o los paquetes definidos en un archivo `package.json`, de existir uno, y los guardará en una carpeta llamada `node_modules` (el cual se creará si no existe). Vamos a utilizar este comando cada vez que querramos instalar una biblioteca definida dentro del archivo `package.json`. Por ejemplo, para agregar un plugin de Grunt al proyecto podemos utilizar el siguiente comando en la consola:

```
npm install grunt-contrib-jshint --save-dev
```

`grunt-contrib-jshint` es un plugin para Grunt que permite utilizar JSHint, una herramienta que analiza el código y lanza advertencias sobre su calidad y posibles errores que pueda tener.

Utilizando la propiedad `--save-dev` en el comando de la consola, NPM agregará una nueva propiedad (`grunt-contrib-jshint`) dentro de la propiedad `devDependencies`. Esta propiedad (`nombre : valor`) tendrá por nombre el nombre del paquete de NPM, y el valor será la versión que se desea instalar:

```
{
  "name": "buena-espina",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0"
  }
}
```

Luego de agregar todos los módulos, debemos crear un segundo archivo, llamado `Gruntfile.js`, el cual también debe estar en la raíz del proyecto.

Un archivo `Gruntfile.js` es un módulo de Node.js; esto es, una función que es guardada en `module.exports`:

```
module.exports = function(grunt) {};
```

Dentro de esta función, debemos realizar 3 pasos:

1. Definir la configuración de cada plugin.

```
module.exports = function(grunt) {
```

```
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
      all: ['scripts/index.js'],  // definir los archivos que se
analizarán
      options: {
        curly: true,    // usar siempre llaves en bloques como if,
while, for
        eqeqeq: true,   // usar === en vez de ==
        browser: true,  // evita lanzar advertencias sobre
variables globales relacionadas al navegador
        globals: {      // evita lanzar advertencias sobre
variables globales específicas
          jQuery: true
        }
      }
    }
  });
};
```

1. Cargar los módulos de Node.js, definidos en el archivo `package.json`.

```
module.exports = function(grunt) {
  grunt.initConfig({
    // ...
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
};
```

1. Registrar las tareas de cada módulo.

```
module.exports = function(grunt) {
  grunt.initConfig({
```

```
    // ...
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.registerTask('default', ['jshint']);
};
```

Al final, el archivo `Gruntfile.js` quedaría así:

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
      all: ['scripts/index.js'],
      options: {
        curly: true,
        eqeqeq: true,
        browser: true,
        globals: {
          jQuery: true
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.registerTask('default', ['jshint']);
};
```

Por último, para correr las tareas, solo basta con ejecutar el siguiente comando en la consola:

```
grunt
```

Bower

Cuando trabajamos con proyectos medianos o grandes, tendremos que utilizar varias bibliotecas, las cuales pueden depender, a su vez, de otras bibliotecas. Si bien hemos visto el manejo de dependencias con RequireJS, esta maneja dependencias a nivel **lógico**, pero no a nivel de archivos. Bower permite manejar este tipo de dependencias (de archivos), descargando las bibliotecas que necesitemos, así como sus dependencias.

Para instalar Bower también necesitamos Node.js. En este caso el comando es el siguiente:

```
npm install -g bower
```

La propiedad `-g` instala el paquete a instalar (en este caso, `bower`), a nivel global, para que pueda ser utilizado por cualquier proyecto en la computadora. Este paquete instalará un comando en la consola, llamado `bower`. Este comando permitirá instalar bibliotecas simplemente con pasarle un nombre.

Por ejemplo, si deseamos instalar jQuery desde bower, solo debemos ejecutar el siguiente comando en la consola:

```
bower install jquery
```

Bower buscará, en su propio repositorio, una biblioteca con ese nombre. En este punto funciona bastante parecido a NPM, ya que cada biblioteca (o paquete de NPM) tiene un nombre único. Estas bibliotecas también manejan versiones, por lo que puedo instalar una versión específica de jQuery:

```
bower install jquery#1.11.1
```

Después de descargar la biblioteca a instalar (jQuery en este caso), creará una carpeta llamada `bower_components`, donde guardará la biblioteca.

RequireJS

Cuando se trabajan en aplicaciones, es necesario separar el código de acuerdo a sus responsabilidades, es decir, lo que realiza cada parte del código, y una buena forma de hacerlo es mediante el patrón Module. De esta forma, separamos el código por responsabilidades, y este se vuelve código reusable.

Sin embargo, si los módulos que creamos dependen de otros módulos (como seguramente será), vamos a tener problemas. En un documento HTML, deberíamos definir primero el módulo que no depende de nadie (llamado módulo `a.js`), luego definir el módulo que depende de `a.js` (el cual será llamado `b.js`), para luego definir al módulo que depende de `b.js`, si existiera, y así sucesivamente. El código quedaría así:

```
<script src="a.js"></script>
<script src="b.js"></script>
<script src="c.js"></script>
```

Pero esto no es óptimo. Si en algún momento `b.js` ya no depende de `a.js`, o `a.js` empieza a depender de un módulo nuevo, las cosas se complican más: No solo vamos a tener que cambiar el código dentro de cada archivo, si no el orden de las etiquetas `<script>`, para que cargue correctamente. Incluso, podría darse el caso en el que `a.js` empieza a depender de `c.js`, y este sigue dependiendo de `b.js` (lo cual pasa, pero debería hacerse lo posible para que no suceda). Es aquí donde aparece RequireJS.

RequireJS permite definir módulos, con sus respectivas dependencias, y solo necesita una etiqueta `<script>`. Para utilizar RequireJS es necesario descargarlo, y luego llamar a la biblioteca agregando la siguiente etiqueta:

```
<script data-main="main" src="require.js"></script>
```

La biblioteca es cargada en una etiqueta `<script>` a la que se define un atributo llamado `data-main`. RequireJS necesita un archivo principal desde donde empezar a cargar la aplicación, usualmente llamado `main.js`. El atributo `data-main` indica la ruta de ese archivo **en relación** al archivo HTML.

Adicionalmente a ello, se puede definir cierta configuración para RequireJS en una etiqueta `<script>` aparte:

```
<script >
  requirejs.config({
    urlArgs: 'timestamp=' + Date.now()
  });
</script>
```

Por ejemplo, utilizando este código tendremos que cada archivo cargado por RequireJS tendrá una dirección parecida a `archivo.js?timestamp=1418873637178`. De esta forma, el navegador evitará guardar en caché a estos archivos (útil cuando probamos un código muy seguido y necesitamos que el navegador siempre utilice el archivo real).

En nuestro caso, solo tenemos dos archivos, `jquery.js` e `index.js`, que será suficiente para hacer un ejemplo básico de RequireJS.

Como ya hemos instalado Bower, lo usaremos para instalar RequireJS. Si bien puede descargarse directamente, en este caso usaremos el comando `bower install`:

```
bower install requirejs
```

Luego de haber instalado RequireJS, vamos a modificar el archivo `index.html`, donde tenemos las siguientes etiquetas:

```
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="scripts/index.js"></script>
```

Lo primero que debemos hacer es dejar una sola etiqueta `<script>`, de la siguiente forma:

```
<script data-main="scripts/index" src="bower_components/requirejs
/require.js"></script>
```

Nuestro archivo `scripts/index.js` necesita convertirse en un módulo de RequireJS. Para esto, definimos un módulo con la función `require`, el cual toma dos parámetros:

- Un arreglo con las rutas de las dependencias del módulo que se está creando (que deben ser otros módulos de RequireJS)
- Una función que contendrá el código del módulo.

Esta función, a su vez, debe tener definidos tantos argumentos como elementos tenga el arreglo, asumiendo que cada elemento es una ruta a un módulo.

Así, el código que estaba dentro de `index.js` debería ir aquí (notemos que la función no tiene ningún argumento):

```
require(['../bower_components/jquery/dist/jquery'], function() {
  // ...
});
```

`../bower_components/jquery/dist/jquery` es la ruta de la biblioteca jQuery descargada desde Bower, y esta ruta es relativa a `scripts/index.js`. Si usamos más bibliotecas desde Bower, o armamos una estructura mucho más compleja de carpetas y archivos, tendremos muchas rutas que escribir, y posiblemente en más de un solo lugar.

RequireJS tiene una propiedad de configuración llamada `paths`, donde se

pueden definir los nombres de cada módulo y sus respectivas rutas. De esta forma, les damos un alias a cada módulo:

```
<script type="text/javascript">
  requirejs.config({
    paths: {
      jquery: '../bower_components/jquery/dist/jquery'
    }
  });
</script>
```

Así, solo necesitamos escribir el alias dentro de `require`.

```
require(['jquery'], function($) {
  // ...
});
```

---

jQuery es una biblioteca que trata de funcionar en todos los casos posibles, ya sea llamándolo desde una etiqueta `<script>` o utilizando RequireJS.

Para el primer caso, la biblioteca agrega una variable global llamada `jQuery` (y su alias `$`) a `window`, mientras que en el segundo caso, crea un módulo del tipo AMD (que es el tipo de módulo que usa RequireJS). Puedes leer más sobre AMD en la misma web de RequireJS.

Estas dos porciones de código están dentro de `jquery.js`, donde el primero crea las variables globales `jQuery` y `$`:

```
if ( typeof noGlobal === strundefined ) {
  window.jQuery = window.$ = jQuery;
}
```

Mientras que el segundo crea un módulo AMD:

```
if ( typeof define === "function" && define.amd ) {
  define( "jquery", [], function() {
    return jQuery;
  });
}
```

Aquí podemos notar dos cosas importantes: se usa la función `define`, y esta toma 3 parámetros. La función `define` permite, como su nombre indica, definir un módulo, y puede tener un nombre *propio* (que es el primer parámetro). Los otros dos parámetros son similares a los usados en la función `require`: un arreglo de dependencias y una función que englobe el código del módulo. Cabe resaltar que `require` solo debe usarse en el archivo principal (definido en el atributo `data-main`), ya que no solo define un módulo, si no que lo **ejecuta inmediatamente**, mientras que `define` solo define un módulo que será utilizado luego.

En el caso de jQuery, el módulo es llamado `jquery`, y es por eso que debemos usarlo en la configuración de RequireJS, dentro de la propiedad `paths`.

Para terminar, volvamos por un momento a la primera implementación de nuestro código con RequireJS:

```
require(['../bower_components/jquery/dist/jquery'], function() {
  // ...
});
```

Aquí vemos que la función que englobará nuestro código no tiene ningún argumento, pero el código funciona sin problemas. Esto significa que jQuery está usando como una variable global (utilizando `window.jQuery` o `window.$`).

Sin embargo, en la última implementación, vemos que la función sí tiene un argumento:

```
require(['jquery'], function($) {
  // ...
});
```

Esto sucede porque, al usar el nombre del módulo de jQuery (definido por `jquery.js`) en la propiedad `paths`, ya estamos usando el módulo de tipo AMD.

← 7. Capítulo 7: jQuery

Acerca de | Disponible en Amazon.com