Desarrollo Web en Entorno Cliente

UD 10. Javascript: del desarrollo clásico al desarrollo moderno

Actualizado Octubre 2020

Licencia

Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

ÍNDICE DE CONTENIDO

UD10. JAVASCRIPT: DEL DESARROLLO CLÁSICO AL DESARROLLO MODERNO

1. INTRODUCCIÓN

En esta unidad aprenderemos de como pasar del desarrollo clásico de Javascript a estilos de desarrollo moderno, donde utilizaremos algunas herramientas que nos pueden ayudar.

Utilizaremos **Webpack** para empaquetar aplicaciones y **Webpack-dev-server** para lanzar un servidor de pruebas. También más adelante veremos los fundamentos tanto de un transpilador como **Babel** como de la configuración de un linter sencillo como **ESLint**.

El uso de **Webpack** lo observaremos a través del proceso de migración de desarrollo clásico a moderno de una aplicación sencilla llamada **FakeChat** (un chat falso, que no funciona).

Posteriormente, trataremos un ejemplo sencillo para explicar los fundamentos de **Babel** y **ESLint**.

2. PASANDO DE JAVASCRIPT CLÁSICO A MODERNO A TRAVÉS DE EJEMPLOS

2.1 Antes de empezar: instalando NodeJS

Para realizar el taller necesitamos NodeJS https://nodejs.org/es/. Podéis descargarlo de la página oficial. Instalar NodeJS nos proporcionará el intérprete de Javascript NodeJS así como otras utilidades interesantes como **NPM** (NodeJS Package Manager), que es el gestor de paquetes de NodeJS.

Es recomendable instalar la última versión estable desde la web. Si instaláis desde algún repositorio como el de Ubuntu, es posible que instale una versión antigua muy por debajo de la estable.

2.2 FakeChat simple – Ejemplo 0 (Todo junto!)

Esta es la versión inicial de nuestro FakeChat simple, de la que partiremos. Consiste en un **"index.html"** donde tenemos todo el Javascript dentro de una etiqueta <script> y como externo únicamente un fichero **"styles.css**" donde guardamos la hoja de estilo CSS.

Como funciones destacadas:

- **escapeHTML**: función auxiliar que recibe un texto y lo devuelve "escapando" el HTML para evitar ataques de XSS. Es llamada tanto al enviar datos como la mostrarlos.
- **inicio**: función llamada al cargarse el body para asegurarnos que el DOM esté cargado. Esta se encarga de iniciar el div del chat, recuperando el valor del chat de localStorage.
- **enviarMensaje**: función encargada de enviar un mensaje a nuestro chat y guardar la actualización del chat en localStorage.

2.3 FakeChat – Ejemplo 1 (Separamos un poco con <script src>)

Similar al primero, solo que separamos utilizando **<script src>** y dividimos la funcionalidad en dos ficheros, uno donde están tanto "inicio" como "escapeHTML· y otra donde está "enviarMensaje". Asimismo del código HTML se han eliminado los manejadores de eventos y se han introducido dentro del código Javascript.

2.4 FakeChat – Ejemplo 2 (Comienza la revolución: empezamos con Webpack)

Comenzamos a usar **NodeJS**, su gestor de paquetes **NPM** y según vuestra configuración o sistema operativo NPX para la ejecución de paquetes.

Pasos para empezar:

- Creamos un directorio para nuestro proyecto.
- Usando en el directorio del proyecto "npm init" (nos pide información) o "npm init -y" (usa información por defecto) inicializa el directorio como proyecto Node con el fichero de configuración "package.json".
    - Para saber más de "package.json" puedes visitar
        - https://medium.com/noders/t%C3%BA-yo-y-package-json-9553929fb2e3
    - Si tenemos un directorio con "package.json" y ejecutamos "npm install", NPM descargara automáticamente todas las dependencias de nuestro proyecto.
- Con "**npm install webpack webpack-cli –save-dev**" instalamos los paquetes webpack y webpack-cli como paquetes utilizados por nuestro proyecto cuando estemos en desarrollo y no en producción (por eso **–save-dev** y no **–save** a secas).
    - Recordamos que las siglas **CLI** suelen indicar que es "**Comand Line Interface**". En este caso webpack-cli nos ayuda a realizar ciertas operaciones con Webpack.

Webpack es una herramienta que nos permite muchas opciones. En esta parte del taller, simplemente la utilizaremos para que automáticamente analice nuestros ficheros ".js" y genere un único fichero ".js" para su distribución.

Si queréis saber más sobre Webpack https://webpack.js.org/guides/getting-started/

Aunque Webpack puede usarse vía consola, es más útil disponer de un fichero de configuración, este suele llamarse por defecto "**webpack.config.json**" aunque puede tener otros nombres sobretodo para tener distintas opciones de configuración.

En el ejemplo nuestro fichero contiene:

```
const path = require('path');
module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

Con esta configuración, Webpack procesa el fichero "**/src/main.js**" de nuestro proyecto y guarda el nuevo fichero generado con todas las inclusiones como "**/dist/main.js**".

En el directorio "**/dist**" hemos dejado a mano nuestro HTML y nuestra hoja de estilo CSS.

Para conseguir el empaquetado, procedemos a ejecutar Webpack usando "**npx webpack**" o "**npx webpack –config webpack.config.json**". En ambos casos tomará como fichero de configuración "webpack.config.json."

En el código Javascript, usamos "export" para indicar que una función es exportable

```
export function escapeHtml(text) {
```

```
  return text.replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#039;");
}
```

Asimismo usamos import con {} para importar funciones:

```
import  { enviarMensaje } from './eventos.js';
import  { escapeHtml } from './eventos.js';
```

Para saber más sobre export y export default

https://stackoverflow.com/questions/33611812/export-const-vs-export-default-in-es6

2.5 FakeChat – Versión 3 (Webpack se alía con Webpack-dev-server)

2.5.1 Webpack en modo "watch"

En primer lugar, modificando "**scripts**" dentro del fichero "**package.json**" podemos poner nombres a scripts y que se ejecuten con "**npm run nombrescript**".

Para este ejemplo creamos dos scripts:

```
"build": "webpack -progress -p",
"watch":"webpack -progress --watch"
```

El primero básicamente ejecuta en modo producción Webpack tal como vimos en el primer ejemplo. El segundo, lanzado con "**npm run watch**" ejecuta Webpack en un modo muy interesante: el modo "**watch**". Este modo básicamente si detecta que un fichero de los que procesa Webpack recibe algún cambio, es que debe procesarlo automáticamente. Es decir, si cambiamos alguno de los ".js" de nuestro proyecto, Webpack lo detecta y genera el fichero **"main.js".**

2.5.2 Copy-webpack-plugin

Para que el proyecto tenga una estructura más amigable y que los ficheros "index.html" y **"styles.css"** no tengan que estar copiados directamente a **"/dist"**, nos hemos apoyado en el plugin "copy-webpack-plugin" hemos definido la siguiente estructura: **"src/js"**, **"src/html"** y **"src/css"**. Configuraremos el plugin para que se detecten cambios en la carpeta **"src/html**" y **"src/css"** (que no son procesadas directamente por Webpack) y sus contenidos se copien a **"/dist"**.

Si queréis saber mas de este plugin podéis visitar

https://webpack.js.org/plugins/copy-webpack-plugin/

Para poner en marcha nuestra configuración daremos los siguientes pasos:

- Instalamos el plugin con "**npm install copy-webpack-plugin –save-dev**"
- Añadiremos arriba del todo del "**webpack.config.js**" la linea "**const CopyPlugin = require('copy-webpack-plugin');**"
- Añadiremos el siguiente código a **"webpack.config.js"** para el copiado automático de los directorios especificados:

```
plugins: [
        new CopyPlugin(
            [
                { from: 'src/html', to: './', force: true },
                { from: 'src/css', to: './', force: true },
            ]
        ),
 ]
```

2.5.3 devtool:source-maps

Cuando Webpack nos empaquete varios ficheros, nos genera una batiburrillo de código difícil de depurar. Para facilitar la depuración existe generación de "source-maps" en el código, que en modo "development" incluyen información para enlazar el código generado con el código original.

Mediante herramientas expertas, como por ejemplo el depurador de "Firefox developers" se puede depurar respecto al código original y no al código empaquetado.

Para ver las distintas opciones de configuración de "source-maps"

https://webpack.js.org/configuration/devtool/

En el ejemplo, mediante "webpack.config.json" nosotros usamos la siguiente configuración:

```
devtool: 'eval-source-map',
```

2.5.4 webpack-dev-server

La característica "watch" es muy útil, pero se hace mucho más útil cuando entra en juego el plugin "webpack-dev-server". Este es un plugin configurable que crea un pequeño servidor web donde se cargará la página para facilitarnos su depuración.

En esta sección hablaremos sobre cómo configurarlo para que cuando se haga un cambio no solo nos genere de nuevo los ficheros empaquetados, sino para que además de forma automática nos recargue el navegador y veamos necesario.

Si queréis saber más sobre este plugin https://webpack.js.org/configuration/dev-server/

Para poner en marcha esta gran utilidad realizamos los siguientes pasos:

- Instalaremos el plugin con "**npm install webpack-dev-server –save-dev**"
- Añadiremos un script a "**package.json**" para llamar al servidor.
  - **"server": "webpack-dev-server –open"** que será llamado con "**npm run server**".
  - También podemos llamar a mano al plugin con "**npx webpack-dev-server –open**"
- Añadiremos la siguiente configuración a "**webpack.config.json**"

```
devServer: {
publicPath:"/",
hot:true,
contentBase: path.resolve(__dirname, 'dist'),
```

```
watchContentBase: true, // Mira cambios en /dist
writeToDisk: true,
}
```

Esta configuración indica que el servidor re-cargue el navegador automáticamente (hot), que su carpeta inicio sea "**/dist**", que vigile los cambios en "**/dis**t" y que al ejecutarlo los ficheros generados no estén en memoria, sino que los copie a disco en "**/dist**" (necesario para que también actualice automáticamente cambios en "**src/html**" y "**src/css**").

2.5.5 Incluyendo bibliotecas en el proyecto para producción

Con un fin puramente didáctico (abajo explicamos porque en este contexto no es lo más útil), podemos guardar nuestras bibliotecas para producción y que Webpack las incluya en el fichero ".js" generado. Podemos instalar por ejemplo jQuery con **"npm install --save jquery**".

Un detalle importante es que al guardarse la biblioteca como parte del proyecto y no solo para modo "**development**", se utiliza en el guardado **--save** lugar de **–save-dev**.

En concreto, para incluir jQuery en NodeJS.

```
import $ from 'jquery';
```

¿Porque en este contexto no es útil hacer esto? A efectos prácticos, al usar bibliotecas tipo jQuery, Vue, etc. nos puede interesar que en vez de obtenerlas de nuestro sitio como hemos hecho en este ejemplo, la obtengas de los CDN optimizados para ello como estamos haciendo en ejemplos anteriores con este código de **"index.html".**

```
<script src="http://code.jquery.com/jquery-3.5.1.slim.js"
  integrity="sha256-DrT5NfxfbHvMHux31Lkhxg42LY6of8TaYyK50jnxRnM="
  crossorigin="anonymous"></script>
```

3. BABEL COMO TRANSPILADOR

Babel es un transpilador (un compilador pasa de código fuente a código máquina. Un transpilador transpila de un código fuente a otro código fuente en otro lenguaje.

A efectos prácticos, nos permite escribir código Javascript en el estándar que queramos (ES6, Typescript, etc.) y traducir ese código a otras versiones del estándar de Javascript diferentes. Generalmente buscando que sean soportadas por navegadores más antiguos.

Para saber más sobre Babel

https://medium.com/@renzocastro/webpack-babel-transpilando-tu-js-502244a61f5b

Para instalarlo utilizamos "**npm install babel-core babel-preset-env babel-loader --save-dev**"

Para configurarlo y que funcione de forma automática y transparente con Webpack modificamos "**webpack.config.json**" añadiendo:

```
module:{
      rules: [
            {
                  test: /\.js$/,
                  exclude: /node_modules/,
                  use: {
                        loader: 'babel-loader'
                  }
            }
      ]
},
```

Esto indica que cada fichero ".js" que pase por "**Webpack**" (excluyendo los paquetes descargados en "**/node_modules**") debe ser procesado por Babel.

Para configurar Babel, usaremos el fichero "**.babelrc**". Aquí un ejemplo donde se pide que transpile según los navegadores que queremos soporte nuestro código. Para indicar navegadores se usa la sintaxis de https://browserl.ist

En el ejemplo está activo Firefox 71, pero están comentado para "jugar" con ellos NodeJS y IE 6.

```
{     "presets": [
            ["@babel/preset-env", {
                  "targets": {
                        "firefox": "71" //"ie": "6"
                  }
            }]
      ]
}
```

4. ESLINT COMO LINTER

ESLint es un linter entre los más populares. Un linter nos permite descubrir errores de código así como puede asesorarnos e incluso obligarnos a seguir un estilo de código.

Entornos como Visual Studio Code y su plugin para ESLint, no solo integran el linter dándote información mientras desarrollas, sino que además te permiten configurar el formateador de documentos para que automáticamente lo formatee al formato deseado por el linter.

- Mas información en https://scotch.io/tutorials/linting-and-formatting-with-eslint-in-vs-code

Para instalar **ESLint** en nuestro proyecto podemos usar "**npm install eslint –save-dev**".

Para generar un fichero de configuración "**.eslint.js**" podéis usar "**npx eslint –init**" y mediante unas preguntas os generará el fichero ".eslint.js" que luego podéis adaptar al gusto.

El fichero usado en el ejemplo es:

```
module.exports = {
  env: {
    browser: true,
    es6: true
  },
  extends: [
    'standard'
  ],
  globals: {
    Atomics: 'readonly',
    SharedArrayBuffer: 'readonly'
  },
  parserOptions: {
    ecmaVersion: 2018,
    sourceType: 'module'
  },
  rules: {
    "no-tabs": 0
  }
}
```

De forma resumida, indicar que en este fichero me baso en las reglas "**standard**" para estilo de código y que he añadido **"no-tabs": 0** para amplias las reglas y me permita el uso de tabuladores.

5. ESLINT VS SONARLINT

A principio del curso recomendaba usar **SonarLint**. Realmente **SonarLint**, actúa únicamente en el entorno gráfico y aunque se puede hacer lo mismo con ESLint (con más configuración que SonarLint).Ahora ya conocéis de la existencia de ambos y podéis usar aquel que os sea más práctico, o ambos incluso.

6. ¿DEBO REPETIR ESTA CONFIGURACIÓN PARA CADA PROYECTO?

Ya sabemos que es **Webpack, Webpack-dev-server, Babel** y **ESLint**. ¿Debo realizar estos pasos para cada proyecto que inicie? La respuesta en NO. Basta con hacerlo una vez y guardar el proyecto con la configuración (por ejemplo en un repositorio de GitHub o GitLab) y simplemente clonarlo cada vez que empecemos un proyecto. Además este proyecto incluye ya un **".gitignore"** para que determinadas cosas no se suban al repositorio (por ejemplo, los módulos descargados).

La base con la configuración aquí descrita está disponible en el repositorio https://github.com/sergarb1/BaseJavascriptModerno

Simplemente debéis bajar el repositorio y tras ello ejecutar "npm install" para que instale los módulos que necesita el proyecto base. **(Webpack, Babel, ESLint)**.

7. ENTORNOS WEB PARA TRABAJAR ONLINE

Aparte de esta configuración para trabajar en local, hay servicios que te permiten tener un entorno online similar. Suelen ser gratuitos para proyectos públicos y de un desarrollado y de pago para proyectos privados y múltiples desarrolladores. Algunos de los más conocidos son:

- CodePen: https://codepen.io/
- StackBlitz: https://stackblitz.com/
- JS Fiddle: https://jsfiddle.net/

8. BIBLIOGRAFÍA

[1] Javascript Mozilla Developer https://developer.mozilla.org/es/docs/Web/JavaScript

[2] Javascript ES6 W3C https://www.w3schools.com/js/js_es6.asp

9. AUTORES (EN ORDEN ALFABÉTICO)

A continuación ofrecemos en orden alfabético el listado de autores que han hecho aportaciones a este documento:

- García Barea, Sergi