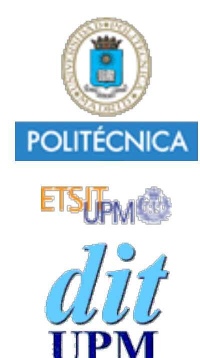

Blockchain: Desarrollo de Aplicaciones

WebPack

Caso de Estudio #2

BCDA 2018

Versión: 2018-11-09

© Santiago Pavón - UPM-DIT

Índice

- ~~Introducción, documentación.~~
- ~~Ganache~~
- ~~Caso de Estudio #1: SC Contador~~
  - ~~Crear un proyecto que usa un Smart Contract llamado Contador.~~
  - ~~Usando Truffle, crear pruebas, scripts, cmds y una dapp.~~
- Caso de Estudio #2: Webpack
  - Añadir Webpack al caso de estudio #1
- Caso de Estudio #3: Introducción a React y Redux
  - Crear un ejemplo de una SPA con un nuevo contador.
    - El contador es diferente al de los casos de estudio anteriores: Sin Blockchain.
    - Partiendo de la configuración de Webpack del caso de estudio #2 y ampliarla para soportar react.
- Caso de Estudio #4: Smart Contract Contador con React y Redux. Y SIN Drizzle.
  - Unir todo lo visto en los casos de estudio anteriores.
- Caso de Estudio #5: SC Contador con React, Redux y CON Drizzle.
  - Añadir Drizzle al caso de estudio #4.

© Santiago Pavón - UPM-DIT

¿Qué es WebPack?

- Muy breve:
  - Se usa para crear Single Page Applications (SPA).
  - Es un empaquetador de módulos que mete nuestra jerarquía módulos en un bundle.
  - Traduce (transpila) el código desarrollado a otras versiones de código que entiendan los navegadores.
    - Ej: Pasar de ES6 a ES5, JSX a ES5, de SASS a CSS, ...
  - Minimizar, ofuscar, ...

- Documentación:

`https://webpack.js.org`

© Santiago Pavón - UPM-DIT

Contador con WebPack

Caso de Estudio #2

© Santiago Pavón - UPM-DIT

Pasos para Usar Webpack

- Partimos del caso de estudio #1.
  - Copiar todos los ficheros del caso de estudio 1.
- Retocamos para eliminar los desarrollos que no nos interesan, limpiar el código, ...:
  - Eliminar el directorio **test** con los test, el directorio **scripts** con los scripts, y el directorio **app1** con las aplicaciones de línea de comando.
    - Solo me interesa la aplicación web que desarrollamos en el directorio **app2** en el caso de estudio #1.
  - Renombrar el directorio **`app2`** a **`src`**.
- Instalar el paquete webpack (versión 4)

```
$ npm install --save-dev webpack webpack-cli

$ npx webpack -v

4.25.1
```

© Santiago Pavón - UPM-DIT

Crear **`webpack.config.js`**

- Crear el fichero **`webpack.config.js`** con la configuración de webpack.
  - Especificar el directorio con los ficheros fuente con los que trabajaremos: **`src`**.
    - Estos ficheros fuentes son los que crearemos y editaremos para crear la aplicación.
  - Especificar el directorio donde webpack creará los ficheros (optimizados, minimizados, ofuscados, ...) que se distribuirán con la aplicación final: **`dist`**.
    - Estos ficheros no los modificaremos, los genera webpack.

© Santiago Pavón - UPM-DIT

webpack.config.js

```
const path = require('path');

const SRC_DIR_PATH = path.resolve(__dirname, './src');
const SRC_APP_PATH = path.resolve(SRC_DIR_PATH, './js/app.js');

const DIST_DIR_NAME = 'dist';
const DIST_DIR_PATH = path.resolve(__dirname, DIST_DIR_NAME);
const DIST_BUNDLE_FILENAME = 'main.js';

module.exports = {
  entry: SRC_APP_PATH,
  output: {
    filename: DIST_BUNDLE_FILENAME,
    path: DIST_DIR_PATH
  },
  mode: "development"
};
```

© Santiago Pavón - UPM-DIT

Adaptar **src/index.html**

- Borrar de **src/index.html**:
  - Las etiquetas las **<script>** que cargaban desde CDNs o localmente los siguientes ficheros javascript: **jquery.min.js**, **web3.min.js**, **truffle-contract.min.js** y **app.js**.
  - La etiqueta **<link>** que cargaba la hoja de estilos CSS.
- Ahora, los módulos y ficheros JS y CSS anteriores se importarán en el fichero **src/js/app.js** para indicarle a webpack que se necesitan en el proyecto, e indicar cuáles son las dependencias (jerarquía) entre ellos.
  - Webpack incluirá todos esos ficheros dentro de **dist/main.js**.
- Añadir al final de **src/index.html**:
  - Una etiqueta **<script>** para importar **dist/main.js**, que es el bundle que generará webpack con todos los ficheros empaquetados.

© Santiago Pavón - UPM-DIT

src/index.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1">
    <title>Contador</title>
  </head>
  <body>
    <h1>Contador</h1>

    <p>
      Valor actual = <span id="valor"></span>
    </p>

    <button type="button" id="cincr">Incrementar</button>

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

© Santiago Pavón - UPM-DIT

Adaptar **src/js/app.js**

- Para meter los módulos **jquery**, **web3** y **truffle-contract** en el bundle creado por webpack, primero debemos instalarlos como paquetes npm.
  - Ejecutar.

```
$ npm install --save-dev jquery@2.2.4
```

    - Nota: Instalar la version 2.2.4 porque la última 3.x.x. no funciona.

```
$ npm install --save-dev web3
```

```
$ npm install --save-dev truffle-contract
```

  - Borrar el fichero **src/js/truffle-contract.min.js**.
    - Usaremos el que acabamos de instalar en node_modules.

- Modificamos **src/js/app.js** para importar estos paquetes:

```
import $ from "jquery";
import "web3";
import "truffle-contract";
```

© Santiago Pavón - UPM-DIT

- Modificamos **src/js/app.js** otra vez para importar el fichero JSON con el artefacto del contrato **Contador**.

```
import cJson from "../../build/contracts/Contador.json";
```

  - Ya no es necesario descargar este fichero usando Ajax.
    - Lo acabamos de importar.
  - Modificar la función **initContractAbstracts** del objeto **App** para eliminar la llamada Ajax:

```
initContractAbstracts: function() {
    console.log("Inicializando abstracción del contrato.");

    // Crear la abstraccion del contrato Contador
    App.Contador = TruffleContract(cJson);

    // Provisionar el contrato con el proveedor web3
    App.Contador.setProvider(App.web3Provider);

    App.initContractInstance();
},
```

© Santiago Pavón - UPM-DIT

- Modificar **src/js/app.js** para importar los ficheros CSS que usa la aplicación.
  - Se añade el siguiente **import** para que webpack empaquete también el fichero CSS:

```
import "../css/style.css";
```

    - Nota: webpack extiende el lenguaje para que pueda hacerse import de cualquier tipo de fichero, siempre que existan los Loaders adecuados.

- Añadir en **webpack.config.js** los loaders que cargan ficheros CSS.

```
module.exports = {
  . . .
  module: {
    rules: [
      {
       test: /\.css$/,
       use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
};
```

- Hay que instalar los loaders con npm

```
$ npm install --save-dev style-loader css-loader
```

© Santiago Pavón - UPM-DIT

- Cambiar la definición de **App** a **window.App** para que el objeto **App** sea visible globalmente en todos los ámbitos.

```
import "../css/style.css";

import $ from "jquery";
import  "web3";
import  "truffle-contract";

import cJson from "../../build/contracts/Contador.json";

window.App = {
    web3Provider: null,
    Contador: {},
    contador: {},

    init: function() {
    . . . ETC . . .
```

© Santiago Pavón - UPM-DIT

- Evitar un error:
  - Cuando lancemos el servidor más adelante y nos conectemos con un navegador que no haya creado un objeto web3, aparecerá un error indicando que no existe la variable **`web3`**.
    - Webpack transpila el código javascript y el nuevo código falla con el siguiente error:

```
▼ ReferenceError: Can't find variable: web3          initWeb3 — app.js:44
    initWeb3 — app.js:44
    init — app.js:31
    (función anónima) — app.js:146
    dispatch — jquery.js:4737
    handle — jquery.js:4549
```

  - Este error no aparece usando un navegador que haya inyectado una instancia de web3, como ocurre al usar la extensión MetaMask, o un navegador tipo Mist.

- Se soluciona definiendo en **`app.js`** la variable **`web3`** con el valor **`windows.web3`**.
  - Añadimos la siguiente sentencia en **`src/js/app.js`**:

```
. . .
import cJson from "../../build/contracts/Contador.json";

let web3 = window.web3;

window.App = {
    web3Provider: null,
. . .
```

© Santiago Pavón - UPM-DIT

webpack.config.js

- También hay que copiar el fichero **src/index.html** al directorio de distribución **dist**.
- Para ello usaremos el plugin **copy-webpack-plugin**.
  - Se instala con el comando:

```
$ npm install --save-dev copy-webpack-plugin
```

- Editar el fichero **webpack.config.js** para añadir/configurar el uso de este plugin:

```
const CopyWebpackPlugin = require('copy-webpack-plugin');
. . .
const SRC_INDEX_PATH = path.resolve(SRC_DIR_PATH, './index.html');
const DIST_INDEX_FILENAME = 'index.html';

module.exports = {
  . . .
  plugins: [
    new CopyWebpackPlugin([
      { from: SRC_INDEX_PATH, to: DIST_INDEX_FILENAME }
    ])
  ],
  . . .
};
```

© Santiago Pavón - UPM-DIT

Servidor Web

- En el caso de estudio #1 creamos un servidor web usando Nodejs y Express para servir las páginas estáticas que forman la aplicación.
  - Lo necesitamos solo mientras estamos desarrollando.

- Hay que adaptar este servidor para que ahora sirva el bundle **main.js** y el fichero **index.html** desde el directorio de distribución:
  - Cambiar **node_server/app.js** para que las rutas estáticas se sirvan desde el directorio **../dist**
    - Quitamos:

```
app.use(express.static(path.join(__dirname, '../app2')));
app.use(express.static(path.join(__dirname,
                                     '../build/contracts')));
app.use(express.static(path.join(__dirname, 'public')));
```

    - y añadimos:

```
app.use(express.static(path.join(__dirname, '../dist')));
```

- Nota: Este servidor no será necesario en breve cuando empecemos a usar el paquete **webpack-dev-server**.

© Santiago Pavón - UPM-DIT

Ejecutar la Aplicación

- Ejecutamos el programa **./node_modules/.bin/webpack** para crear el bundle

```
$ npx webpack

$ npx webpack --config webpack.config.js
```

  - La opción **--config** indica cual es el fichero de configuración a usar.
    - El valor por defecto es **webpack.config.js**.
      - En nuestro caso podríamos omitir la opción al ser el mismo valor.
  - Tras ejecutar este comando
    - Se debe haber generado el bundle **dist/main.js**.
    - Se debe haber copiado **index.html** desde **src** a **dist**.

- Ya podemos lanzar el servidor node ejecutando:

```
$ cd node_server

$ npm start
```

- Lanzamos el navegador y nos conectarnos a **localhost:3000**

© Santiago Pavón - UPM-DIT

Mejorar **package.json**

- Para que nuestro paquete sea privado, y evitar que lo publiquemos por error, modificamos **package.json**.
  - Eliminamos la propiedad **main**
  - Creamos la propiedad **private** a **true**
- Limpieza:
  - Eliminar la sección "directories".
  - ...
- Crear un script llamado **build** para facilitar la invocación de webpack.
  - Ejecutando "npm run build"
- package.json:

```
...
"private": true,
"scripts": {
     "build": "webpack --config webpack.config.js"
},
...
```

© Santiago Pavón - UPM-DIT

Usar **webpack-dev-server**

- El paquete **webpack-dev-server** se usa en desarrollo.
- Este paquete instala un servidor web que sirve la aplicación, y vigila si se ha modificado algún fichero fuente.
  - Si detecta que ha cambiado algún fichero fuente:
    - Ejecuta automáticamente webpack.
    - Recarga automáticamente en el navegador los módulos usados en la página web.
  - Nos evita tener que preocuparnos de ejecutar webpack manualmente y luego recargar la página web en el navegador cada vez que cambiamos un fichero fuente.
- Hay que instalar este paquete con:

```
$ npm install --save-dev webpack-dev-server
```

© Santiago Pavón - UPM-DIT

- La configuración puede hacerse como opciones en línea de comando o proporcionándolas en **webpack.config.js**.
  - Editar **webpack.config.js** para añadir la siguiente propiedad:

```
devServer: {
     contentBase: DIST_DIR_PATH,
     port: 3000
},
```

    - Estas opciones indican cuál es el directorio donde están los ficheros a servir, y el puerto que usará el servidor web.
  - Para lanzar el servidor invocamos:

```
$ npx webpack-dev-server --watch-content-base --hot
```

    - Estas opciones indican que vigile cambios en los ficheros de contentBase, y que actualice los módulos de la página de la aplicación en el navegador.
- Para facilitar la invocación del servidor de desarrollo de webpack, podemos crear en **package.json** un script llamado **dev**:

```
"scripts": {
   ...,
   "dev": "webpack-dev-server --watch-content-base --hot"
}
```

© Santiago Pavón - UPM-DIT

Usar **clean-webpack-plugin**

- Para hacer limpieza de ficheros viejos antes de rehacer el contenido a distribuir.
  - Cada vez que ejecutemos webpack se borra el contenido del directorio **dist**.
    - Util cuando cambiamos los nombres de los ficheros generados para evitar que queden restos de cosas viejas que ya no se usan.
- Hay que instalar el paquete **clean-webpack-plugin** con:

```
$ npm install --save-dev clean-webpack-plugin
```

- Hay que editar **webpack.config.js** para instalar el plugin:

```
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {
  . . .
  plugins: [
    new CleanWebpackPlugin([DIST_DIR_NAME]),
    . . .
  ],
  . . .
};
```

© Santiago Pavón - UPM-DIT

Usar Source Maps

- Para que los logs y trazas permitan localizar en qué punto de los ficheros se han producido los warning y errores.

- Sin este plugin los logs y las trazas hacen referencia a los ficheros bundles.
  - Es muy difícil examinar estos ficheros.
  - Es prácticamente imposible depurar con esa información.

- Hay que editar **`webpack.config.js`** para añadir la opción **`devtool`**:

```
module.exports = {
  . . .
  devtool: 'inline-source-map',
  . . .
};
```

  - Nota: el valor **inline-source-map** no es bueno en producción.

© Santiago Pavón - UPM-DIT

Desarrollo o Producción

- Los entornos de desarrollo y producción son diferentes.
  - En desarrollo interesa tener Source Maps pesados, usar webpack-dev-server para regenerar los bundles y repintarlos automáticamente, ...
  - En producción nos interesa minimizar y ofuscar el código, tener Source Maps ligeros, mejorar el tiempo de carga, ...
- Usaremos dos ficheros de configuración diferentes.
  - Un fichero para cada modo con las propiedades adecuadas en cada caso.
- Para no repetir (DRY) las partes comunes usaremos una utilidad llamada **`webpack-merge`** que permite combinar ficheros de configuración.
  - Crearemos un fichero de configuración para desarrollo (**`webpack.dev.js`**), otro para producción (**`webpack.prod.js`**), otro con las partes comunes (**`webpack.common.js`**) que mezclaremos con los dos primeros, y otro con las constantes (**`webpack.const.js`**).
  - La utilidad **`webpack-merge`** se instala como un paquete npm:

```
$ npm install --save-dev webpack-merge
```

© Santiago Pavón - UPM-DIT

```
const path = require('path');

const SRC_DIR_PATH = path.resolve(__dirname, './src');
const SRC_APP_PATH = path.resolve(SRC_DIR_PATH, './js/app.js');
const SRC_INDEX_PATH = path.resolve(SRC_DIR_PATH, './index.html');

const DIST_DIR_NAME = 'dist';
const DIST_DIR_PATH = path.resolve(__dirname, DIST_DIR_NAME);
const DIST_BUNDLE_FILENAME = 'main.js';
const DIST_INDEX_FILENAME = 'index.html';

module.exports = {
  SRC_DIR_PATH,
  SRC_APP_PATH,
  SRC_INDEX_PATH,
  DIST_DIR_NAME,
  DIST_DIR_PATH,
  DIST_BUNDLE_FILENAME,
  DIST_INDEX_FILENAME
};
```

webpack.const.js

© Santiago Pavón - UPM-DIT

```
const CopyWebpackPlugin = require('copy-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

const {
  SRC_APP_PATH, SRC_INDEX_PATH,
  DIST_DIR_NAME, DIST_DIR_PATH,
  DIST_BUNDLE_FILENAME, DIST_INDEX_FILENAME } = require('./webpack.const');

module.exports = {
  entry: SRC_APP_PATH,
  output: {
    filename: DIST_BUNDLE_FILENAME,
    path: DIST_DIR_PATH
  },
  plugins: [
    new CleanWebpackPlugin([DIST_DIR_NAME]),
    new CopyWebpackPlugin([
      { from: SRC_INDEX_PATH, to: DIST_INDEX_FILENAME }
    ])
  ],
  module: {
    rules: [
      {
       test: /\.css$/,
       use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
};
```

webpack.common.js

© Santiago Pavón - UPM-DIT

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

const { DIST_DIR_PATH } = require('./webpack.const');

module.exports = merge(common, {
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
     contentBase: DIST_DIR_PATH,
     port: 3000
  }
});
```

webpack.dev.js

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: "production"
});
```

webpack.prod.js

© Santiago Pavón - UPM-DIT

- También modificaremos los scripts de **package.json** para poder lanzar webpack en modo desarrollo o en modo producción.

```
"scripts": {
    "build": "webpack --config webpack.dev.js",
    "prod": "webpack --config webpack.prod.js",
    "dev": "webpack-dev-server --config webpack.dev.js
                                   --watch-content-base --hot"
  },
```

© Santiago Pavón - UPM-DIT

Prod: Ofuscar, Minimizar y Source Maps

- En producción hemos asignado el valor **`'production'`** a la propiedad **`mode`**.
  - Este valor provoca que se cargue el plugin **`UglifyJSPlugin`**, que ofusca y minimiza el bundle.

- En producción también es útil tener **Source Maps** para su uso al depurar o ejecutar benchmarks.
  - Usaremos **`'source-map'`** como valor de la propiedad **`devtool`**.
    - Es un valor adecuado para producción.
      - Consultar la documentación de webpack para ver otros valores también adecuados en producción.
  - Y configuraremos el plugin **`UglifyJSPlugin`** para indicarle que use **Source Maps**.

© Santiago Pavón - UPM-DIT

webpack.prod.js

```
const merge = require('webpack-merge');
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: "production",
  devtool: 'source-map',
  plugins: [
    new UglifyJSPlugin({
      sourceMap: true
    })
  ]
});
```

© Santiago Pavón - UPM-DIT

Definir NODE_ENV

- **Nodejs** usa la variable de entorno **NODE_ENV** para definir el modo de trabajo.

- Muchas librerías consultan el valor de esta variable de entorno para decidir qué deben incluir en la librería.
  - El comportamiento de estas librerías puede ser muy diferente dependiendo del valor de esta variable de entorno.

```
if (process.env.NODE_ENV === 'production') {
    // Hacer unas cosas
} else {
    // Hacer otras cosas
}
```

- Usaremos el plugin **DefinePlugin** para definir el valor de esta variable.
  - Así se hace visible en los módulos que se incluyen en el bundle.
    - El plugin **DefinePlugin** es parte del paquete **webpack**.

© Santiago Pavón - UPM-DIT

webpack.prod.js

```
const webpack = require('webpack');
const merge = require('webpack-merge');
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: "production",
  devtool: 'source-map',
  plugins: [
    new UglifyJSPlugin({
      sourceMap: true
    }),
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
});
```

© Santiago Pavón - UPM-DIT

webpack.dev.js

```
const webpack = require('webpack');
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

const { DIST_DIR_PATH } = require('./webpack.const');

module.exports = merge(common, {
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
     contentBase: DIST_DIR_PATH,
     port: 3000
  },
  plugins: [
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('development')
    })
  ]
});
```

© Santiago Pavón - UPM-DIT

Minimizar CSS

- Puede que se incluya de serie en la versión 5 de webpack.
- De momento hay que usar un plugin, retocar los loaders, ...
- y además hay que configurar también la minimización de los módulos JS, ya que se pierde la configuración que viene por defecto.

© Santiago Pavón - UPM-DIT