Introducción a ES6+ (Parte I)

Estructuración de un proyecto ES6+ con Babel/Webpack

Competencias

- Estructurar un proyecto web sencillo, considerando que ES6+ se debe transpilar bajo una librería.
- Utilizar ejemplos de estructuras de proyectos.

Introducción

Los navegadores modernos son capaces de interpretar correctamente las versiones más antiguas de JavaScript, hablamos en este caso de ES5 o versiones anteriores. Sin embargo en la actualidad, esa versión de JavaScript no es la más popular, se considera en varios aspectos incompleta (punto que veremos más adelante), y en proyectos que utilizaban esas versiones, era necesario utilizar algunas librerías que facilitaran el trabajo, considerando las falencias que tenían dichas versiones del lenguaje. Sin embargo, existe un margen de navegadores que no permiten el uso de ES6 de forma nativa, por ende, cuando se trabaja con productos que quieren ser compatibles para la gran mayoría de los dispositivos y navegadores, es necesario realizar un trabajo adicional para compatibilizar la sintaxis ES6 a una versión más antigua y de esa forma, volverla compatible con mayor cantidad de versiones.

El objetivo es enseñarles algunos elementos básicos para configurar un entorno de trabajo de modo que permita transpilar código ES6 a alguna versión más antigua y de esa forma, volverlo compatible para un mayor espectro de navegadores.

Node Package Manager (NPM)

No entraremos a entender el concepto de **Node.js** (Node) como tal, puesto que, este contenido da para un curso completo. Sin embargo, es importante señalar que Node es un framework popular y conocido, se creó con el objetivo de llevar JavaScript a nivel de servidor, de esta forma, no solamente sería utilizado a nivel de cliente, sino que también se podrían crear servicios asíncronos en los

servidores con el objetivo de unificar lenguajes y poder trabajar proyectos completos con JavaScript como tal. Node tiene sus propias librerías y dependencias, y se manejan con su gestor, llamado **Node Package Manager**.

Node Package Manager o simplemente npm es un gestor de paquetes, que hará más fáciles nuestras vidas al momento de trabajar con Node, ya que gracias a él podremos tener cualquier librería disponible con solo una línea de código, npm nos ayudará a administrar nuestros módulos, distribuir paquetes y agregar dependencias de una manera sencilla. Particularmente hablar de dependencia es similar a un package en Java o un módulo en python, una dependencia como tal es una librería que se puede componer de otras librerías, tanto internas como utilizar recursos externos a ésta. Las librerías están hechas para facilitarnos la vida como desarrolladores, en este sentido es válido considerar que siempre antes de buscar por cuenta propia resolver un problema, buscar en la web si esto ya ha sido resuelto antes, dicha práctica ahorra mucho trabajo a los desarrolladores, por lo general son soluciones probadas. En general, las personas que generan soluciones específicas para siertos problemas, transforman dichas soluciones en librerías, de esta forma es posible implementar esta solución para cualquier individuo que la busque.

Retomando el concepto de npm, cuando instalamos nuevos paquetes lo que hace npm es instalarlo de manera local en nuestro proyecto dentro de la carpeta **node_modules**. Esta carpeta contendrá todas las librerías y dependencias necesarias para iniciar el proyecto, según lo que se defina en el archivo **package.json**. Se debe considerar que para no escribir las dependencias a utilizar a mano en dicho archivo, es necesario utilizar el siguiente comando:

```
npm install **nombre_dependencia** --save
```

Con esto nos aseguramos que la dependencia se instale en nuestra carpeta de proyecto. Por otra parte, si queremos instalarla en nuestra máquina, independiente del proyecto debemos ejecutar el siguiente comando:

```
npm install **nombre_dependencia** -g
```

Con esto nos aseguramos de guardar dicha dependencia o librería de forma global ya que **-g** indica que sea global.

Para instalar npm en nuestro ordenador es necesario primero tener Node.js. Así que verificaremos que se encuentre instalado, para eso vamos a usar el siguiente comando en la consola del computador:

```
nodejs -v
```

Si fue instalado correctamente verán en consola la versión de node instalada.

En caso de no tenerlo instalado, es necesario seguir los pasos de instalación que se encuentran en la página oficial de node, según el sistema operativo que se esté utilizando. Particularmente siempre recomiendo Linux o MacOS para trabajar en estos entornos, además de tener instalado algún **IDE**. Particularmente, recomiendo utilizar Visual Studio Code o Sublime Text, también está la opción de utilizar Brackets, es bastante útil para FrontEnd pero no es tan completo como los 2 mencionados anteriormente. Cabe destacar que se sugiere seguir alguna guía de referencia para instalar node en distribuciones de Linux, en cada una de ellas se instala de forma distinta.

Es importante la instalación de node puesto que para usar ES6 es necesario correr ciertos comandos que proveen librerías de node para poder convertir el código en una forma legible para los navegadores.

En este curso enseñaremos lo escencial para transpilar código ES6 en formato legible, existen diferentes formas de levantar proyectos que tienen esto integrado de forma automática, pero requiere conociemientos sobre otros tópicos que no se abordarán por el momento.

Babel y Webpack

En esta sección abordaremos las principales características de Babel y webpack. Dichas dependencias van de la mano puesto que, en su conjunto permiten configurar un entorno de trabajo. Transpilar código ES6 para volverlo interpretable y generar una configuración para ejecutar los comandos necesarios para que la aplicación o el servicio funcione correctamente.

Babel

Para solucionar la problemática de compatibilidad entre estándares y navegadores se utilizan compiladores que convierten el código JavaScript de un estándar a otro estándar más antiguo y, por lo tanto, más compatible.

Babel es un compilador que convierte un estándar nuevo en una versión totalmente compatible de JavaScript. Así, tenemos la ventaja de poder programar en un estándar nuevo sin renunciar a la compatibilidad entre navegadores. Lo utilizaremos en este curso para transformar ES6 en alguna versión que sea reconocible por la gran mayoría de los navegadores.

Webpack

Webpack se define como un empaquetador de módulos (un **bundler** en la jerga habitual) pero que hace muchísimas cosas más:

- Gestión de dependencias
- Ejecución de tareas
- Conversión de formatos
- Servidor de desarrollo
- Carga y uso de módulos de todo tipo (AMD, CommonJS o ES6)

Y esto último lo que hace que destaque especialmente. Es una herramienta extremadamente útil cuando desarrollas aplicaciones web diseñadas con filosofía modular, es decir, separando el código en módulos que luego se utilizan como dependencias en otros módulos. Una de las cosas que hace realmente bien Webpack es la gestión de esos módulos y de sus dependencias, pero también puede usarse para cuestiones como concatenación de código, minimización y ofuscación, verificación de buenas prácticas (linting), carga bajo demanda de módulos, etc.

Una de las cosas interesantes de Webpack es que no solo el código JavaScript se considera un módulo. Las hojas de estilo, las páginas HTML e incluso las imágenes se pueden utilizar también como tales, esto da un extra de potencia muy interesante.

Webpack se puede considerar como un Task Runner muy especializado en el procesamiento de unos archivos de entrada para convertirlos en otros archivos de salida, que utiliza unos componentes que se denominan loaders.

Lo habitual es utilizar un archivo de código especial llamado `webpack.config.js` que se debe ubicar en la raíz de tu proyecto y que define mediante código JavaScript las operaciones que quieres realizar. En este archivo defines toda la información necesaria para poder utilizar Webpack para tus propósitos.

Estructura de un proyecto Node básico

Por lo general los programadores trabajan en equipo, los equipos pueden ser pequeños o grandes dependiendo de la complejidad del producto en sí, para que el trabajo en equipo sea efectivo, es necesario que los proyectos sean entendibles por el equipo de trabajo, de esta forma, los programadores pueden trabajar en diversas partes de un proyecto sin estorbarse unos a otros o reduciendo la dependencia de trabajo. Es por ello que es necesario definir una estructura inicial.

Los ejemplos presentados a continuación serán básicos orientados a proyectos personales, en otras instancias formativas ustedes pueden participar directamente en cursos de frameworks conocidos que tienen una estructura mucho más compleja y completa. Desde esta perspectiva, siempre se puede dividir un proyecto a un nivel *atómico* en donde los desarrolladores de un equipo entiendan sus funcionalidades y qué hacer en un proyecto.

Observemos un ejemplo de estructura.

Dentro del contenido visto, se revisaron ejemplos básicos de estructura de un proyecto, ahora que utilizamos Node, las cosas cambian un poco, la verdad, no es tan complejo.

```
proyecto
|   README.md
|   index.html
|
└──css
|   |   style.css
|   |   otherStyle.css
|
|
|
└──js
|   |   main.js
|
|
|
|
|___img
|   |   img1.png

...
```

Como se observa en el ejemplo tenemos las mismas 3 carpetas principales:

1. css: Carpeta en donde se guardan todos los archivos de estilos del proyecto.
2. js: Carpeta en donde se guardan todos los archivos de tipo del proyecto. En este caso escribiremos en estos archivos con el formato de ES6. Explicaremos el lenguaje más adelante, por ahora nos enfocaremos en entender que dichos archivos estarán escritos en ES6.
3. img: Carpeta en donde se guardan las imágenes del proyecto.

Para importar la lectura del archivo, importaremos main.js, sin embargo, al momento de hacer la configuración, se hará lo siguiente:

```
<script type="text/javascript" src="./js/main.js"></script>
```

Antes que nada, verificaremos que tu computador tenga instalado node JS con el comando `node -v`, si la consola entrega una versión, quiere decir que tienes instalado node en tu equipo.

Posterior a ello debes correr el comando: `npm init` en la carpeta raiz de tu proyecto. De esta forma se inicializarán los archivos necesarios para transformar tu proyecto en uno que pueda ser compilado por babel. Al correr el comando el sistema te hará unas preguntas, presiona enter para crear la configuración con los valores predeterminados.

Luego, debes correr la siguiente secuencia de comandos para instalar las dependencias necesarias:

```
npm i webpack webpack-cli @babel/core @babel/plugin-proposal-object-rest-
spread @babel/preset-env babel-loader -D
```

Básicamente lo que hace este comando es instalar las dependencias de webpack y babel para configurar tu proyecto como uno compilable.

Vas a observar que en tu carpeta raíz se crearon diversos archivos nuevos. Uno de ellos se llama `package.json` para proyectos Node, este archivo es fundamental, es una guía completa con las configuraciones básicas necesarias para generar comandos que permitan ejecutar ciertos servicios. Por defecto, siempre estará configurado el script: `npm start`, sin embargo, tu puedes modificar este archivo a tu gusto, altamente recomendado cuando sabes trabajar con estos archivos.

Abriremos el archivo `package.json` y reemplazamos su contenido por lo siguiente:

```
{
    "name": "proyecto",
    "version": "1.0.0",
    "description": "",
    "main": "webpack.config.js",
    "scripts": {
        "build": "webpack --mode production"
    },
    "repository": {
        "type": "git", // opcional
        "url": ""
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "bugs": {
        "url": "" //opcional
    },
    "homepage": "",
    "devDependencies": {
        "@babel/core": "^7.0.0-beta.42",
        "@babel/plugin-proposal-object-rest-spread": "^7.0.0-beta.42",
        "@babel/preset-env": "^7.0.0-beta.42",
        "babel-loader": "^8.0.0-beta.2",
        "webpack": "^4.2.0",
        "webpack-cli": "^2.0.12"
    }
}
```

Analizaremos algunas etiquetas importantes a considerar de este archivo.

```
"Scripts": {...}
```

Esta etiqueta provee comandos para ejecutarse y generar ciertas acciones. En este caso, solo tenemos un comando configurado que es `build`, que llamará a webpack para que ejecute los comandos avanzados de ejecución. Se pueden crear diversos tipos de scripts, más adelante enseñaré algunos otros ejemplos.

```
"devDependencies": {...}
```

esta etiqueta son las dependencias que estarán instaladas para el entorno de desarrollo de un producto, en este curso al no trabajar con node, nos limitaremos solo a conocer las dependencias necesarias para generar la compilación del código.

Ahora, en nuestro archivo ubicado en `/js/main.js` colocaremos el siguiente bloque de código:

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }
[5, 6].map(n => console.log(n));
```

Este bloque de código tiene una sintaxis que analizaremos más adelante, por ahora es importante señalar que está escrito en ES6, por ende, en navegadores antiguos no podrá ser interpretado por éstos.

Ahora iremos al archivo de configuración de babel y colocaremos el siguiente bloque de código:

```
{
 test: /\.js$/, //Regular expression
 exclude: /(node_modules|bower_components)/,//excluded node_modules
 use: {
 loader: "babel-loader",
 options: {
   presets: ["@babel/preset-env"]  //Preset used for env setup
  }
 }
}
```

Este es un archivo en formato JSON que toma los archivos de tipo `.js`, con ciertas excepciones y los transforma en archivos legibles por un navegador. Por otra parte, es necesario hacer una configuración de webpack cambiando y reemplazando lo escrito en el archivo `webpack.config.js` por lo siguiente:

```
const path = require("path");
const webpack = require("webpack");
const webpack_rules = [];
const webpackOption = {
    entry: "./app.js",
    output: {
        path: path.resolve(__dirname, "dist"),
        filename: "bundle.js",
    },
    module: {
        rules: webpack_rules
    }
};
let babelLoader = {
```

```
    test: /\.js$/,
    exclude: /(node_modules|bower_components)/,
    use: {
        loader: "babel-loader",
        options: {
            presets: ["@babel/preset-env"]
        }
    }
};
webpack_rules.push(babelLoader);
module.exports = webpackOption;
```

Explicaré algunas de las cosas presentes acá, cabe destacar que varios de los elementos presentes se explicarán con detalle más adelante.

El archivo `webpack.config.js` es JavaScript y como tal, define un conjunto de objetos asignados a diversas variables con el objetivo de generar una configuración de webpack ordenada y legible. Es importante señalar que los archivos de webpack, por lo general, cuentan con **expresiones regulares** para poder interpretar y buscar dentro de las carpetas y subcarpetas de un proyecto, todos los archivos que cumplan con la regla de expresiones regulares. En el glosario podrás encontrar una definición de expresiones regulares, el concepto es bastante complejo, para facilitar el trabajo te recomiendo que por ahora solo apliques las configuraciones necesarias.

Cabe destacar además que webpack es una dependencia, por ende, las configuraciones entregadas son entregadas a la dependencia para ser procesadas y generar un orden en la compilación y ejecución de la serie de comandos señalados.

Ahora, configuraremos el archivo final de `babelrc`:

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "targets": {
                    "node": "8.9.9", // debes colocar acá la versión de node
que tengas
                    "esmodules": true
                }
            }
        ]
    ],
    "plugins": ["@babel/plugin-proposal-object-rest-spread"]
}
```

Para validar que todo funciona correctamente, es necesario correr el comando:

```
npm run build
```

El cual habíamos definido en el `package.json` y con esto se generará un archivo `.js` compilado en una carpeta llamada `dist`, dentro de la carpeta estará el archivo llamado `bundle.js`.

Ahora nos vamos al `index.html`, en este archivo ya no haremos referencia al archivo real que trabajamos en ES6, sino que a su compilado ubicado en `dist`, de esta forma:

```
<script type="text/javascript" src="./dist/bundle.js"></script>
```

Y listo, ahora cada vez que hagamos cambios, podemos hacer el bluid para reflejar dichos cambios en la compilación y automáticamente `index.html` apuntará al archivo compilado.

Claramente a nivel de configuraciones es posible automatizar muchas cosas, si profundizas tus conocimientos con Node, te darás cuenta que existen frameworks que hacen todo este proceso de forma automática. Es importante que como estudiantes profundices tus conocimientos investigando sobre los conceptos que se presentan. Por otra parte, todos los desafíos que se presenten desde ahora serán trabajados con babel y webpack, recomiendo replicar dicha configuración de forma local.

Por ultimo, te presento como se debe ver tu carpeta final de proyecto en base al ejempĺo que realizamos.

```
proyecto
|   README.md
|   index.html
|   .babelrc
|   package.json
|   webpack.config.js
└───node_modules/...
|
└───css
|   |   style.css
|   |   otherStyle.css
|
|
|
└───js
|   |   main.js
|
|
|
|
|___img
|   |   img1.png
|
|
|___dist
|   |   bundle.js

...
```

Si tu proyecto al configurarlo queda con una estructura similar y sigues los pasos explicados acá, podrás trabajar sin problemas compilando los archivos necesarios para tus proyectos en ES6. Las configuraciones con webpack pueden ser muy complejas y diversas, te mostraré un ejemplo particular de una configuración extendida, para que te hagas una idea de la complejidad que puede tener un proyecto. Te invito a que investigues y profundices estos temas.

Acá va el ejemplo:

```
'use strict';

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

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CaseSensitivePathsPlugin = require('case-sensitive-paths-webpack-
plugin');
const InterpolateHtmlPlugin = require('react-dev-
utils/InterpolateHtmlPlugin');
const WatchMissingNodeModulesPlugin = require('react-dev-
utils/WatchMissingNodeModulesPlugin');
const eslintFormatter = require('react-dev-utils/eslintFormatter');
const ModuleScopePlugin = require('react-dev-utils/ModuleScopePlugin');
const getClientEnvironment = require('./env');
const paths = require('./paths');

const publicPath = '/';
const publicUrl = '';
const env = getClientEnvironment(publicUrl);
module.exports = {
  devtool: 'cheap-module-source-map',
  entry: [
    // We ship a few polyfills by default:
    require.resolve('./polyfills'),
    require.resolve('react-dev-utils/webpackHotDevClient'),
    // Finally, this is your app's code:
    paths.appIndexJs,

  ],
  output: {
    pathinfo: true,
    filename: 'static/js/bundle.js',
    chunkFilename: 'static/js/[name].chunk.js',
    publicPath: publicPath,
    devtoolModuleFilenameTemplate: info =>
      path.resolve(info.absoluteResourcePath).replace(/\\/g, '/'),
  },
  resolve: {
    modules: ['node_modules', paths.appNodeModules].concat(
      process.env.NODE_PATH.split(path.delimiter).filter(Boolean)
    ),
    extensions: ['.web.js', '.mjs', '.js', '.json', '.web.jsx', '.jsx'],
    alias: {
      'react-native': 'react-native-web',
    },
    plugins: [
      new ModuleScopePlugin(paths.appSrc, [paths.appPackageJson]),
    ],
  },
  module: {
    strictExportPresence: true,
    rules: [
      {
        test: /\.(js|jsx|mjs)$/,
        enforce: 'pre',
        use: [
          {
            options: {
              formatter: eslintFormatter,
              eslintPath: require.resolve('eslint'),

            },
```

```
            loader: require.resolve('eslint-loader'),
          },
        ],
        include: paths.appSrc,
      },
      {
        oneOf: [
          {
            test: [/\.bmp$/, /\.gif$/, /\.jpe?g$/, /\.png$/],
            loader: require.resolve('url-loader'),
            options: {
              limit: 10000,
              name: 'static/media/[name].[hash:8].[ext]',
            },
          },
          // Process JS with Babel.
          {
            test: /\.(js|jsx|mjs)$/,
            include: paths.appSrc,
            loader: require.resolve('babel-loader'),
            options: {
              cacheDirectory: true,
            },
          },
          {
            test: /\.css$/,
            use: [
              require.resolve('style-loader'),
              {
                loader: require.resolve('css-loader'),
                options: {
                  importLoaders: 1,
                },
              },
              {
                loader: require.resolve('postcss-loader'),
                options: {
                  ident: 'postcss',
                  plugins: () => [
                    require('postcss-flexbugs-fixes'),
                    autoprefixer({
                      browsers: [
                        '>1%',
                        'last 4 versions',
                        'Firefox ESR',
                        'not ie < 9',
                      ],
                      flexbox: 'no-2009',
                    }),
                  ],
                },
              },
            ],
          },
          {
            exclude: [/\.(js|jsx|mjs)$/, /\.html$/, /\.json$/],
            loader: require.resolve('file-loader'),
            options: {
```

```
              name: 'static/media/[name].[hash:8].[ext]',
            },
          },
        ],
      },
    ],
  },
  plugins: [
    new InterpolateHtmlPlugin(env.raw),
    new HtmlWebpackPlugin({
      inject: true,
      template: paths.appHtml,
    }),
    new webpack.NamedModulesPlugin(),
    new webpack.DefinePlugin(env.stringified),
    new webpack.HotModuleReplacementPlugin(),
    new CaseSensitivePathsPlugin(),
    new WatchMissingNodeModulesPlugin(paths.appNodeModules),
    new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/),
  ],
  node: {
    dgram: 'empty',
    fs: 'empty',
    net: 'empty',
    tls: 'empty',
    child_process: 'empty',
  },
  performance: {
    hints: false,
  },
};
```