CODER HOUSE

APUNTES DE CLASE

| CLASE 3 |

Babel, Polyfill, Webpack y JSX

Links de interés:

- ¿Qué es y para qué sirve Babel?
- Polyfill: módulos de código para la web moderna
- Webpack: qué es, para qué sirve y sus ventajas e inconvenientes
- Presentando JSX

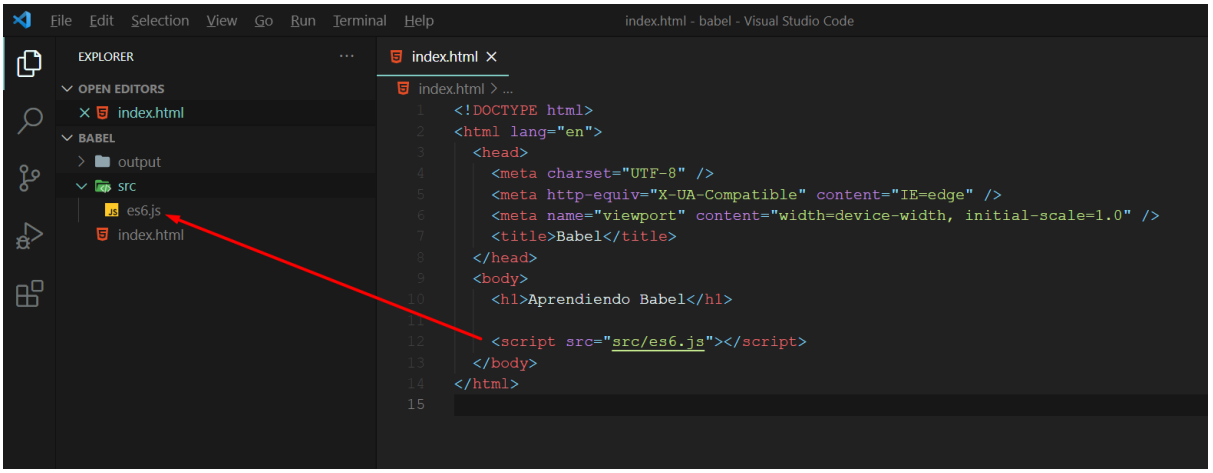
Videos:

- What is Polyfill (Web Development) ***(inglés)***
- JavaScript - ECMAScript 6, Babel y WebPack
- WEBPACK: ¿QUÉ ES?

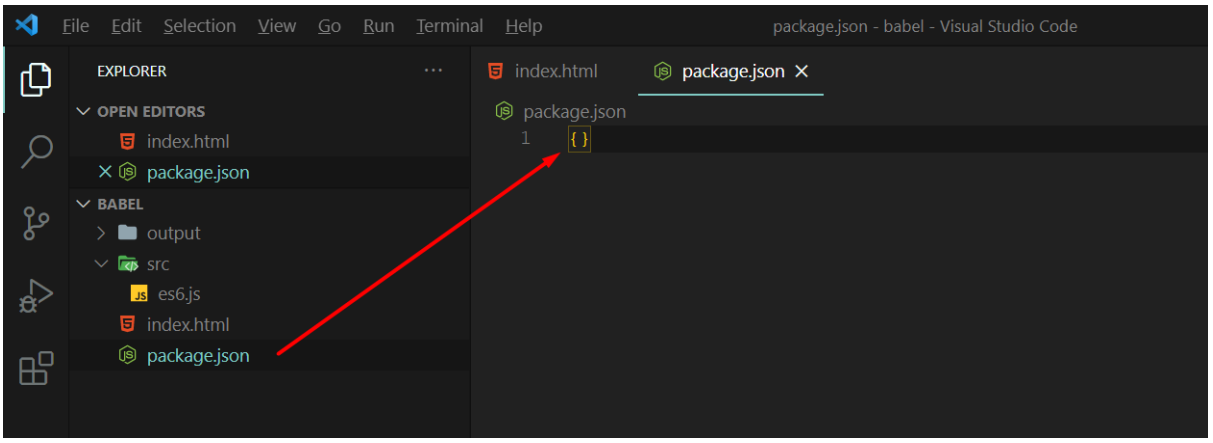
Utilizando Babel en nuestro proyecto de JS

Babel es un transpilador de código que nos permitirá convertir sintaxis de ES6 a código que pueda ser interpretado por todos los browsers, con lo cual estaremos ganando en la retrocompatibilidad de nuestras aplicaciones

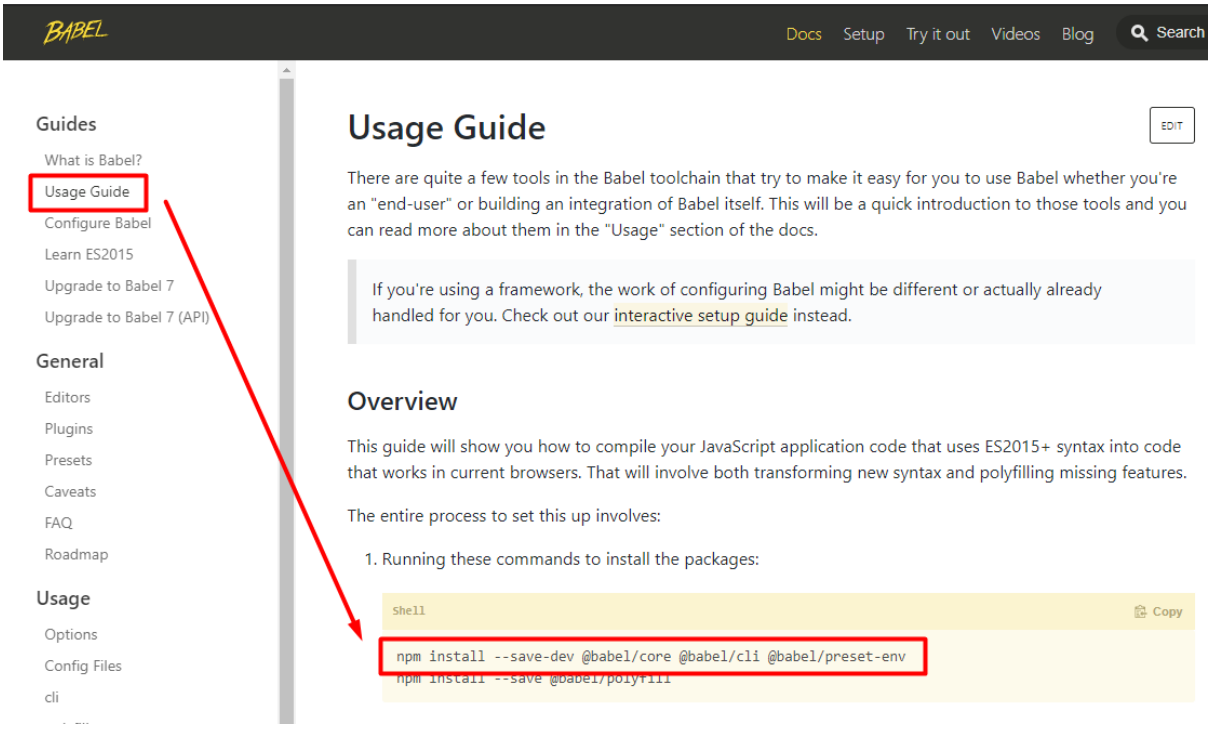
Para comenzar a utilizar Babel de forma local, debemos crearnos un proyecto con dos carpetas dentro, a las cuales podemos llamar - por ejemplo - ***src*** y ***output***. A su vez, debemos crear dos archivos más: uno de ellos es un archivo de js que colocaremos dentro de la carpeta ***src*** al que podemos llamar ***es6.js***, y por otro lado debemos crear un archivo ***index.html*** desde donde llamaremos al js dentro de nuestra carpeta ***src*** mediante la etiqueta <scripts>



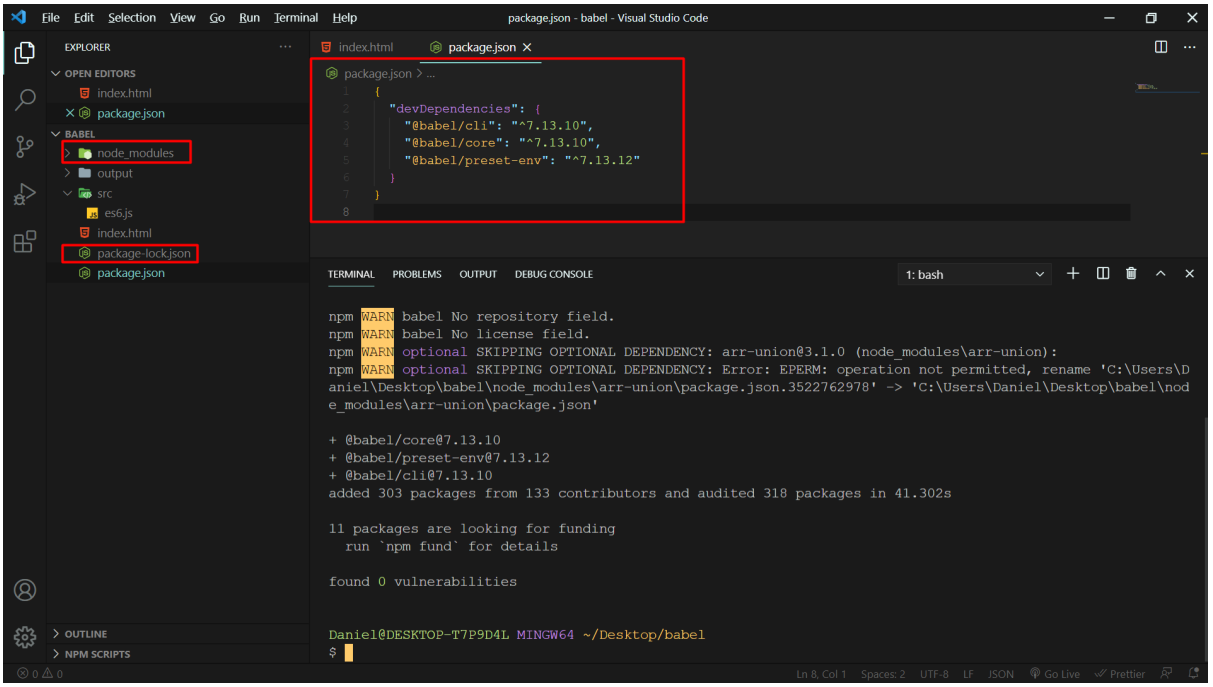
Ahora, debemos crear en el root (la raíz de nuestra carpeta que contiene src, output, y el index.html) un archivo llamado ***package.json*** Dentro de ese archivo generamos un objeto vacío, de la siguiente manera:



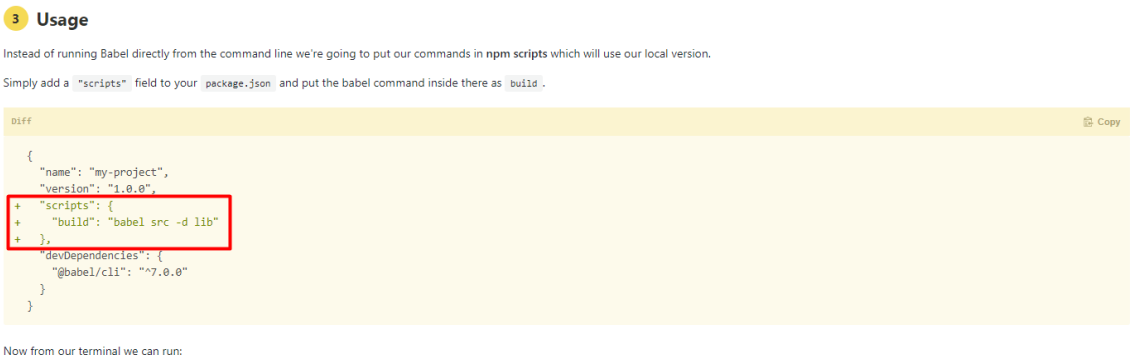
Luego, debemos instalar Babel como devDependencies. Para eso debemos utilizar el siguiente comando de npm, el cual se encuentra en el siguiente link: https://babeljs.io/docs/en/usage



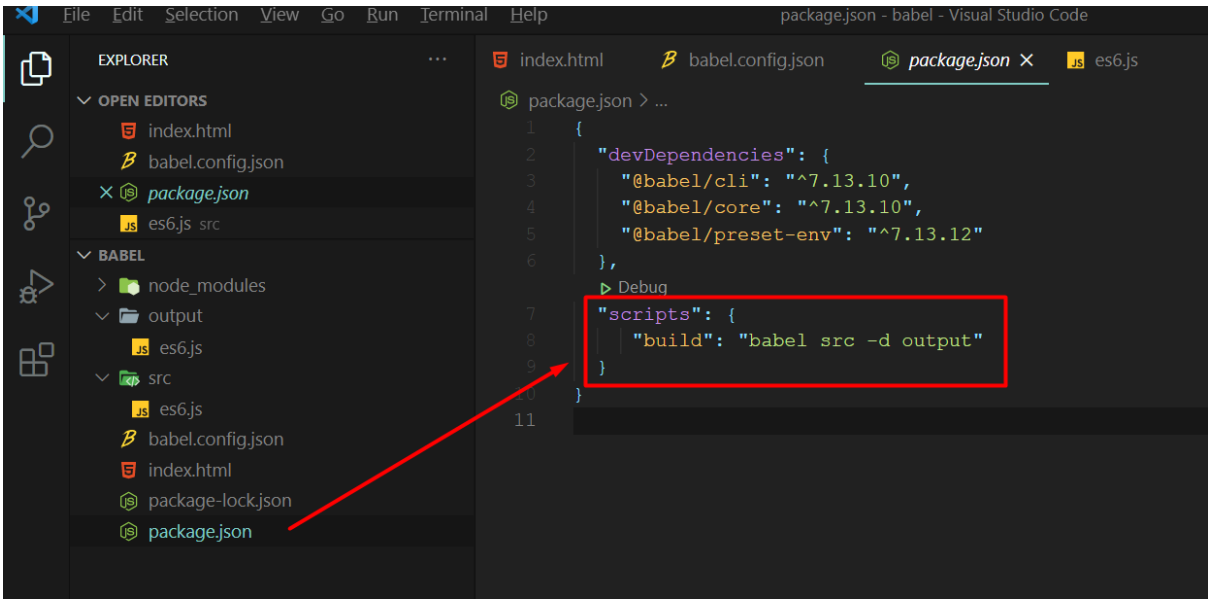
Una vez ejecutado el comando `npm install --save-dev @babel/core @babel/cli @babel/preset-env` por la terminal, podremos ver como el objeto vacío que creamos inicialmente, contiene ahora las dependencias de desarrollo de Babel, y a su vez se nos crea una carpeta ***node_modules***, y un nuevo archivo llamado ***package-lock.json***



Luego debemos ir a la url → https://babeljs.io/setup#installation donde nos encontraremos con un script para correr el build.

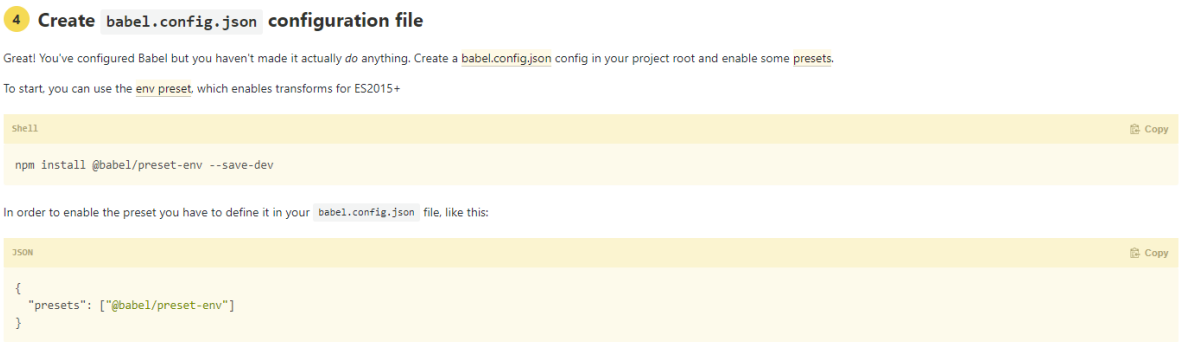


En nuestro visual studio code, debemos utilizar el siguiente script, teniendo en cuenta los nombres de las carpetas que hemos utilizado (src y output).



En el src vamos a tener nuestro archivo de Js con nuestro código de ES6 y en el output veremos el código que se transpila con babel, a una versión que es interpretada por todos los navegadores (de esta manera logramos que nuestra app sea retrocompatible con todos los navegadores).

Por último, debemos crear un archivo llamado **babel.config.json** en el root de nuestro proyecto.



4 **Create `babel.config.json` configuration file**

Great! You've configured Babel but you haven't made it actually *do* anything. Create a babel.config.json config in your project root and enable some presets.

To start, you can use the env preset, which enables transforms for ES2015+

Shell Copy

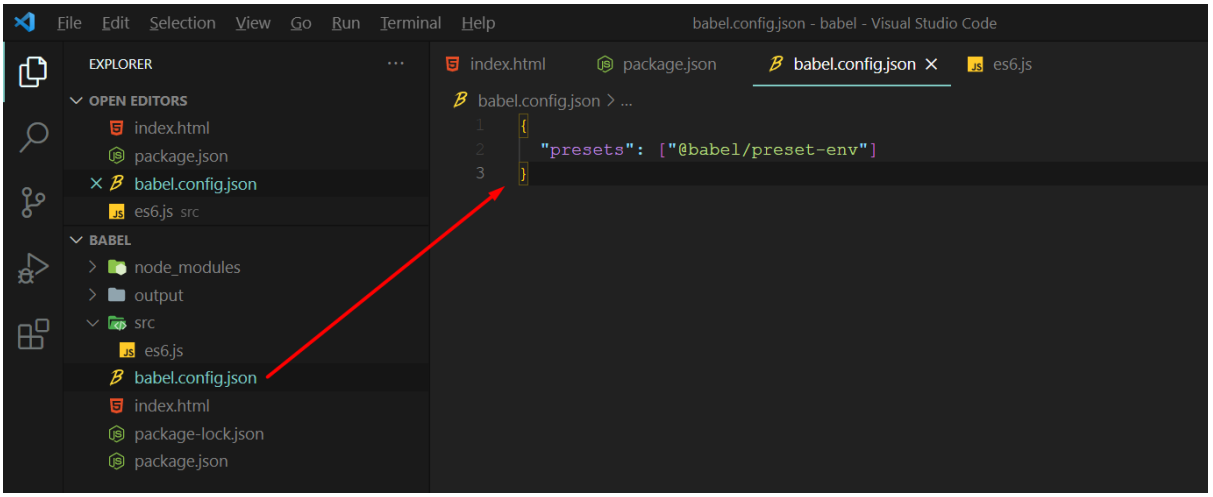
```
npm install @babel/preset-env --save-dev
```

In order to enable the preset you have to define it in your `babel.config.json` file, like this:

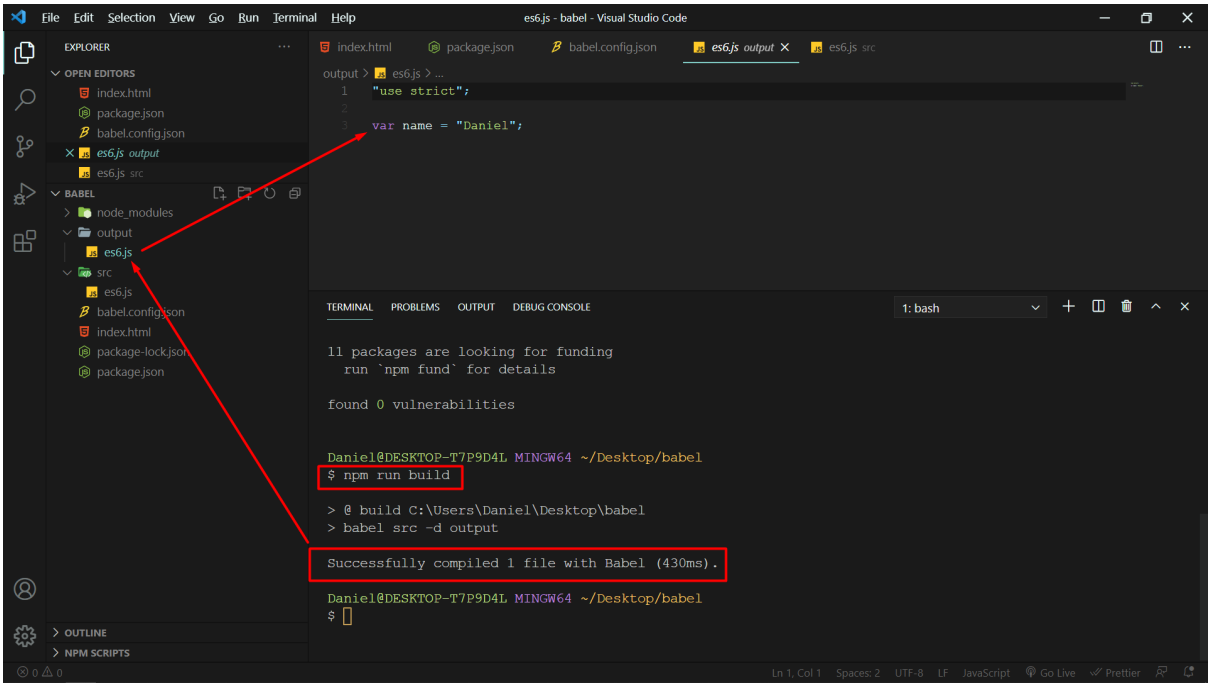
JSON Copy

```
{
  "presets": ["@babel/preset-env"]
}
```

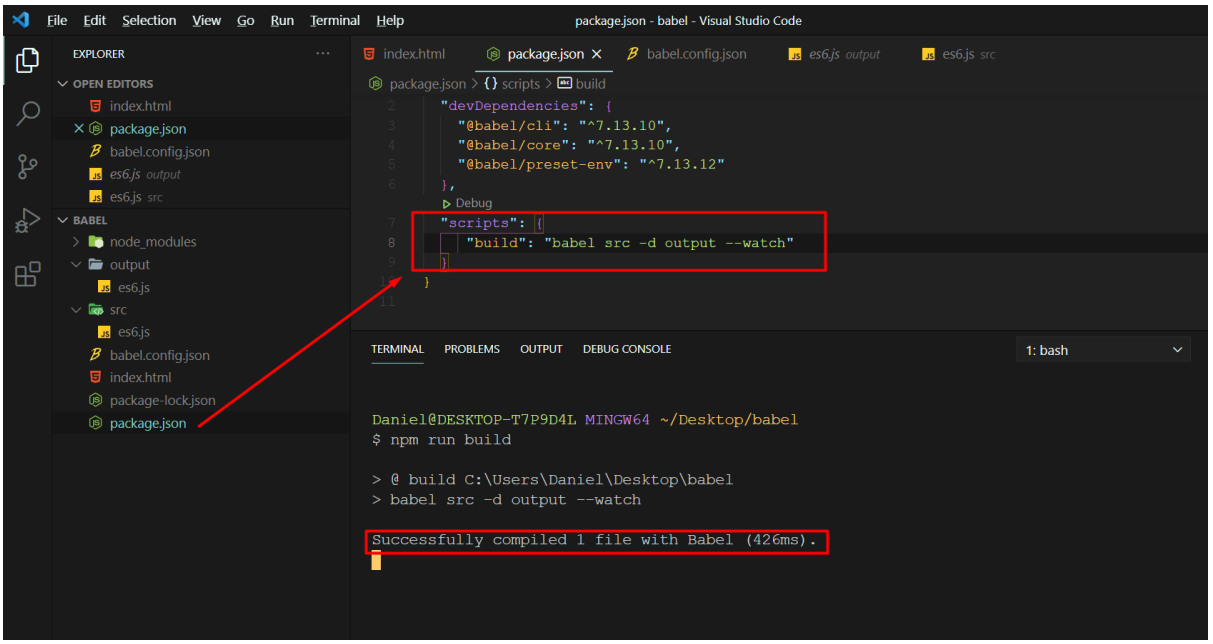
Ese archivo nos quedará de la siguiente manera:



Ahora ya podemos ejecutar la instrucción ***npm run build***, lo que nos ayudará a transpilar nuestro código de Js moderno. Para hacer una prueba, podemos introducir una variable de tipo ***let*** - let es sintaxis de es6 para definir variables - en nuestro archivo de ***es6.js*** alojado dentro de la carpeta ***src***. Al correr la instrucción ***npm run build***, podremos abrir el archivo ***es6.js*** dentro de la carpeta output y verificar como nuestro código de Es6 es compilado a una versión de Js que será interpretada por todos los navegadores.



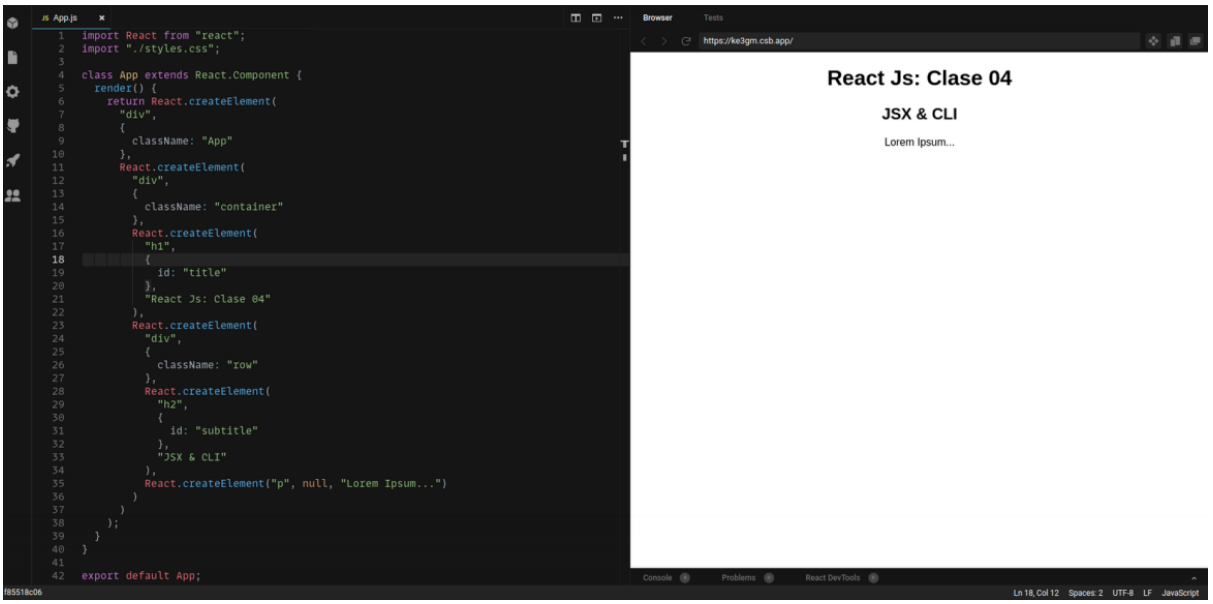
Podemos sumar en el package.json, la instrucción ***--watch*** para que Babel quede escuchando los cambios que realizamos en nuestro archivo de Js de la siguiente manera:



JSX: introducción

JSX es una extensión de sintaxis de JavaScript que nos permite mezclar JS y HTML (XML), de ahí su nombre JavaScript XML.Esta extensión nos facilita la vida pues nos permite escribir un código más limpio, sin tantas repeticiones (DRY), y con muy pocos factores o condiciones a tener en cuenta.

Es posible escribir componentes de React sin JSX, utilizando React.createElement()



Pero el código se vuelve difícil de mantener…

Beneficios de utilizar JSX

- Escribir código similar a HTML para elementos y componentes
- Nos permite escribir menos código
- Evitar el patrón DRY (Don't repeat yourself)
- Embeber expresiones de JS, variables, etc.
- Que el código de React.js sea más simple y elegante .
- Se transpila a JS puro que es entendido por el navegador mediante Babel
- JSX no es obligatorio, es posible usar React.js sin JSX

Con JSX

JS App.js

```jsx
import React from "react";
import "./styles.css";

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <div className="container">
          <h1 id="title">React Js: Clase 04</h1>
          <div className="row">
            <h2 id="subtitle">JSX & CLI</h2>
            <p>Lorem Ipsum...</p>
          </div>
        </div>
      </div>
    );
  }
}

export default App;
```

f85518c06

Sin JSX

App.js

```
import React from "react";
import "./styles.css";

class App extends React.Component {
  render() {
    return React.createElement(
      "div",
      {
        className: "App"
      },
      React.createElement(
        "div",
        {
          className: "container"
        },
        React.createElement(
          "h1",
          {
            id: "title"
          },
          "React Js: Clase 04"
        ),
        React.createElement(
          "div",
          {
            className: "row"
          },
          React.createElement(
            "h2",
            {
              id: "subtitle"
            },
            "JSX & CLI"
          ),
          React.createElement("p", null, "Lorem Ipsum...")
        )
      )
    );
  }
}

export default App;
```

f85518c06