00 ESTRUCTURA DE PROYECTO

Requisitos previos - Tener instalado node.js y cmder (opcional)

Vamos a crear una aplicación React desde 0. Para ello, vamos a utilizar el cmder (ya deberíais tenerlo instalado).

Creamos la carpeta donde vamos a alojar nuestro proyecto y nos posicionamos en ella, para ello ejecutamos el siguiente comando

mkdir ojete && cd ojete

Ejecutamos el comando **npm init**.

Instalaremos webpack como dependencia de desarrollo. (-dev !!)

npm install webpack webpack-cli --save-dev

Instalaremos webpack-dev-server de manera local tambien como dependencia de desarrollo (con esto nos aseguramos que puede ser lanzado en cualquier máquina sin nada instalado de forma global, excepto Node.js)

npm install webpack-dev-server --save-dev

En este punto instalaremos a cholón una serie de plugins y loaders para darle funcionalidades diversas a nuestra configuración de webpack, como manejar los CSS, TypeScript…….

npm install css-loader style-loader file-loader url-loader html-webpack-plugin awesome-typescript-loader mini-css-extract-plugin --save-dev

```
D:\ojete  (ojetecalor@2.0.0)
λ npm install css-loader style-loader file-loader url-loader html-webpack-p
npm WARN awesome-typescript-loader@5.0.0 requires a peer of typescript@^2.7
```

Veremos como nos pide instalar typescript al menos en su versión 2.7

npm install typescript --save-dev

Vamos a modificar el package.json para indicarle la configuración de compilación y arranque. (Ya era hora de abrir el VS Code)

```
"scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "webpack-dev-server --mode development --inline --hot --open",
        "build": "webpack --mode development"
},
```

Necesitamos tambien añadir el archivo tsconfig.json a la carpeta raiz del proyecto, con la siguiente configuración de las opciones de compilación.

```
{
	"compilerOptions": {
		"target": "es6",
		"module": "es6",
		"moduleResolution": "node",
		"declaration": false,
		"noImplicitAny": false,
		"jsx": "react",
		"sourceMap": true,
		"noLib": false,
		"suppressImplicitAnyIndexErrors": true
		},
	"compileOnSave": false,
	"exclude": [
	"node_modules"
	]
}
```

Necesitaremos tambien transpilar ES6 en ES5. Para ello, vamos a utilizar dos módulos de babel.

```
npm install babel-core babel-preset-env --save-dev
```

Babel necesita tambien de configuración, tenemos que crear el archivo .babelrc en la raiz del proyecto.

```
{
	"presets": [
		[
			"env",
			{
			"modules": false
			}
		]
	]
}
```

Para finalizar la configuración de proyecto, vamos a instalar bootstrap

```
npm install bootstrap --save
```

Creamos una subcarpeta que llamaremos src (por convención) y a continuación crearemos dentro el fichero main.ts que será el punto de entrada de nuestra aplicación. De momento solo con algo de este palo…..

./src/main.ts

```
document.write("Estas en el main.ts !");
```

Crearemos también el clásico index.html

./src/index.html

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<title></title>
	</head>
	<body>
		<div class="well">
			<h1>Ejemplo configuración</h1>
		</div>
	</body>
</html>
```

Ahora vamos a crear la configuración del webpack.config.js. Vamos a indicarle que:

- Lance un web dev server
- La transpilación de TypeScript a JavaScript
- Configuración de Bootstrap (con fuentes y demas)
- Generar el resultado de la compilación en una carpeta dist

Creamos el fichero webpack.config.js a nivel de raiz del proyecto e incluimos:

```
let path = require('path');
let HtmlWebpackPlugin = require('html-webpack-plugin');
let MiniCssExtractPlugin = require('mini-css-extract-plugin');
let webpack = require('webpack');

let basePath = __dirname;

module.exports = {
  context: path.join(basePath, "src"),
  resolve: {
    extensions: ['.js', '.ts', '.tsx']
```

```
  },
  entry: [
    './main.ts',
    '../node_modules/bootstrap/dist/css/bootstrap.css'
  ],
  output: {
    path: path.join(basePath, 'dist'),
    filename: 'bundle.js'
  },
  devtool: 'source-map',
  devServer: {
    contentBase: './dist', // Content base
    inline: true, // Enable watch and live reload
    host: 'localhost',
    port: 8085,
    stats: 'errors-only'
  },
  module: {
    rules: [{
          test: /\.(ts|tsx)$/,
          exclude: /node_modules/,
          loader: 'awesome-typescript-loader',
          options: {
            useBabel: true,
          },
        },
        {
          test: /\.css$/,
          use: [MiniCssExtractPlugin.loader, "css-loader"]
        },
        {
          test: /\.(png|jpg|gif|svg)$/,
          loader: 'file-loader',
          options: {
            name: 'assets/img/[name].[ext]?[hash]'
          }
        },
    ],
  },
  plugins: [
    //Generate index.html in /dist => https://github.com/ampedandwired/html-webpack-plugin
    new HtmlWebpackPlugin({
      filename: 'index.html', //Name of file in ./dist/
      template: 'index.html', //Name of template in ./src
      hash: true,
    }),
    new MiniCssExtractPlugin({
      filename: "[name].css",
      chunkFilename: "[id].css"
    }),
  ],
};
```

Vamos a probar que funciona, ejecutemos la aplicación con **npm start**

01 – WELCOME TO REACT

Summary steps:

- Install react and react-dom libraries.
- Install react and react-dom typescript definitions.
- Update the index.html to create a placeholder for the react components.
- Create a simple react component.
- Wire up this component by using react-dom.

Vamos a comenzar instalando las librerias de react y react-dom como dependencias de proyecto

```
npm install react react-dom --save
```

A continuación instalaremos tambien las definiciones TypeScript para las librerias que acabamos de instalar, pero como dependencias de desarrollo.

```
npm install @types/react @types/react-dom --save-dev
```

Actualizaremos el ./src/index.html con el fin de crear un punto de entrada para los componentes React

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title></title>
 </head>
 <body>
   <div class="well">
     <h1>Ejemplo configuración</h1>
     <div id="root"></div>
   </div>
 </body>
</html>
```

Ahora, vamos a crear un componente sencillo, Hello.tsx, aunque primero por ordenación de código crearemos la carpeta Components dentro de src

./src/components/hello.tsx

```
import * as React from 'react';

export const HolaReactComponent = () => {
  return (
    <h2> Hola ! Soy un componente React</h2>
  );
}
```

Ahora, renombraremos el archivo main.ts a main.tsx, y conectaremos el componente al contenedor donde lo vamos a representar

./src/main.tsx

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

import { HolaReactComponent } from '../src/components/hello';

ReactDOM.render(
  <HolaReactComponent/>,
  document.getElementById('root')
);
```

Modificaremos el webpack.config y cambiaremos el punto de entrada de la aplicación a main.tsx.

```
module.exports = {
    context: path.join(basePath, "src"),
    resolve: {
        extensions: ['.js', '.ts', '.tsx']
    },
    entry: [
        // './main.ts',
        './main.tsx',
        '../node_modules/bootstrap/dist/css/bootstrap.css'
    ],
```

Comprobamos que funciona todo correctamente ejecutando un **npm start**

02 – PROPIEDADES

Vamos a empezar a utilizar la potencia de React, modificaremos el componente HolaReact que hemos creado para que reciba propiedades que le pasaremos vía interfaz. Quedaría de la siguiente forma.

```
import * as React from 'react';

interface Props {
  username: string;
}

export const HolaReactComponent = () => {
export const HolaReactComponent = (props: Props) => {
  return (
    <h2> Hola ! Soy un componente React</h2>
    <h2> Hola usuario: {props.username}</h2>
  );
}
```

Ahora modificaremos main.tsx que es donde estamos consumiendo el componente, y le pasaremos un valor a la propiedad que acabamos de añadir al componente.

```
ReactDOM.render(
  <HolaReactComponent/>,
  <HolaReactComponent username="Enjuto Mojamuto"/>,
  document.getElementById('root')
);
```

03 - ESTADO

Ahora vamos a introducir otro concepto esencial en React, el concepto de estado.

Como paso inicial, vamos a crear un componente que llamaremos app.tsx y cuyo fin es contener el resto de componentes de primer nivel de la aplicación.

./src//app.tsx

```
import * as React from 'react';
import {HolaReactComponent} from './hello';

export const App = () => {
  return (
    <HolaReactComponent username="Julen Lopetegui" />
  );
}
```

Cambiaremos en el punto de entrada de nuestra aplicación el componente que se va a renderizar en primer lugar, por el app.tsx que acabamos de crear.

./src/main.tsx

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import { App } from '../src/app';

ReactDOM.render(
<HolaReactComponent username="Enjuto Mojamuto" />,
  <App/>,
  document.getElementById('root')
);
```

Vamos a rehacer el componente app.tsx como clase que extienda la funcionalidad de un componente de React al uso, y maneje la funcionalidad de estado, que es la que vamos a representar ahora y tenga un ciclo de vida. Para que tenga un estado inicial, lo vamos a tener que definir en el constructor.

./src/components/app.tsx

```
import * as React from 'react';
import { HolaReactComponent } from './hello';

interface Props {

}

interface State {
  userName: string;
}

export class App extends React.Component<Props, State> {
  constructor(props: Props) {
    super(props);

    this.state = { userName: 'Julen Lopetegui' };
  }

  public render() {
    return (
      <>
        <HolaReactComponent username={this.state.userName} />
      </>
    );
  }
}
```

Crearemos el componente nameEdit, que utilizaremos para actualizar el userName del componente padre.

./src/components/nameEdit.tsx

```
import * as React from 'react';

interface Props {
  userName : string;
  onChange : (event) => void;
}

export const NameEditComponent = (props : Props) => {
  return (
  <>
      <label>Update name:</label>
      <input value={props.userName} onChange={props.onChange}/>
  </>
  );
}
```

A las etiquetas html vacias (<> y </>) se les denomina Fragmentos y sirven para devolver varios elementos al mismo nivel sin tener que añadir un adicional. Esto solo está disponible en React desde la versión 16.2. La propiedad onChange la utilizamos para ejecutar una función en el Componente padre, como veremoos ahora.

Vamos a modificar el componente app.tsx para cambiar el userName del estado inicial utilizando el componente que acabamos de crear, para ello crearemos una función que setee el estado. En este punto cabe recordar que el state de un componente es seteable, pero las props son inmutables dentro del ciclo de vida del componente.

```
import * as React from 'react';
import { HolaReactComponent } from './hello';
import { NameEditComponent } from './nameEdit'

interface Props {

}

interface State {
  userName: string;
}

export class App extends React.Component<Props, State> {
  constructor(props: Props) {
    super(props);

    this.state = { userName: 'Julen Lopetegui' };
  }
  private setUsernameState = (event) => {
    this.setState({ userName: event.target.value });
  }

  public render() {
    return (
      <>
        <HolaReactComponent username={this.state.userName} />
        <NameEditComponent userName={this.state.userName} onChange={this.setUsernameState} />
      </>
    );
  }
}
```

Finalmente, comprobamos que funciona todo como esperamos.

04 – CALLBACK

En el ejemplo anterior estamos obligando al componente padre a controlar el valor que le está devolviendo el componente hijo, vamos a refactorizarlo para tener controlado lo que nos va a llegar, en este caso un string. Vamos a transformar el componente NameEdit de un componente stateless a un componente de clase.

```
import * as React from 'react';

interface IProps {
  // userName : string;
  initialUserName:string;
  //onChange : (event) => void;
  onNameUpdated:(newName: string)=>any;
}

interface IState {
    editingName: string;
   }


// export const NameEditComponent = (props : Props) => {
//   return (
//   <>
//       <label>Update name:</label>
//       <input value={props.userName} onChange={props.onChange}/>
//   </>
//   );
// }

export class NameEditComponent extends React.Component<IProps, IState> {

  constructor(props: IProps) {
    super(props);

    this.state = { editingName: this.props.initialUserName };
  }

 private onChange = (event) => {
    let newName=event.target.value;
    this.setState({ editingName: newName });
  }

 private onNameSubmit = ():void => {
    this.props.onNameUpdated(this.state.editingName);
  }

  public render() {
    return (
      <div>
        <label>Update Name:</label>
        <input value={this.state.editingName} onChange={this.onChange} />
        <input type="submit" value="Change" className="btn btn-default" onClick={this.onNameSubmit} />
      </div>
    );
  }
}
```

Modificaremos también nuestro App.tsx, pues hemos cambiado el valor de una de las propiedades del componente hijo, así como tenemos control sobre el tipo que nos devuelve.

```
export class App extends React.Component<Props, State> {
  constructor(props: Props) {
    super(props);

    this.state = { userName: 'Julen Lopetegui' };
  }
  // private setUsernameState = (event) => {
  //   this.setState({ userName: event.target.value });
  // }

  private setUsernameState = (newName:string) => {
    this.setState({ userName: newName });
  }

  public render() {
    return (
      <>
        <HolaReactComponent username={this.state.userName} />
        {/* <NameEditComponent userName={this.state.userName} onChange={this.setUsernameState} /> */}
        <NameEditComponent initialUserName={this.state.userName} onNameUpdated={this.setUsernameState}
/>
      </>
    );
  }
```

Verificamos que seguimos teniendo el comportamiento esperado