Curso de React.js AVANZADO

# ¿Qué es React.js?

Es un sistema de plantillas para gestionar las vistas que utiliza un lenguaje de marcado llamado JSX, similar a HTML  
.  
**Características básicas de react**  
.

* Está basado en **componentes**: Toda la UI se divide en elementos más pequeños llamados componentes, en react, todo es un componente,
* Es [**declarativo**](https://dev.to/itsjzt/declarative-programming--react-3bh2): **expresa la lógica de un cálculo sin describir su flujo de control**, es un estilo de programación en el que el que se define “qué” es la solución sin importar cómo se llegó a ella.

# Proyecto y tecnologías que usaremos

En este curso realizaremos una aplicación muy parecida a Instagram, llamada petgram. Tendremos nuestras rutas, gestión de usuarios y likes.

Utilizaremos como empaquetador y transpilador:

* [Webpack](https://platzi.com/cursos/webpack/)
* [Babel](https://babeljs.io/)

Estilado con CSS en JS con:

* [styled-components](https://www.styled-components.com/)

Como linter utilizaremos:

* [Standard JS](https://standardjs.com/)

Para fetching (obtención) de datos:

* [GraphQL](https://platzi.com/cursos/graphql/)
* [React Apollo](https://www.apollographql.com/docs/react/)

Para el enrutado de la SPA utilizaremos:

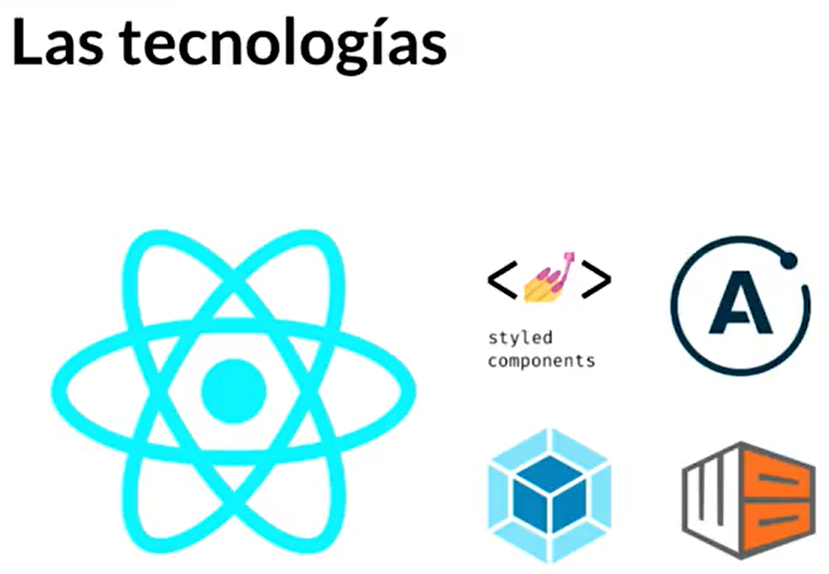
* [Reach Router](https://reach.tech/router)

Para las buenas prácticas utilizaremos:

* [Lighthouse](https://developers.google.com/web/tools/lighthouse/?hl=es)
* [Cypress](https://www.cypress.io/)

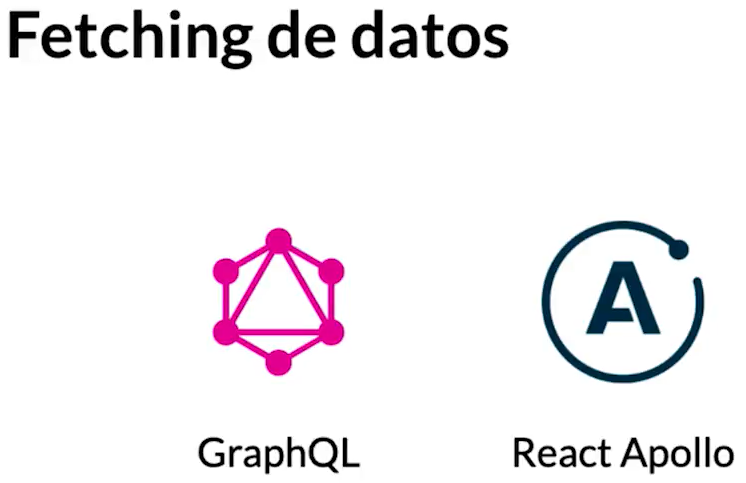
Por último haremos SEO, PWA y Deploy con:

* [React Helmet](https://github.com/nfl/react-helmet)
* [Workbox](https://developers.google.com/web/tools/workbox/)
* [Progressive Web App](https://platzi.com/cursos/pwa-react-js/)
* [Deply con Vercel](https://platzi.com/cursos/deploy-now/)



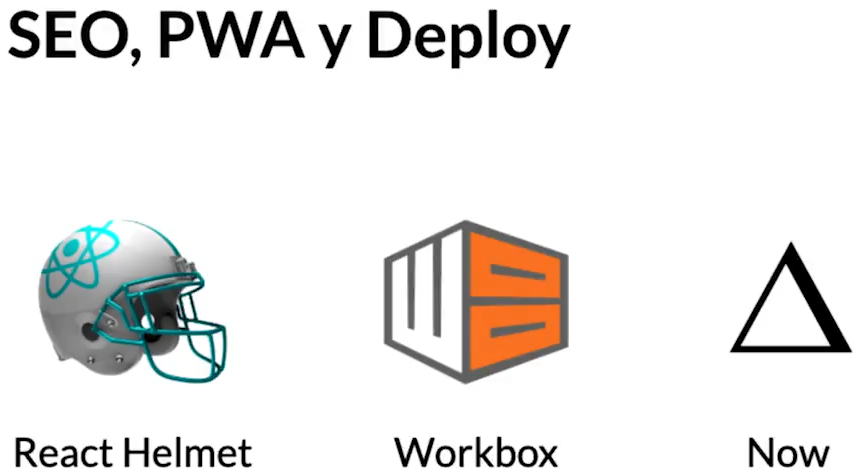












# Clonando el repositorio e instalando Webpack

Pasos para iniciar nuestro proyecto:

* **Paso 1:** Vamos a clonar el repositorio desde [github.com/midudev/curso-platzi-react-avanzado](https://github.com/midudev/curso-platzi-react-avanzado) usando git clone URL\_DEL\_REPO en nuestra consola.
* **Paso 2:** Vamos a instalar webpack y webpack-cli como dependencias de desarrollo con: npm i webpack wepack-cli --save-dev.
* **Paso 3:** Crearemos una carpeta llama src y dentro de ella un archivo index.js en el cual colocaremos solo un console.log('Empezamos el curso!').
* **Paso 4:** Crearemos en el root de nuestro proyecto un archivo webpack.config.js el cual tendrá toda la configuración de webpack
* **Paso 5:** Instalaremos html-webpack-plugin con: npm i html-webpack-plugin --save-dev.
* **Paso 6:** Instalaremos webpack-dev-server con: npm i webpack-dev-server --save-dev.
* **Paso 7:** Añadiremos un nuevo script llamado dev: "dev": "webpack-dev-server".

// webpack.config.js

**const** HtmlWebpackPlugin = require("html-webpack-plugin")

module.exports = {

output: {

filename: 'app.bundle.js'

},

plugins: [

**new** HtmlWebpackPlugin()

]

}

Nota:

Comando que corregir

npx webpack --entry ./src/index.js

"dev": "webpack serve --open",

# Instalación de React y Babel

En esta clase vamos a configurar React instalando las dependencias npm i react react-dom y Babel para poder transpilar código *jsx* a JavaScript Vanilla con: npm i @babel/core @babel/preset-env babel-loader @babel/preset-react --save-dev.

Ahora añadiremos en nuestro webpack.config.js lo siguiente:

// webpack.config.js

{/\*...\*/}

module.exports = {

{/\*...\*/}

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader',

options: {

presets: [

'@babel/preset-env',

'@babel/preset-react'

]

}

}

}

]

}

}

# Linter, extensiones y deploy con Vercel

En esta clase haremos que el desarrollo sea más ágil y correcto siguiendo los siguientes pasos:

* Vamos a instalar StandardJS como dependencia de desarrollo con: npm i standard --save-dev. StandardJS nos va a servir de Linter para una mejor escritura de JavaScript/React.
* Agregaremos un nuevo script en nuestro package.json: ""lint"": ""standard"".
* Ahora vamos a ignorar aquellos archivos que no queremos que el Linter arregle, añadiremos en nuestro package.json lo siguiente:

""standard"": [

""ignore"": [

""/api/\*\*""

]

]

* Ahora, queremos que nuestro Linter nos detecte los errores a medida que vamos escribiendo, para hacer esto añadimos lo siguiente a nuestro package.json:

""eslintConfig"": {

""extends"": [

""./node\_modules/standard/eslintrc.json""

]

}

* Ahora debemos tener lo siguiente en nuestro editor de código para que funcione todo al pie de la letra:
  + Tener instalada la extensión [ESLint](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint" \t "_blank)
  + Si quieres que al guardar los cambios se formatee tu código deberás tener instalada la extensión [Prettier](https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode" \t "_blank)
  + Tener las siguientes configuraciones en VSCode:
    - Format On Save: false
    - Prettier: Eslint Integration: true
    - Eslint: Auto Fix On Save: true
* Ahora utilizaremos [Vercel](https://zeit.co/login" \t "_blank) para hacer el deploy de nuestro proyecto.
* Descargaremos e instalaremos [Vercel](https://zeit.co/download" \t "_blank) para que nos registre de una manera mucho más fácil los tokens de acceso y podamos continuar con el curso.
* Entraremos a la carpeta de api y notaremos que ya tiene un archivo Vercel.json que preparamos para ti con toda la configuración necesaria para hacer el deploy.
* Para desplegar el proyecto del backend haremos lo siguiente en nuestra terminar:
  + cd api
  + Cambiamos el name de la aplicación en el Vercel.json
  + Finalmente ejecutamos Vercel
* Ahora para desplegar nuestro front haremos lo siguiente:
  + Crearemos un archivo Vercel.json en el root de nuestro proyecto con lo siguiente:

{

""version"": 2,

""name"": ""petgram"",

""builds"": [

{

""use"": ""@Vercel/static-build"",

""src"": ""package.json""

}

],

""routes"": [

{

""src"": ""(.\*).js"",

""dest"": ""/$1.js""

},

{

""src"": ""(.\*).json"",

""dest"": ""/$1.json""

},

{

""src"": ""/.\*"",

""dest"": ""index.html""

}

]

}

* En nuestro package.json añadiremos el siguiente script: ""Vercel-build"": ""npm run build"".
* Finalmente en la raíz de nuestro proyecto ejecutaremos Vercel para que nos dé una URL en la que se verá nuestro proyecto."

Seria bueno que actualizarán este video. Now ya no es now ahora es vercel y tiene algunas diferencias, puede llegar a ser confuso. En todo caso lo que deben hacer es:

1. Instalar vercel a traves del comando **npm i -g vercel**
2. Despues de instalar vercel deben hacer **vercel login** y verificar su correo electronico, para que la instancia de versel local este conectada con el remoto y de esta forma evitar los tokens.
3. Luego si realizar los pasos que dice el profe, pero con la difencia que el comando **no es now, es vercel**

Si alguno tuvo inconvenientes en VSCode con la siguiente validación

* Format On Save: false
* Prettier: Eslint Integration: true
* Eslint: Auto Fix On Save: true

Instalé las dependencias de desarrollo

npm i prettier eslint-config-prettier eslint-plugin-prettier -D

Y configure manualmente en Open Settings (settings.json) ubicado en la parte superior derecha del editor (Tiene forma de {} o como un archivo" y revisan si su archivo de configuración tiene las siguiente lineas y sino las insertan

"editor.codeActionsOnSave": {

"source.fixAll.eslint": **true**

}

"eslint.format.enable": **true**,

"prettier.eslintIntegration": **true**,

eslint.autofixonsave: true esta obsoleto por eso debe reemplazarse con editor.codeActionsOnSave

# ¿Qué es CSS-in-JS?

Antes la forma en la que construíamos nuestras aplicaciones era con HTML-centric:

<button className='button button-red'>

Click here!

button>

**const** button = document.querySelector('button')

button.addEventListener('click', **function** () {

doSomething()

})

.button {

border-radius: 4px;

}

.button-red {

background: red;

color: #fff;

}

Luego pasamos a JavaScript-centric:

render () {

**return** (

<button

className='button button-red'

onClick={doSomething}

>

Click here!

button>

)

}

.button {

border-radius: 4px;

}

.button-red {

background: red;

color: #fff;

}

Ahora hoy en día pasamos a CSS-in-JS nos permite no solo hacer el jsx si no colocar los estilos en el mismo JavaScript, de la siguiente manera:

**const** Button = styled.button`

border-radius: 4px;

${props => props.accent && `

background: red;

color: #fff;

`}

render () {

return (

Click here!

)

}

Con CSS-in-JS no debemos preocuparnos por tener dos clases igual, no nos debemos preocupar por el Critical Path CSS, tendremos un CSS mucho más óptimo.

En esta clase utilizaremos [styled-components](https://www.styled-components.com/" \t "_blank) el cual es una biblioteca para CSS-in-JS pensada más que todo para React

## Styled-components

Los styled-components son una de las nuevas formas de usar CSS en JavaScript moderno. Está basado en los CSS modules, una forma de escribir CSS que se enfoca el que el alcance de los estilos funcione por componente y no se filtren a ningún otro elemento de la página.  
.  
Los styled components se basan en crear componentes de carácter estético para hacer código más semántico y evitar los problemas de especificidad que trae el CSS tradicional.  
.  
Los styled components le permiten escribir CSS simple en sus componentes sin preocuparte por las colisiones de nombres de clase y haciendo uso del poder de JavaScript para manejar estilos con lógica (basado en props).  
.  
**Nota: en styled components escribes CSS exacto, así que prácticamente todo lo que puedes hacer en CSS lo puedes hacer en styled components y se escribe igual.**  
.  
Es bastante sencillo, sólo tenemos que seguir 3 pasos:

1. importar styled components

**import** styled **from** 'styled-components'

1. crear un componente con estilo basado en algún elemento html

**const** DivConEstilo = styled.div`

color:red;

background:black;

`

1. Utilizar el componente como si fuera cualquier componente de React

**const** Contenedor = () =>{

**return** (

<DivConEstilo>

Hola Styled Component

</DivConEstilo>

)

}

<https://styled-components.com/>

# Creando nuestro primer componente: Category

npm install styled-components

Vamos a construir nuestro primer componente el cual será y usaremos [styled-components](https://www.styled-components.com/" \t "_blank) para hacer los estilos de nuestro proyecto.

Para que el código del styled-component esté mejor formateado en nuestro editor utilizaremos esta extensión [vscode-styled-components](https://marketplace.visualstudio.com/items?itemName=jpoissonnier.vscode-styled-components).

Una buena practica seria definir desde styled-components, los globalStyles, de esta manera podras definir los valores que mejor te convengan… aqui un ejemplo:

**import** { createGlobalStyle } **from** 'styled-components';

**const** GlobalStyles = createGlobalStyle`

html {

box-sizing: border-box;

font-size: 10px;

}

\*, \*:before, \*:after {

box-sizing: inherit

}

body {

padding: 0;

margin: 0;

font-size: 1.5rem;

font-family: sans-serif;

}

h1, h2, h3, h4, h5, p {

margin: 0;

}

a {

text-decoration: none;

color: #393939

}`;

**export** **default** GlobalStyles;

# Creando ListOfCategories y estilos globales

Para crear nuestros estilos globales con styled-components haremos lo siguiente:

// src/GlobalStyles.js

**import** { createGlobalStyle } **from** 'styled-components'

**export** **const** GlobalStyle = createGlobalStyle`

html {

box-sizing: border-box;

font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-serif;

}

\*, \*::before, \*::after {

box-sizing: inherit;

}

ul, li, h1, h2, h3, p, button {

margin: 0;

}

ul {

list-style: none;

}

button {

background: transparent;

border: 0;

outline: 0;

}

body {

background: #fefefe;

height: 100vh;

margin: 0 auto;

max-width: 500px;

overscroll-behavior: none;

width: 100%;

}

#app {

box-shadow: 0 0 10px rgba(0, 0, 0, 0.05);

overflow-x: hidden;

min-height: 100vh;

padding-bottom: 10px;

}`