 arielZarate / HenryAll

Public

Code

Issues

Pull requests

Actions


Projects

Wiki


Security

Insights


Settings

 main


HenryAll / FT-M2-master / 07-React-Estilos /

 arielZarate ...


8 days ago




..

 .vscode


8 days ago

 demo


19 days ago

 homework


8 days ago


 README.json

27 days ago

 README.md

27 days ago

 README.md





Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

Dando Estilos en React

Este es un tema muy particular, ya que se está cambiando la filosofía que teníamos de usar CSS como lo conocemos para dar estilos, es decir en forma *global* y *en cascada*. Ahora hay muchos desarrolladores que creen que esta forma no es *Escalable* y que hay que adoptar nuevas formas de dar estilos a cada Componente, en particular hablan de tener estilos *locales* y no globales.

Esta discusión de paradigmas todavía no tiene un ganador claro, hay defensores y detractores de ambas formas por todos lados. Tomen con pinzas todo lo que lean online sobre este tema.

Estilos Inline

Una de las formas de dar Estilos a los Componentes es usando el atributo `style` del mismo. En react esta propiedad recibe un *objeto JavaScript* y no una *Css String* como en HTML nativo. Por lo tanto, vamos a tener que cambiar un poco la sintaxis de las reglas CSS. Veamos el ejemplo de la [documentación](#) de react:

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

Como vemos, no podemos usar los mismos nombres de las propiedades CSS porque tiene conflicto con el operador `-`, por lo tanto decidieron que es mejor utilizar los nombres de las reglas usando *camelCase*.

Existen traductores de CSS a CSS JavaScript como [este](#).

Clases

En los ejemplos anteriores usamos una clase para setear el estilo de un Link activo. La clase la habíamos definido en el `index.html` en donde se cargaba todo nuestro código, por lo tanto esa definición era accesible por los Links. Del mismo modo, si tenemos clases definidas de esa manera (podríamos usar un framework como Bootstrap, por ejemplo), vamos a poder dar el nombre de las clases que tendrán nuestros Componentes y sus childrens, usando el keyword `className`. Como se imaginan el keyword `class` está reservado en JavaScript y no se puede utilizar para eso. Por ejemplo

```
function HelloWorldComponent() {
  return <div className='activado'>Hello World!</div>;
}
```

```
}
```

Frameworks

Veamos como podemos usar Webpack, para *requerir* un Framework y utilizarlo dentro de nuestra app. Para esto, vamos a necesitar instalar algunos loaders extras, para que Webpack sepa como manejar un `require('./estilo.css)` ya que no se tratan de archivos js!

Para eso vamos a instalar varios loaders :

- **css-loader** : Sirve para requerir archivos `.css` y tenerlos como objetos JS.
- **script-loader**: Algunos frameworks utilizan jQuery u otras librerías como dependencias, con este loader vamos a poder incluirlas también en nuestra app.
- **style-loader**: Por último nos va a faltar inyectar el CSS en nuestro HTML, lo vamos a hacer usando este loader.
- **url-loader**: Bootstrap viene con sus propias fuentes y utiliza archivos como `.woff` o `.ttf` , vamos a usar este loader para poder inyectar estos archivos en nuestra página.

Para instalar los loaders hacemos: `npm install css-loader script-loader style-loader url-loader --save` . Tambien tenemos que instalar, en este caso Bootstrap y jQuery: `npm install jquery bootstrap --save` .

Ahora tenemos que configurar de nuevo nuestro `webpack.config` para hacer uso de los loaders nuevos, para eso vamos a agregar entradas en el arreglo `loaders` :

```
module.exports = {
  entry: [
    './index.js'
  ],
  module: {
    loaders: [
      {
        test: /\.js|\.jsx$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        query: { presets: ['es2015', 'react'] }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
      {
        test: /\.?(png|woff|woff2|eot|ttf|svg)$/,
        loader: 'url-loader?limit=100000'
      }
    ]
  },
}
```

```

    output: {
      filename: "index_bundle.js",
      path: __dirname + '/dist'
    }
  }
}

```

Hemos agregado `loaders` para tres casos distintos, cuando el archivo es `.css` vamos a usar primero el `style-loader` y luego `css-loader`. Esto requiere el archivo `.css` y luego lo inyecta a la página como un stylesheet.

Como dijimos, vamos a usar el `url-loader` para varios archivos que tienen que ver con fuentes como: `.woff`, `.ttf`, `.eot`, además de algunas imagenes que puede incluir el framework (`.png` o `.svg`).

Todavía no hemos usado el `script-loader`, porque por ahora sólo estamos incluyendo `Css`.

Por último nos falta agregar Bootstrap en nuestro proyecto, por lo tanto en nuestro `index.js` vamos a agregar la siguiente línea:

```
require('bootstrap/dist/css/bootstrap.css');
```

Básicamente, estamos *importando* el archivo `.css` de bootstrap que habíamos instalado con `npm`, y este es pasado por los loaders que hemos definido y así llega a nuestra página.

Genial! Ahora ya tenemos por cargado Bootstrap en nuestra App, podemos verlo en los estilos de los Headers y Links!

Ahora agregemos la clase `btn btn-default` a cada Link de nuestra Nav y vemos cómo queda:

```

<IndexLink className="btn btn-default" to="/" activeClassName="active" >Home</IndexLi
<Link className="btn btn-default" to="/about" activeClassName="active" >Componente2</
<Link className="btn btn-default" to="/ejemplos" activeClassName="active">Componente3

```

Ahora, vamos a probar agregar una NavBar de Bootstrap, podemos copiarla [acá](#).

Recuerden cambiar los `class` por `className`, y los comentarios de HTML tampoco funcionan en JSX.

Ahora, si abrimos la página vamos a ver que se ve cómo debería, pero por ejemplo, el dropdown no funciona. Esto se debe a que el NavBar de Bootstrap utiliza una librería propia de JS y además jQuery, y nosotros todavía no lo hemos incluido ninguna.

Veamos como hacerlo.

Por empezar debemos incluir el archivo `.js` en nuestro `index.js`:

```
require('script-loader!jquery/dist/jquery.min.js');
require('bootstrap/dist/css/bootstrap.css');
require('script-loader!bootstrap/dist/js/bootstrap.min.js');
```

Ahora sí estamos usando el `script-loader`, cuando cargamos jQuery. En este caso lo usamos, porque necesitamos que jQuery este accesible de manera global, así el `js` de bootstrap pueda acceder a él.

También existen otras formas de hacer lo mismo con webpack, pero indicando qué cosas necesitamos en el `webpack.config.js`. Para probarlo vamos a comentar las líneas de los requires:

```
//require('script!jquery/dist/jquery.min.js');
//require('bootstrap/dist/css/bootstrap.css');
//require('bootstrap/dist/js/bootstrap.min.js');
```

Y vamos a agregar los siguiente a nuestro `webpack.config.js`:

```
module.exports = {
  entry: [
    'script-loader!jquery/dist/jquery.min.js',
    'bootstrap/dist/js/bootstrap.min.js',
    'bootstrap/dist/css/bootstrap.css',
    './index.js'
  ],
  externals: {
    jquery: 'jQuery'
  },
  module: {
    loaders: [
      {
        test: /\.js|\.jsx$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        query: { presets: ['es2015', 'react'] }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
      {
        test: /\.?(png|woff|woff2|eot|ttf|svg)$/,
        loader: 'url-loader?limit=100000'
      }
    ]
  },
}
```

```

    output: {
      filename: "index_bundle.js",
      path: __dirname + '/dist'
    }
  }
}

```

Lo que hicimos fue decirle a Webpack que nos incluya en el `bundle` a los archivos necesarios, lo hicimos agregando cada uno de ellos en el arreglo `entry`. Noten que a `jquery.js` tuvimos que avisarle que utilice el loader `script`, ya que necesitamos que sea *global*.

Webpack es genial, pero también es muy complejo. A veces se hace difícil pensar que está haciendo por atrás. Además hay miles de formas de hacer lo mismo, lo que no lo hace más sencillo.

Nuestros propios estilos

Con *Webpack* encontramos la forma de *importar* estilos. Veamos que sucede cuando queremos importar nuestro propio `.css`. Primero comencemos definiendo un archivo `estilos.css` simple, en una carpeta separa, por ejemplo `styles`.

```

.prueba {
  background-color: red;
}

```

Bien, ahora vamos a ir a nuestro Componente `Home` y vamos a importarlo y usar la clase `prueba` en un `div`:

```

var React = require('react');
var Link = require('react-router').Link;

require('../styles/estilos.css');

module.exports = React.createClass({
  render: function(){
    return (
      <div className=' prueba '>
        Hola, Henry!!
      </div>
    )
  }
});

```

Antes de mostrar el resultado, vamos a ir a otro Componente, por ejemplo `About` y usar la misma clase 'prueba', pero sin *importar* la hoja de estilos:

```

var React = require('react');
var Link = require('react-router').Link;

module.exports = React.createClass({
  render: function(){
    return (
      <div>
        <h1>About</h1>
      </div>
    )
  }
});

```

Como vemos, el estilo fue importado de manera *global* y lo podemos usar en cualquier Componente. Esto para algunos puede ser bueno, pero para otros no. Veamos porqué sucedió esto y como utilizar otro patrón.

Esto ocurre, porque en nuestro `webpack.config.js` hemos definido que cada archivo `.css` que sea importado, pase por los loaders: `style-loader` y `css-loader`. El primero justamente lo que hace, es importar el `css` como si fuera una hoja de estilo convencional, por lo tanto le da el comportamiento que todos conocemos.

Ahora, si no quisiéramos que esto ocurra, tendremos que usar otro approach. Vamos a usar uno conocido como `CSS-Modules`, básicamente lo que vamos a hacer es dejar que webpack haga un poco de *magia* usando los loaders que ya tenemos, y que cuando importemos el archivo CSS nos devuelva un objeto JS, con estilos listos para usar con react y con un *namespace* local:

```

{
  test: /\.ncss$/,
  loader: 'style-loader!css-loader?modules&importLoaders=1&localIdentName=[name]__[lo
},

```

Primero vamos a agregar esta entrada en los loaders, vamos a tener que usar otra extensión (`.css` ya pasa por otros loaders distintos) para que los archivos que usemos como CSS Modules puedan ser indentificados por Webpack, en este caso yo elegí `.ncss`. Si se fijan, la magia sucede en el string que le pasamos al `loader`.

Ahora, vamos a crear un archivo `estilos.ncss` con el mismo contenido que `estilos.css`. Y lo vamos a requerir en nuestro Componente `Home`:

```

var prueba = require('../styles/estilos.ncss').prueba;

module.exports = React.createClass({
  render: function(){

```

```

    console.log(prueba);
    return (
      <div className= {prueba}>
        Hola, Henry!!
      </div>
    )
  }
});

```

Como ven, requerimos el archivo, y el nombre de la clase como propiedad del mismo. Y luego utilizamos la variable donde lo guardamos como `className`.

Bien! Hemos logrado importar un archivo css de manera local. El Componente `About` sigue teniendo la clase `prueba`, pero no se activa con el CSS que hemos importado. Esto se debe a que Webpack le puso un hash al nombre de la clase para que sea único, en nuestro ejemplo la clase `.prueba` terminó llamandose `.estilos__prueba__2wKns`, emulando así un *namespace* local en CSS.

Múltiples clases

Si usamos este método para importar un archivo `.css` que contenga múltiples clases, vamos a poder acceder a ellas como *propiedades* del objeto en donde importes el `css`.

Por ejemplo, si este fuera el `css` que importamos:

```

// estilos.ncss
.prueba {
  background-color: red;
}

.title {
  color: blue;
}

.size {
  font-size: 25px;
}

```

Entonces podríamos usar las clases de la siguiente manera:

```

var s = require('../styles/estilos.ncss');

module.exports = React.createClass({
  render: function(){
    return (
      <div className= {s.prueba}>
        <h1 className={s.title}>Hola, Henry!!</h1>
        <p className=[s.title, s.size].join[' ']>Prueba</p>
      </div>
    )
  }
});

```



```
    </div>
  )
}
});
```

Como vemos, si quisieramos que un mismo elemento tenga múltiples clases, podemos usar el siguiente *truco*:

```
[s.title, s.size].join(' ')
```

Esto funciona porque lo que hace el `style-loader` es darle un nombre único a cada clase del archivo `css`, y lo guarda en el objeto donde importamos bajo en una propiedad con el nombre original de la clase, y como valor el valor nuevo de la clase. Por ejemplo, si importamos `estilos.ncss` en el objeto `s` este sería algo así:

```
var s = require('../styles/estilos.ncss');
// s = {
//   prueba: '.estilos__prueba__2wKns',
//   title: '.estilos__title__3dsns;
//   size: '.estilos__size__7d8f8;
// }
```

Esto es un ejemplo ilustrativo, probablemente el objeto `s` no debe ser exactamente así.

Sabiendo esto, si hacemos un `join` de un arreglo formado por cada clase que usemos, vamos a obtener un string con los nombres únicos de las clases concatenados.

Este cambio de filosofía es relativamente nuevo y todavía están surgiendo ideas nuevas y nuevas formas de hacer las cosas, así que por ahora está sucediendo lo mismo que en este comic:



Están apareciendo muchas formas distintas de incluir CSS en React, todavía no se puede decir cual es la mejor, todas tienen sus pros y sus contras. Así que hay que tener paciencia, probar varias y quedarse con la que más nos gusta. Lo importante es entender que está sucediendo por detrás.

Homework

Completa la tarea descrita en el archivo [README](#)