



Mini Proyecto ReactJS [1]


Introducción

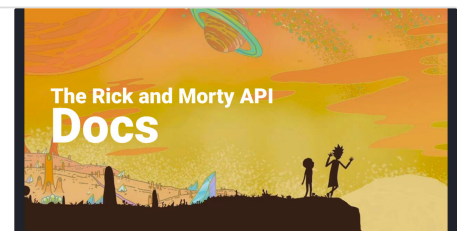
¡Por fin a ha llegado el momento que tanto esperábamos!, hoy creamos nuestro primer proyecto de React para iniciarnos a tope con esta tecnología. Vamos a trabajar en una pequeña aplicación que nos pinte un listado de personajes de React bada en Rick & Morty. Lets go! 🔥

Antes de comenzar te dejamos la documentación de la API sobre la que vamos a trabajar:

Documentation

This documentation will help you get familiar with the resources of the Rick and Morty API and show you how to make different queries, so that you can get the most out of it. <https://rickandmortyapi.com/graphql> Base url:

 <https://rickandmortyapi.com/documentation/#get-all-characters>



Guía

Arrancamos nuestro proyecto haciendo uso de vite , ya que vamos a trabajar con ello de forma normal.

```
npm create vite@latest
```

Una vez que tenemos nuestro proyecto creado, lo primero que vamos a hacer es mockear los datos simulando los de la API e iremos explicando alguna de las particularidades de React. Por ello en el fichero `App.jsx` creamos una variable con un array de objetos que simula los datos de la API.

```
const charactersMock = [
  {
    id: 1,
    name: "Rick Sanchez",
    status: "Alive",
  },
  {
```

```

      id: 2,
      name: "Morty Smith",
      status: "Alive",
    },
  ],
];

const App = () => {
  return <h1>Hello React!</h1>;
};

export default App;

```

Una vez tenemos nuestro mock podemos crear un `state` que se inicia con este conjunto de datos.

```

import React from "react";

const charactersMock = [
  {
    id: 1,
    name: "Rick Sanchez",
    status: "Alive",
  },
  {
    id: 2,
    name: "Morty Smith",
    status: "Alive",
  },
];

const App = () => {
  const myState = React.useState(charactersMock);

  const characters = myState[0];
  const charactersSet = myState[1];

  return <h1>Hello React!</h1>;
};

export default App;

```

`myState` es un array que en su primer valor tiene un “getter” con los datos que le hemos pasado, `charactersMock`, y como segundo es un “setter” que nos permite realizar cambios, por ello podemos hacer destructuring y hacerlo así más sencillo de entender y usar.

```

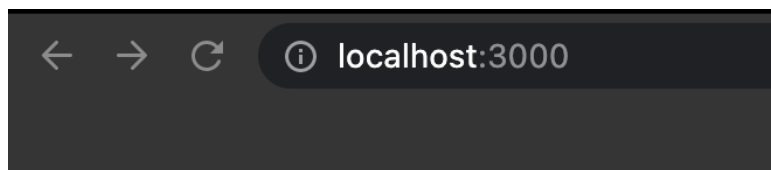
const [characterList, setCharacterList] = React.useState(charactersMock);

```

Ahora vamos a ver un poquito de la potencia de JSX realizando un map de nuestros characters para pintarlos en el DOM.

```
return (  
  <div>  
    {characterList.map((character) => (  
      <p>{character.name}</p>  
    ))}  
  </div>  
);
```

Si ahora levantamos nuestro proyecto deberíamos ver algo así:



Rick Sanchez

Morty Smith

Pero si abrimos nuestra consola de Chrome veremos que tenemos el siguiente error:

```
react-jsx-dev-runtime.development.js:117 Warning: Each child in a list should have a unique "key" prop.  
Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.  
    at p  
    at App (http://localhost:3000/static/js/bundle.js:40:80)
```

Esto simplemente nos indica que cada elemento del bucle que hemos generado tiene que tener una `key` propia para guardarlo como referencia, es decir podemos escoger una de las propiedades de nuestros objetos para convertirla en el id único que no pide React.

Las `key` son una información adicional que podemos añadir a nuestros elementos de React para gestionar adecuadamente su ordenación en listas dinámicas sin crear bugs inesperados. ¡Nos encontraremos con esto en el futuro!

```
{characterList.map((character) => (<p key={character.id}>{character.name}</p>))}
```

Vamos a pintar el resto de propiedades de nuestros elementos.

```
import React from "react";
import logo from "../logo.svg";
import "../App.css";

const charactersMock = [
  {
    id: 1,
    name: "Rick Sanchez",
    status: "Alive",
  },
  {
    id: 2,
    name: "Morty Smith",
    status: "Alive",
  },
];

const App = () => {
  const [characterList, setCharacterList] = React.useState(charactersMock);
  return (
    <>
      {characterList.map((character) => (
        <div key={character.id}>
          <h2>id: {character.id}</h2>
          <h2>name: {character.name}</h2>
          <h2>status: {character.status}</h2>
        </div>
      ))}
    </>
  );
};

export default App;
```

Si os extraña en el `return` las etiquetas `<>` `</>` son denominadas React Fragments y nos sirve para agrupar contenido sin necesidad de una

etiqueta `HTML`.

Mini-ejercicio

Tomando de ejemplo la guía anterior os proponemos crear vuestro propio mock con vuestra temática preferida y pintar un listado en ella haciendo uso de las ventajas que nos proporciona `JSX`.

Eliminar nuestro mock

¡Genial! Ya pintamos nuestros datos mock y vemos algunos personajes, pero la idea inicial era realizar una petición a la API de Rick & Morty y pintar esos datos, por ello vamos hacer uso de otro Hook como es el `useEffect` (veremos en profundidad más adelante) para lanzar nuestra petición.

El `useEffect` lo que nos va permitir es tener disponible el ciclo de vida de nuestro componente para ejecutar funciones en distintos momentos.

Este Hook se lanzará siempre cuando se haya creado el componente, es decir cuando está montado, de tal modo que nos permite pasar info al componente y React se encargará de re-renderizar con los datos obtenidos sin bloquear la vista inicial al cliente web.

Adicionalmente, podemos invocarlo tantas veces como necesitemos utilizando un array de dependencias que veremos en detalle en próximos posts.

```
//Se ejecuta solamente cuando se monte el componente
React.useEffect(()=> {
}, [])

// Se ejecuta cuando se monte el componente y siempre que haya re-render
React.useEffect(()=> {
})

// Se ejecuta cuando se monte el componente y
// cada vez que cambia nuestros personajes en un re-render
React.useEffect(()=> {
}, [characterList])
```

Vamos a lanzar nuestra petición, si aún no sabéis cómo funciona un fetch o el uso de async/await tenemos un post con todo lo que necesitas saber para trabajar con React.

```
React.useEffect(() => {
  (async () => {
    let data = await fetch(`https://rickandmortyapi.com/api/character/`).then(
      (res) => res.json()
    );

    setCharacterList(data.results);
  })();
}, []);
```

Nuestro código en `App.jsx` sería.

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

const App = () => {
  const [characterList, setCharacterList] = React.useState([]);

  React.useEffect(() => {
    (async () => {
      let data = await fetch(`https://rickandmortyapi.com/api/character/`).then(
        (res) => res.json()
      );

      setCharacterList(data.results);
    })();
  }, []);

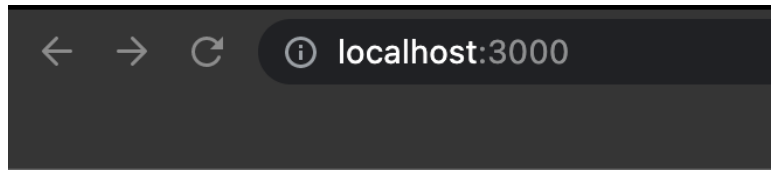
  return (
    <>
      {characterList.map((character) => (
        <div key={character.id}>

          <h2>name: {character.name}</h2>

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

export default App;
```

Obteniendo el mismo resultado que cuando teníamos el mock.



Rick Sanchez

Morty Smith

Mini-ejercicio

En la API de Rick & Morty obtenemos más información que el nombre de cada uno de los personajes por lo tanto os proponemos recoger más información y renderizarla en la lista. Como mínimo estaría bien:

1. Recoger el nombre del personaje.
2. Recoger el id del personaje → usarlo como key.
3. Recoger la image proporcionada por la API.
4. Recoger el status e informar si nuestro personaje está “alive”
 - a. Si un personaje no está vivo hacer uso de JSX para no renderizar ese personaje.
5. Recoger el lugar de origen.
6. Estilar el listado haciendo uso de los CSS modules y obtener una visualización más atractiva.