UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Programador Web Inicial
Front End Developer
React JS Componentes y Virtual DOM

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

Bloques temáticos:

- Map
- Trabajando con nuestra aplicación
- API REST
- Promise
- Aplicando una consulta API a nuestro código
- Aplicando axios en lugar de fetch

Map

El método `map()` crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. Por ejemplo:

```
{this.props.cervezas.map(cerveza=><li>{cerveza}</li>)}
```

Cervezas es un array enviado como propiedad desde el llamado al componente:

```
class App extends Component {
    render() {
        return (
            <div className="App">
                <h1>Bienvenido a Red Social UTN FRBA</h1>
                <Productos cervezas={['Quilmes', 'Brahma']} />
            </div>
        );
    }
}
```

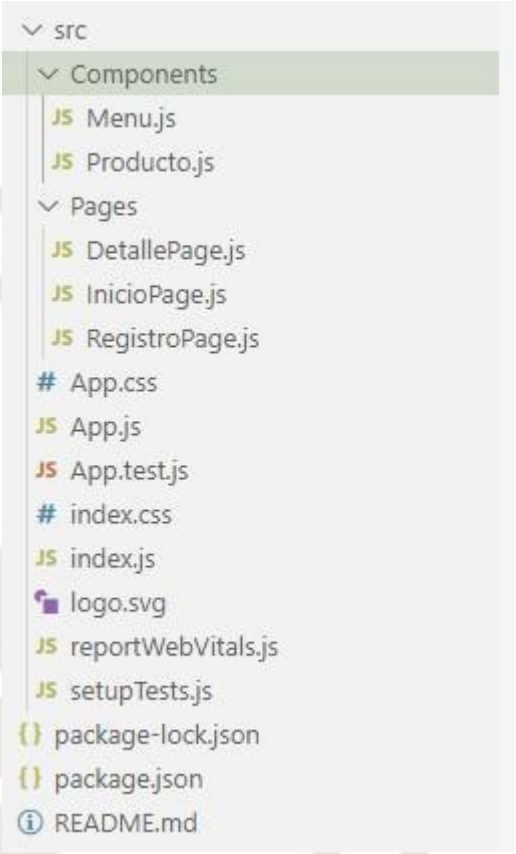
Podemos ver que el primer parámetro recibido por la función dentro de `map` hace referencia a cada objeto devuelto por la iteración del array.

Trabajando con nuestra aplicación

Para mejorar la organización de nuestro código, podemos crear dos directorios:

- `Pages` En este directorio ubicaremos aquellos componentes que sean una página completa. Por ejemplo, el componente Home
- `Components` En este directorio colocaremos todo componente que sea parte de una página, pero no sea una página completa. Por Ejemplo, el componente Menú

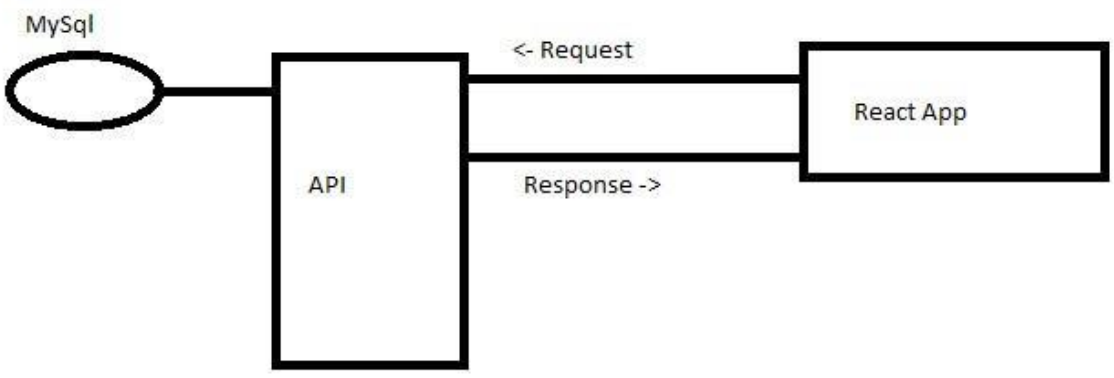
Nos queda de la siguiente manera:



Es una estructura propuesta, cada alumno puede utilizar la estructura deseada.

API REST

REST es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON.
Es una alternativa en auge a otros protocolos estándar de intercambio de datos como SOAP (Simple Object Access Protocol), que disponen de una gran capacidad, pero también mucha complejidad. **A veces es preferible una solución más sencilla de manipulación de datos como REST.**



Ventajas

- **Separación entre el cliente y el servidor:** el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos. Eso tiene algunas ventajas cuando se hacen desarrollos.
  Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.

- **Visibilidad, fiabilidad y escalabilidad.** La separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas.
  Se puede migrar a otros servidores o realizar todo tipo de cambios en la base de datos, siempre y cuando los datos de cada una de las peticiones se envíen de forma correcta.
  Esta separación facilita tener en servidores distintos el front y el back y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.

- **La API REST siempre es independiente del tipo de plataformas o lenguajes**: la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo.
  Con una API REST se pueden tener servidores PHP, Java, Python o Node.js. Lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

Promise

Una `Promise` (promesa en castellano) es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Una promesa puede ser creada usando su constructor. Sin embargo, la mayoría de la gente son consumidores de promesas ya creadas devueltas desde funciones.

Esencialmente, una promesa es un objeto devuelto al cual enganchas las funciones callback, en vez de pasar funciones callback a una función.

Por ejemplo, en vez de una función del viejo estilo que espera dos funciones callback, y llama a una de ellas en caso de terminación o fallo:

```
function exitoCallback(resultado) {
    console.log("Tuvo éxito con " + resultado);
}
function falloCallback(error) {
    console.log("Falló con " + error);
}
hazAlgo(exitoCallback, falloCallback);
```

Las funciones modernas devuelven una promesa a la que puedes enganchar tus funciones de retorno

```
hazAlgo().then(exitoCallback, falloCallback);
```

A diferencia de las funciones callback pasadas al viejo estilo, una promesa viene con algunas garantías:

- Las funciones callback nunca serán llamadas antes de la terminación de la ejecución actual del bucle de eventos de JavaScript.
- Las funciones callback añadidas con `.then` serán llamadas después del éxito o fracaso de la operación asíncrona, como arriba.
- Pueden ser añadidas múltiples funciones callback llamando a `.then` varias veces, para ser ejecutadas independientemente en el orden de inserción.
- Pero el beneficio más inmediato de las promesas es el encadenamiento.

```
hazAlgo().then(function (resultado) {
    return hazAlgoMas(resultado);
})
    .then(function (nuevoResultado) {
        return hazLaTerceraCosa(nuevoResultado);
    })
    .then(function (resultadoFinal) {
        console.log('Obtenido el resultado final: ' + resultadoFinal);
    })
    .catch(falloCallback);
```

Aplicando una consulta API a nuestro código

Modificaremos nuestro código anterior para consultar los datos de nuestros usuarios a una API REST

Componente función

En el componente `Home` creamos los siguientes hooks:

```
const [loading, setLoading] = useState(true)
const [productos, setProductos] = usestate([])
```

Luego aplicamos el hook `useEffect` para la ejecución de la consulta al servicio al renderizarse el componente

```
useEffect(
    () => {
        fetch("https://jsonfy.com/items")
        .then(res => res.json())
        .then(data => {
            console.log("data", data)
            setLoading(false)
            setProductos(data)
        })
    },
    []
)
```

Agregamos condiciones para mostrar el listado de usuarios solo en el caso de que se haya completado la respuesta desde el api. En caso contrario mostramos el loader o bien un error.

```
if (loading) {
    return (
        <div> loading...
        </div>
    )
} else {
    return (
        <div>
            <h1>Productos</h1>
            <button onCLick={() => setDestacados(true)}> Ver
Destacados</button>
            <button onclick={() => setDestacados(false)}>Ocultar
Destacados</button>
            {productos.map(producto => <Producto datos={producto}
destacados={true} />)}
        </div >
    )
}
```

Este es el resultado obtenido:

Componente de clase

En el componente `Home` realizaremos lo siguiente:

```
constructor(props) {
    super(props)
    this.state = {
        error: null,
        isLoaded: false,
        perfiles: []
    };
}
```

Definimos en el estado del componente las variables `error`, `isLoaded`, `perfiles`

Luego modificamos el ciclo de vida utilizado anteriormente por `componentDidMount`

```
componentDidMount(){
    fetch("https://jsonplaceholder.typicode.com/users")
        .then(res => res.json())
        .then(
            (result) => {
                console.log(result)
                this.setState({
                    isLoaded: true,
                    perfiles: result
                });
            },
            // Note: it's important to handle errors here
            // instead of a catch() block so that we don't swallow
            // exceptions from actual bugs in components.
            (error) => {
                console.log(error)
                this.setState({
                    isLoaded: true,
                    error
                });
            })
}
```

Dentro de este método incluimos el llamado a la API REST, como vemos debemos trabajar la respuesta con el promise (then)

```
render() {
    const { error, isLoaded, perfiles } = this.state;
    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoaded) {
        return <div>Loading...</div>;
    } else {
        return (
            <ul>
                {perfiles.map(
                    perfil => <Perfil datos={perfil} />)
                }
            </ul>
        );
    }
}
```

En el render agregamos condiciones para mostrar el listado de usuarios solo en el caso de que se haya completado la respuesta desde la api.

En caso contrario mostramos el loader o bien un error. Este es el resultado obtenido:

Aplicando axios en lugar de fetch

Es una librería JavaScript que puede ejecutarse en el navegador y que nos permite hacer sencillas las operaciones como cliente HTTP, por lo que podremos configurar y realizar solicitudes a un servidor y recibiremos respuestas fáciles de procesar.

Axios es una alternativa que nos brinda multitud de ventajas:

- La API es unificada para las solicitudes Ajax.
- Está optimizado para facilitar el consumo de servicios web, API REST y que devuelvan datos JSON.
- De fácil utilización y como complemento perfecto para las páginas convencionales.
- Pesa poco, apenas 13KB minimizado. Menos aún si se envía comprimido al servidor.
- Compatibilidad con todos los navegadores en sus versiones actuales.

Instalar axios

Debemos ingresar por consola al directorio raíz de nuestro proyecto y ejecutar: **npm install axios**

```
C:\sites\react\pwa2c>pm install axios
```

Una vez instalado podemos consumir directamente sus métodos, a continuación, veremos cómo hacerlo creando un directorio y concepto de services.

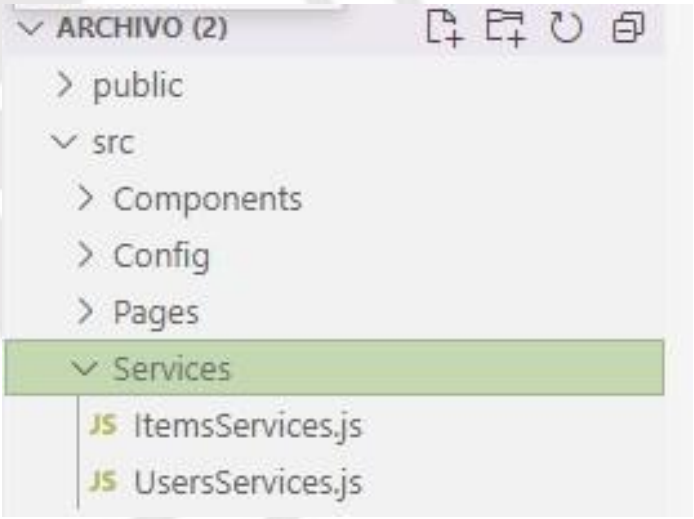
Vamos a crear un directorio config y dentro un archivo llamado axios.js

Dentro de dicho archivo vamos a realizar un requiere de la biblioteca y crear una instancia en axios la cual vamos a exportar:

```
import axios from "axios"
export default axios.create({
    baseURL: "https://jsonfy.com/"
})
```

Vemos que créate recibe un parámetro el cual nos permite definir la url base

Por otro lado, dentro de src creamos un directorio llamado “services” y dentro de dicho directorio crearemos un archivo js por cada “entidad”. Ejemplo si consultamos ítems creamos ítems.js. Si la entidad es compras, compras.js



ItemServices.js tendrá el siguiente contenido:

```
import instance from "../Config/axios"
export function getAll(query = "") {
    return instance.get("items" + query)
}
export function getById(id) {
    return instance.get("items/" + id)
}
```

En este caso lo que estamos haciendo es un import del archivo axios.js, para poder importar la instancia creada.

Por otro lado, creamos dos métodos, uno que nos permite consultar todos los ítems retornados por jsonfy y el otro un ítem dado su id.

Por último, en el componente en el cual queramos usar alguno de estos métodos hacemos un import de la función a utilizar y el llamado al método. Por ejemplo:

```
import { getAll } from "../Services/ItemsServices"
//Definicion de clase
function InicioPage() {
    const [loading, setloading] = useState(true)
    const [productos, setProductos] = useState([])
    const [destacados, setDestacados] = useState(false)
    console.log("Database", firebase.db)
    useEffect(
        () => {
            getAll()
                .then(({ data }) => {
                    console.log("data", data)
                    setLoading(false)
                    setProductos(data)
                })
        }, [])
}
```

Axios nos permite tener algunas funcionalidades extra respecto de fetch, por ejemplo, la definición de la url base o el uso de interceptors. No es necesaria su utilización, ya que con fetch podemos realizar una app react completa.

Para ver más acerca de axios: https://github.com/axios/axios

Bibliografía y Webgrafía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://reactjs.org/docs/faq-ajax.html https://github.com/axios/axios