Creación de una Pokédex básica

Diego Fernández Gutiérrez

1. Introducción

En este corto proyecto se tratará de crear desde cero una Pokédex básica en la que se incorporarán elementos básicos que suelen tener los Pokémon, tales como sus nombres, el tipo y una breve descripción. Todos estos elementos se sacan de la API de Pokemon llamada PokeAPI, la cual de forma gratuita nos permite seleccionar todos los datos que queramos de los Pokemon.

Este trabajo ha sido realizado con React.js y Typescript, los cuales se tratan de lenguajes que nos permiten crear aplicaciones web en las que nos ofrece la opción de interactuar directamente a través de un navegador. Además, en futuras iteraciones, se tiene previsto mejorarlo para que se muestre en un teléfono móvil.

2. Desarrollo

2.1. Creación del proyecto con Create Router Dom

Para esta primera fase, vamos a tratar de crear nuestra pokedex inicial. Para esto, nos vamos al siguiente enlace para que podamos crear una plantilla utilizando Typescript. Entonces, vamos a nuestro directorio de trabajo, y creamos una carpeta llamada, por ejemplo, "pokedex" y desde una terminal (Linux en mi caso) introducimos el siguiente comando (puede que tarde un poco).

```
$ npx create-react-app . --template typescript
```

Nota. En función de donde estemos situados en la terminal, debemos cambiar el '.' por el directorio actual para decir dónde queremos que se descargue todo, como en nuestro caso estábamos en el directorio "pokedex" se ha puesto de esta forma.

Una vez que ya tenemos nuestra plantilla descargada, vamos a comprobar que funcione correctamente, por ello, vamos a Visual Studio Code (o la misma terminal), y abrimos una terminal en donde escribiremos lo siguiente para ejecutar la aplicación.

```
$ npm start
```

Esperamos un poco, y veremos en la Figura 1 cómo nos aparece el siguiente mensaje, indicándonos que podemos ver la ejecución del programa en un navegador.

```
Starting the development server...

One of your dependencies, babel-preset-react-app, is importing the
"@babel/plugin-proposal-private-property-in-object" package without
Compiled successfully!

You can now view pokedex in the browser.
```

Figura 1: Mensajes de correcta compilación del programa

Por ello, se abrirá por defecto, un navegador con la dirección *localhost:3000*, y como vemos en la Figura 2 se ha ejecutado correctamente.

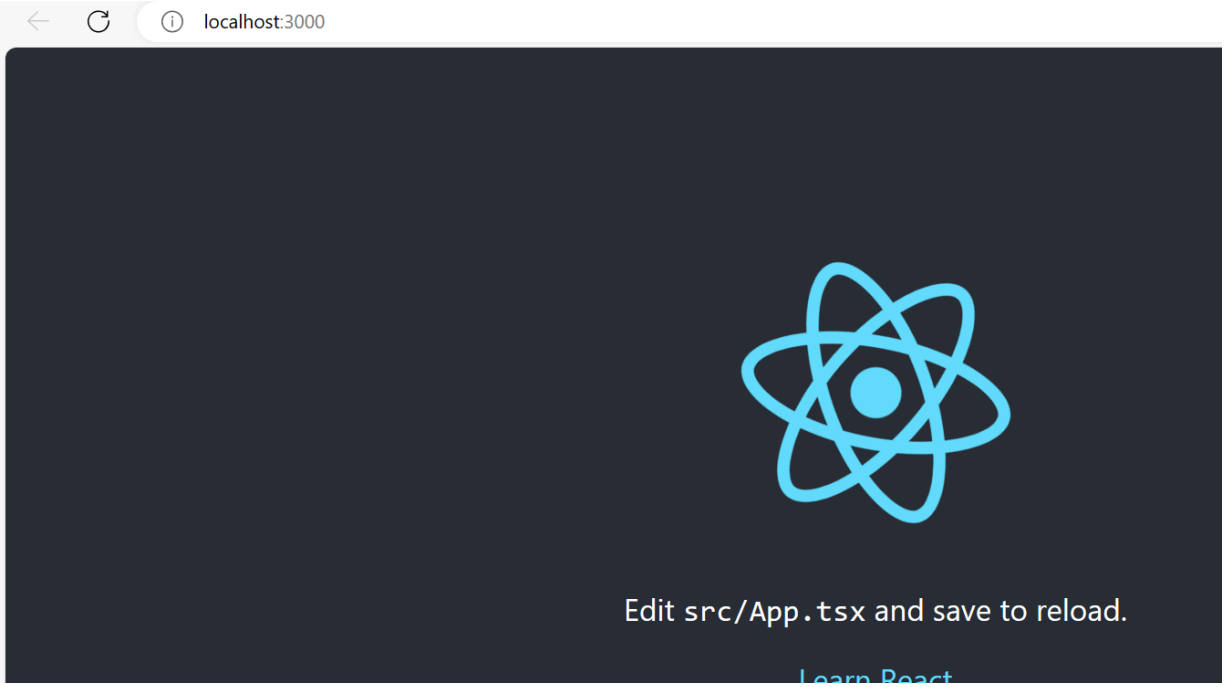


Figura 2: Puesta en marcha del programa de prueba

Una vez verificado esto, vamos a eliminar en nuestro espacio de trabajo todo lo que no vayamos a utilizar, quedando nuestro proyecto de la siguiente forma (ver Figura 3).

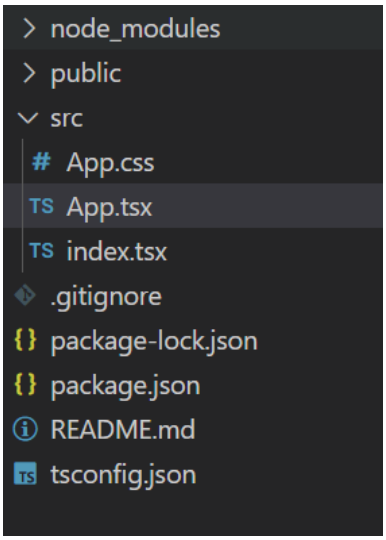


Figura 3: Ficheros y directorios actuales del directorio de trabajo

Ahora, vamos a arreglar los fallos y errores que ha dejado realizar estos cambios. En el fichero *index.tsx* eliminamos los *imports* que no nos interesan y las llamadas a funciones de estos, quedando de la siguiente forma.

```
1
2  import React from 'react';
3  import ReactDOM from 'react-dom/client';
4  import App from './App';
5
6  const root = ReactDOM.createRoot(
7          document.getElementById('root') as HTMLElement
8  );
9  root.render(
10         <React.StrictMode>
11                 <App />
12         </React.StrictMode>
13 );
```

Luego, en el fichero *App.tsx* hacemos lo mismo, y eliminamos casi todo el contenido del *div* e introducimos la palabra "Pokemon", quedando de la siguiente forma.

```
1
2  import React from 'react';
3  import './App.css';
4
5  function App() {
6          return (
7                  <div className="app">
8                          <h1>Pokemon</h1>
9                  </div>
10         );
11 }
12
13 export default App;
```

Y por último, en el fichero *App.css* eliminamos todo su contenido. Finalmente, la vista de previa de la aplicación quedaría de la siguiente forma (ver Figura 4).

Figura 4: Vista de previa actual de la aplicación

Con esto quedaría lista la plantilla inicial de nuestra aplicación, a partir de ahora solo tendremos que ir añadiendo más información para ir creando poco a poco nuestra Pokédex.

2.2. Añadiendo información e imágenes

Ahora vamos a empezar a ir añadiendo datos a la aplicación, para ello contamos con un fichero comprimido llamado *assets.zip*, en donde contiene una carpeta con el mismo nombre, esa carpeta la copiamos dentro de nuestro directorio del proyecto, concretamente dentro de *src*. Esta carpeta contiene diferentes imágenes de la Pokédex, como los iconos, diseños y algunos detalles.

También, vamos a crear una carpeta dentro de *src* para más adelante que llamaremos *api*, ya que en este proyecto utilizaremos dos APIs distintas, ya que cada una nos ofrece datos distintos, en una la utilizaremos para tener todos los Pokémon, y la otra para conseguir la información detallada de cada Pokémon. Por ello, vamos a crear dentro de este directorio dos ficheros, uno se va a llamar *fetchPokemon.js* (por el momento es un fichero JavaScript, pero más adelante lo pasaremos a TypeScript) y el otro *fetchPokemons.js*. Entonces, vamos a copiar por el momento las direcciones de las APIs en estos ficheros (provisionales), en *fetchPokemon.js* introducimos la dirección de PokeAPI.

```
// https://pokeapi.co/api/v2/pokemon/bulbasaur
```

Y luego, en *fetchPokemons.js*, metemos la API para que nos cargue todos los Pokémon.

```
// https://unpkg.com/pokemon@1.1.0/pokemons.json
```

Ya con todo esto, tendríamos las imágenes que usaremos en el proyecto, y la información de las APIs que vamos a usar. A continuación, también vamos a añadir algo de variables para el estilo de la aplicación. Nos vamos al fichero *App.css* y hacemos, en primer lugar, un reseteo básico.

```
*,
*:before,
*:after {
        box-sizing: border-box;
        margin: 0;
        padding: 0;
}
```

Y ahora añadimos las siguientes variables, con el fin de que quede una apariencia del estilo de Pokémon.

```
:root {
        --bg-color: #fafafa;
        --bg-gradient: linear-gradient(
                105deg,
                rgba(184, 203, 237, 1) 0%,
                rgba(222, 245, 221, 1) 100%
        );
        --bg-border: linear-gradient(
                90deg,
                rgba(124, 166, 252, 1) 0%,
                rgba(119, 235, 129, 1) 100%
        );
}
```

Y luego, para el *body*, vamos a utilizar, por ejemplo, la fuente de Apple.

```
body {
        font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Oxygen,
        Ubuntu, Cantarell, "Open Sans", "Helvetica Neue", sans-serif;
}
```

Y ya con esto, vamos a ver la vista previa que tenemos ahora en la aplicación (ver Figura 5).

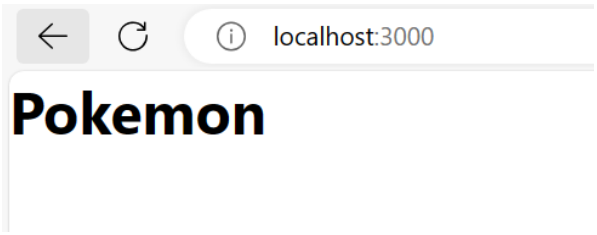


Figura 5: Vista de previa actual de la aplicación con las cambios en el estilo

2.3. Añadiendo sistemas de rutas con React Router Dom

A continuación, vamos a importar todo lo relacionado con las rutas, aquí es donde entra en juego React Router Dom. Lo primero que vamos a hacer va a ser instalar la librería. Para ello, desde la terminal escribimos lo siguiente.

```
$ npm i react-router-dom
```

Ahora, vamos a irnos al fichero *App.tsx*, y vamos a añadir un *Router* para hacer wrapper de todo, el *Router* es lo que suele llamar *BrowserRouter*, dentro de lo aplicación añadimos toda nuestra ruta, mediante *Route*. Quedando de la siguiente forma.

```
function App() {
        return (
                <Router>
                        <div className="app">
                                <Routes>
                                        <Route path="/" element={<Pokemons />} />
                                </Routes>
                        </div>
                </Router>
        );
}
```

De esta forma, estamos añadiendo las páginas que queremos renderizar en cada una de las rutas. Provisionalmente, vamos a añadir más rutas (aunque no las vayamos a utilizar en este momento), como la de */items* y */pokemons*, y también vamos a añadir la página del Pokémon en específico. Quedando todo esto de la siguiente forma.

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import './App.css';

function App() {
        return (
                <Router>
                        <div className="app">
                                <Routes>
                                        <Route path="/" element={<Pokemons />} />
                                        <Route path="/items" element={<Items />} />
                                        <Route path="/pokemons" element={<Pokemons />} />
                                        <Route path="/pokemons/:name" element={<Pokemon />} />
                                </Routes>
                        </div>
                </Router>

        );
}
```

Ahora, lo que vamos a hacer va a ser crear nuestras páginas, entonces creamos un directorio dentro *src* llamado *pages*, y en el creamos los ficheros *items.tsx*, *Pokemons.tsx* y *Pokemon.tsx*. En ellos, vamos a introducir el siguiente contenido provisional. Por ejemplo, en *items.tsx* introducimos lo siguiente.

```
import React from 'react';

const Items = () => {
        return (
                <div>
                        Items
                </div>
        );
};

export default Items;
```

Y lo mismo con los otros dos ficheros, pero cambiando Items por lo que corresponda. Adicionalmente, vamos a realizar lo siguiente, tras haber creado y creado el template inicial de las páginas, en *App.tsx* añadimos los siguientes *imports*.

```
import Pokemon from './pages/Pokemon';
import Pokemons from './pages/Pokemons';
import Items from './pages/Items';
```

Pero, en vez de hacer esto, lo podemos simplificar de la siguiente forma, creamos un fichero en *pages* llamado *index.tsx* y añadimos lo siguiente.

```
export { default as Pokemon } from './Pokemon';
export { default as Pokemons } from './Pokemons';
export { default as Items } from './Items';
```

Y ahora, en el fichero *App.tsx*, quitamos los *imports* anteriores y tan solo añadimos lo siguiente.

```
import { Items, Pokemon, Pokemons } from './pages';
```

Y de esta forma, estaríamos haciendo lo mismo, pero con un código muchas limpio y simplificado.

Ahora, vamos a realizar una prueba, si nos vamos a la página y tratamos de buscar: *localhost:3000/pokemons/diego*, este Pokémon no existe, y no pasaría nada, solo pone Pokémon. Pero, podemos hacerlo de una forma que pueda recibir ese nombre. Esto se hace mediante *useParams*, por ello, nos vamos al fichero *Pokemon.tsx* de *pages*, y escribimos lo siguiente.

```
import { useParams } from "react-router-dom";

const Pokemon = () => {
        const { name } = useParams();
        return (
                <div>
                        {name}
                </div>
        );
};
```

De esta forma, la aplicación se adaptará a los factores de búsqueda que indiquemos. Por tanto, si volvemos a realizar la búsqueda de *localhost:3000/pokemons/diego*, ahora en vez de salirnos Pokémon nos sale el nombre del Pokémon diego (ver Figura 6).

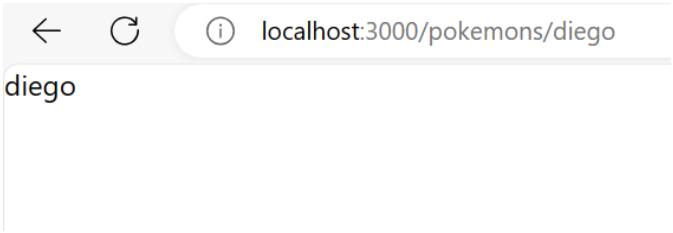


Figura 6: Vista de previa actual de la aplicación con las nuevas rutas

Lo importante de todo esto, es que ya tenemos establecidas las rutas vamos a utilizar en la aplicación, pero con el template más básico, más adelante vamos a ir mejorándolo.

2.4. Creando el Header estático

Ahora vamos a realizar los *Headers*, por ello, nos vamos a ir al fichero *Pokemons.tsx*, y vamos a ir añadiendo el cuerpo de la aplicación (header, main, footer).

```
const Pokemons = () => {
        return (
                <>
                <header>

                </header>

                <main>
                        <h1>POKEMONS</h1>
                </main>

                <footer>FOOTER</footer>
                </>
        );
};

```

Y también, vamos a añadir los componentes, para eso vamos crear una nueva carpeta dentro de *src* llamada *components*, y vamos a llamar al componente *Header.tsx*, creamos una plantilla inicial como habíamos hecho anteriormente.

```
const Header = () => {
        return (
                <header>
                        HEADER
                </header>
        );
};

export default Header;
```

Y ahora en *Pokemons.tsx*, vamos a importar este componente y cambiamos la sección de *header* por lo siguiente.

```
import Header from '../components/Header';

const Pokemons = () => {
        return (
                <>
                <Header />

                (....)
        );
};
```

De esta forma, ya tendríamos la estructura principal de nuestro programa. Ahora, nuestro *Header*, tendrá por supuesto un *header* y un *input* que será lo que nos permitirá hacer la búsqueda, será un *input* de tipo *text*. Para ello, en el componente de *Header* cambiamos a lo siguiente.

```
const Header = () => {
        return (
                <header>
                        <input placeholder="Search a Pokemon" type="text" />
                </header>
        );
};
```

Esto será un *input* provisional en donde podremos buscar los nombres de los Pokémon. Ahora, vamos a intentar darle estilo, para ello, vamos a utilizar una mezcla de estilos globales y módulos de CSS. Los módulos de CSS son un tipo de convención que se usa cuando queremos cargar un estilo, si el componente que lo tiene, lo solicita. Entonces, creamos dentro de la carpeta *components* el fichero *header.module.css*, y ahora en el fichero del componente Header, añadimos el siguiente *import*, y añadimos un *className* llamando a los estilos del módulo.

```
import styles from './header.module.css';


const Header = () => {
        return (
                <header className={styles.header}>
                        <input placeholder="Search a Pokemon" type="text" />
                </header>
        );
};
```

De esta forma, todo lo que pongamos en el *header* del módulo de CSS, lo debería de leer. Sin embargo, en el *import* veremos que no está fallando debido a que no lo está encontrado. Por ello, hay que crear un nuevo fichero dentro de *src* llamado *global.d.ts* (da igual el nombre, podemos poner el que queremos), y dentro de este fichero vamos a añadir el siguiente contenido.

```
declare module "*.module.css"{
        const classes: { [key: string]: string };
        export default classes;
}
```

De esta forma, ya automáticamente el *Header* detectará el módulo de CSS. Ahora, vamos a ponernos a configurar nuestro estilo, para ello, vamos al módulo de CSS y vamos a añadir el siguiente código.

```
.header {
        display: grid;
        font-size: 1.2rem;
        font-weight: bold;
        padding: 10px;
        place-items: center;
        background: var(--bg-gradient);
}

.input {
        background-color: rgba(121, 121, 121, 0.267);
        padding: 10px 10px;
        border-radius: 5px;
        margin: 10px 10px;
        border: none;
        width: 80vw;
}
```

Sin embargo, veremos que no se ha aplicado ya que todavía no lo hemos añadido en el componente, por ello, lo añadimos.

```
import styles from './header.module.css';


const Header = () => {
        return (
                <header className={styles.header}>
                        <input className={styles.input} placeholder="Search a Pokemon" type="text" />
                </header>
        );
};
```

Y si ahora entramos en la aplicación, veremos que se han efectuado correctamente todos estos cambios (ver Figura 7).

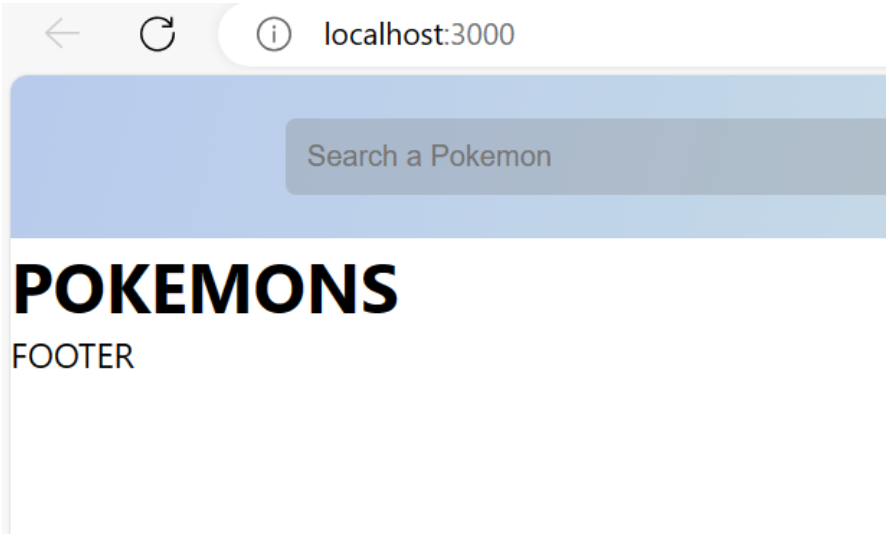


Figura 7: Vista de previa actual de la aplicación con el nuevo componente

Con esto, vamos a dejar preparado nuestro *Header* para más adelante. Por ello, vamos a tratar de realizar las búsquedas mediante el fichero *Pokemons.tsx*, en donde le vamos a ir pasando los datos, porque en función de la búsqueda del *Header*, se va influenciar lo que tengamos dentro del *main*, por eso no podemos hacerlo todo en el componente sino que tendremos que pasarle los datos. Esto lo vamos a realizar de la siguiente forma, dentro de este fichero vamos a crear un *useState*, por ello, escribimos lo siguiente.

```
import Header from '../components/Header';
import { useState } from 'react';

const Pokemons = () => {
        const [query, setQuery] = useState('BULBASAUR');
        return (

                <>
                <Header query={query} setQuery={setQuery} />

                (....)

        );
};
```

Pero veremos que nos está dando error al interpretar los datos de *query*, por ello, nos vamos al fichero *Header.tsx* e introducimos el siguiente contenido.

```

type HeaderProps = {
        query: string;
        setQuery: (query: string) => void;
}


const Header = ({query, setQuery}: HeaderProps) => {
        return (
        <header className={styles.header}>
        <input className={styles.input} placeholder="Search a Pokemon" type="text" />
        </header>
        );
};
```

Como podemos observar, hemos definido dentro del componente *query* y *setQuery*, pero esto de primeras no lo reconoce y nos dará error. Para esto, definimos un *type* de nombre *HeaderProps*, y definimos de qué tipos de datos se tratan estas variables, que será un consulta de cadena de caracteres y el *setQuery* que es una función. Por último, se indica en el componente que los tipos de datos se definen en el *type HeaderProp*. Ahora, con esto ya vemos que lo reconoce sin problema.

Ahora, vamos a añadir en el *Header.tsx* el *value* para las consultas.

```
const Header = ({query, setQuery}: HeaderProps) => {
        return (
                <header className={styles.header}>
                        <input
                                className={styles.input}
                                placeholder="Search a Pokemon"
                                type="text"
                                value={query}
                        />
                </header>
        );
};
```

Ahora, si entramos en la aplicación vemos que nos aparece el nombre BULBASAUR (ver Figura 8) debido a que lo estamos pasando a través del *query* del *header*. Pero si lo quitamos, veremos que no aparece nada.

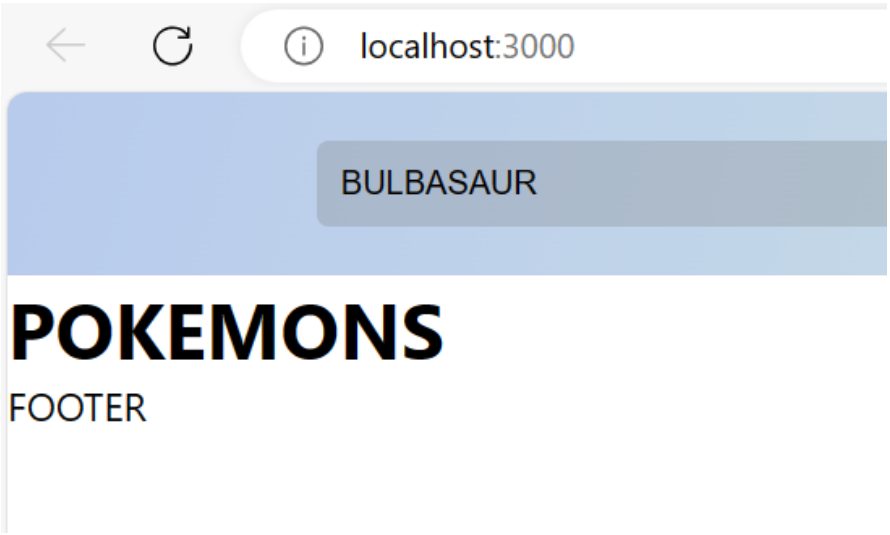


Figura 8: Vista de previa actual de la aplicación al introducir las consultas

Ahora mismo si intentamos escribir algo no pasa nada, debido a que no tenemos ningún *onChange*, de forma que cada vez que escribamos algo, que lo reconozca y ocurra un evento. De forma que lo que haremos será cambiar el estado mediante la función *setQuery*.

```
        const Header = ({query, setQuery}: HeaderProps) => {
                return (
                        <header className={styles.header}>
                                <input
                                        className={styles.input}
                                        placeholder="Search a Pokemon"
                                        type="text"
                                        value={query}
                                        onChange = {(event) => setQuery(event.target.value)}
                                />
                        </header>
                );
        };
```

Ahora, como podemos observar en la Figura 9, ya podemos escribir en el buscador lo que sea, es decir, ya podemos acceder a la variable de *query* no solo desde el *header*, sino que también podemos utilizarla a través del *main*.

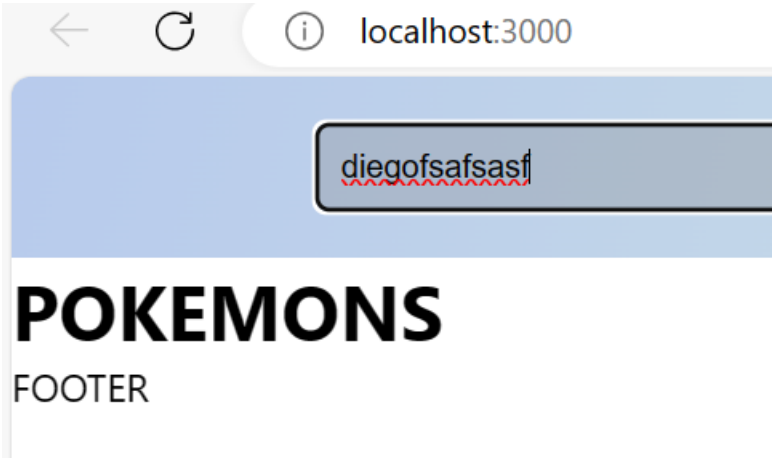


Figura 9: Vista de previa actual de la aplicación al introducir el *onChange* para las consultas

2.5. Creando el Footer

Para el *Footer* vamos a hacerlo de la misma forma que con el *Header*, vamos a crear un componente llamado *Footer.tsx*, y creamos un template básico como habíamos hecho antes, y en el fichero *Pokemons.tsx* añadimos el componente como habíamos hecho antes con el *Header*.

```
import Header from '../components/Header';
import Footer from '../components/Footer';
import { useState } from 'react';

const Pokemons = () => {
        const [query, setQuery] = useState('');
        return (
                <>
                <Header query={query} setQuery={setQuery} />

                <main>
                        <h1>POKEMONS</h1>
                </main>

                <Footer />
                </>
        );
};
```

Ahora, vamos a pensar cómo vamos a querer nuestro *footer*. Lo que tendrá será tres partes, es decir, una fila con los tres elementos, entonces, simplemente trataremos de imitar esa estructura. En el componente *Footer.tsx* escribimos lo siguiente.

```
import { Link } from "react-router-dom";

// Assets
import PokemonPic from '../assets/pikachu.png';

const Footer = () => {
        return (
                <footer>
                        <Link to="/pokemons">
                                <img src={PokemonPic} alt="Pokeball"/>
                                Pokemons
                        </Link>

                        <Link to="/pokemons">
                                <img src={PokemonPic} alt="Pokeball"/>
                                Items
                        </Link>

                        <Link to="/pokemons">
                                <img src={PokemonPic} alt="Pokeball"/>
```

```
                                Map
                        </Link>
                </footer>
        );
};
```

Pero veremos que nos estará dando una serie de errores, pero cómo habíamos hecho anteriormente, nos vamos al fichero *global.d.ts* y añadimos la siguiente información.

```
declare module "*.png";
declare module "*.gif";
declare module "*.svg";
```

Ahora, vamos a irnos a la aplicación, y veremos que ya ha podido detectar las imágenes correctamente, pero como podemos observar (ver Figura 10), son las tres imágenes iguales, por lo que vamos a cambiarlas por las adecuadas para cada caso.

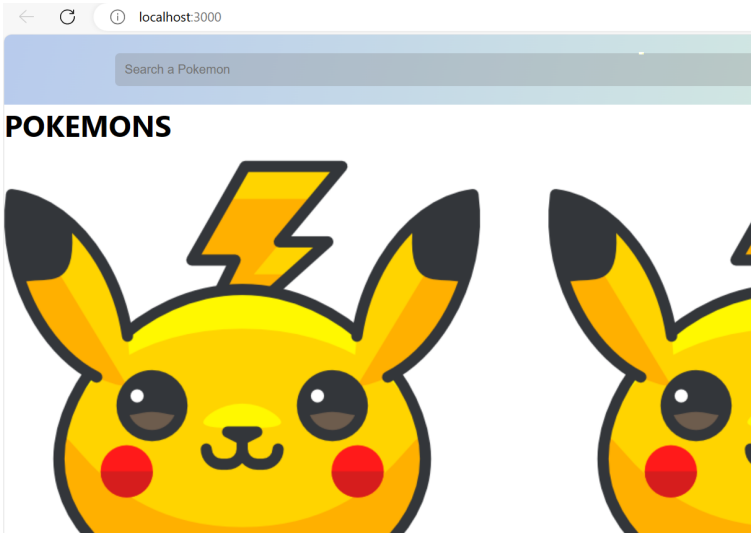


Figura 10: Vista de previa actual de la aplicación al introducir las imágenes de prueba

Por ello, vamos a cambiarlo a lo siguiente.

```
import { Link } from "react-router-dom";

// Assets
import PokemonPic from '../assets/pikachu.png';
import LocationPic from '../assets/pointer.png';
import ItemsPic from '../assets/pokeball.png';




const Footer = () => {
        return (
                <footer>
                        <Link to="/pokemons">
                                <img src={PokemonPic} alt="Pokeball"/>
                                Pokemons
                        </Link>

                        <Link to="/pokemons">
                                <img src={ItemsPic} alt="Pokeball"/>
                                Items
                        </Link>

                        <Link to="/pokemons">
                                <img src={LocationPic} alt="Pokeball"/>
                                Map
                        </Link>
                </footer>
```

```
        );
};
```

Quedando ahora de la siguiente forma (ver Figura 11).

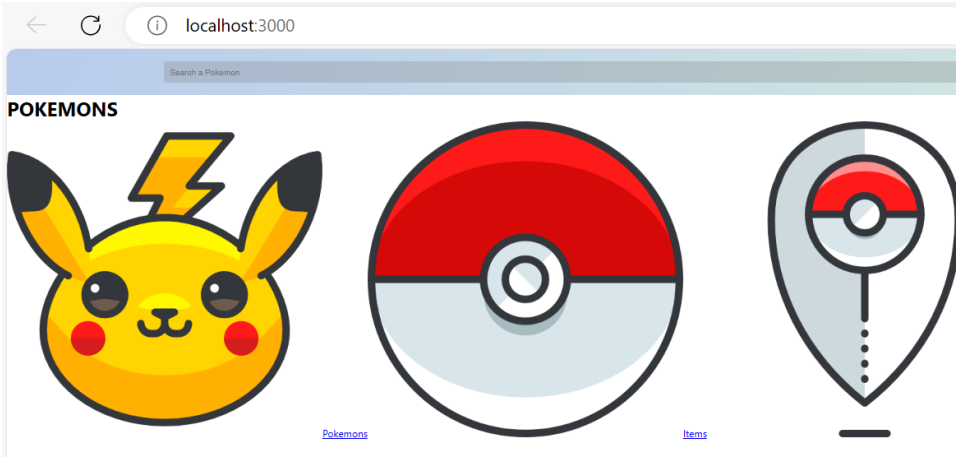


Figura 11: Vista de previa actual de la aplicación al introducir las imágenes buenas

Y ahora, como habíamos hecho anteriormente con el *header*, vamos a añadirle el estilo con los módulos de CSS. Por ello, creamos el fichero *footer.module.css*, y escribimos un estilo que nos guste, por ejemplo, el siguiente.

```
.footer {
        padding: 10px;
        background: var(--bg-gradient);
        display: flex;
        justify-content: space-around;
        width: 100%;
}

.footerLink {
        display: flex;
        flex-direction: column;
        align-items: center;
        text-decoration: none;
        text-align: center;
        color: #292929;
        margin: 10px 0;
        font-size: 12px;
}

.footerIcon {
        width: 35px;
        margin: 5px 0;
}
```

Y ahora, en el fichero *Footer.tsx* vamos a añadir estos estilos.

```
import styles from './footer.module.css';


const Footer = () => {
        return (
                <footer className={styles.footer}>


                (.....)
```

Ahora, vemos cómo quedan estos estilos en la aplicación (ver Figura 12).

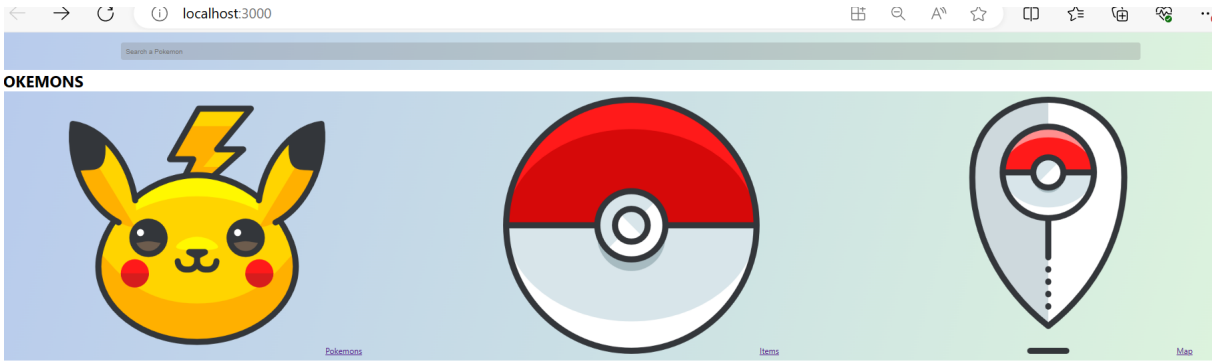


Figura 12: Vista de previa actual de la aplicación al introducir los estilos del *footer*

Como podemos observar, vemos que quedan bastante bien. Ahora, hacemos una ligeras modificaciones en el Footer extra, como añadir el *className* de *Link*, actualizar las rutas correctas e inhabilitar los botones del Pokéball y el Mapa. Y también, en las imágenes, vamos a añadir el *className* de *footerIcon*.

```
const Footer = () => {
        return (
                <footer className={styles.footer}>
                        <Link className={styles.footerLink} to="/pokemons">
                                <img className={styles.footerIcon} src={PokemonPic} alt="Pokeball"/>
                                Pokemons
                        </Link>

                        <Link onClick={(event) => event.preventDefault()} className={styles.footerLink}
    to="/items">
                                <img className={styles.footerIcon} src={ItemsPic} alt="Pokeball"/>
                                Items
                        </Link>

                        <Link onClick={(event) => event.preventDefault()} className={styles.footerLink}
    to="/map">
                                <img className={styles.footerIcon} src={LocationPic} alt="Pokeball"/>
                                Map
                        </Link>
                </footer>
        );
};
```

Quedando de la aplicación de la siguiente forma (ver Figura 13).

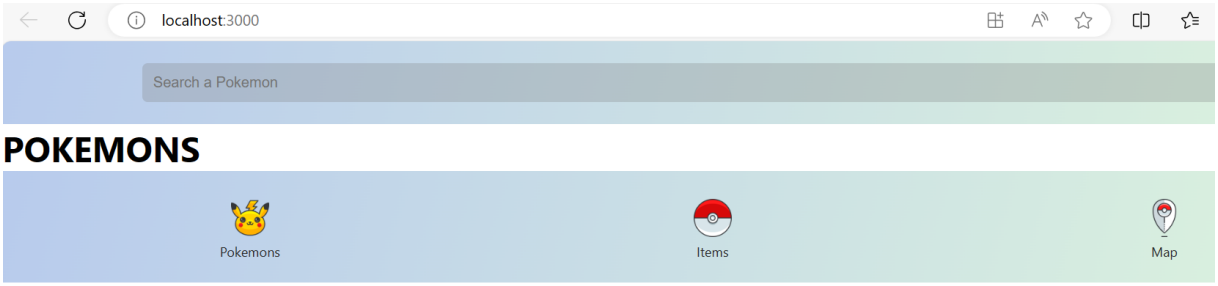


Figura 13: Vista de previa actual de la aplicación al introducir los ajustes de los estilos

Ahora, como podemos observar, se ha ajustado el tamaño de las imágenes (ya que en las anteriores imágenes estaba ajustado el zoom), haciéndolas ver como iconos en la aplicación.

2.6. Empezando con el main y el diseño de los Pokémon

Vamos a empezar con esto, primero entrando en el fichero *App.css* y le vamos a dar un poco de estilo a lo que seria la aplicación. Por ello, vamos a añadir el siguiente contenido.

```
.app {
        display: grid;
        grid-template-rows: auto 1fr;
        min-height: 100vh;
}
```

Y en el fichero *footer.module.css* añadimos el siguiente contenido.

```
.footer {
        (....)
        position: fixed;
        bottom: 0;
}
```

Con esto, volvemos a la aplicación y nos encontramos con lo siguiente (ver Figura 14).

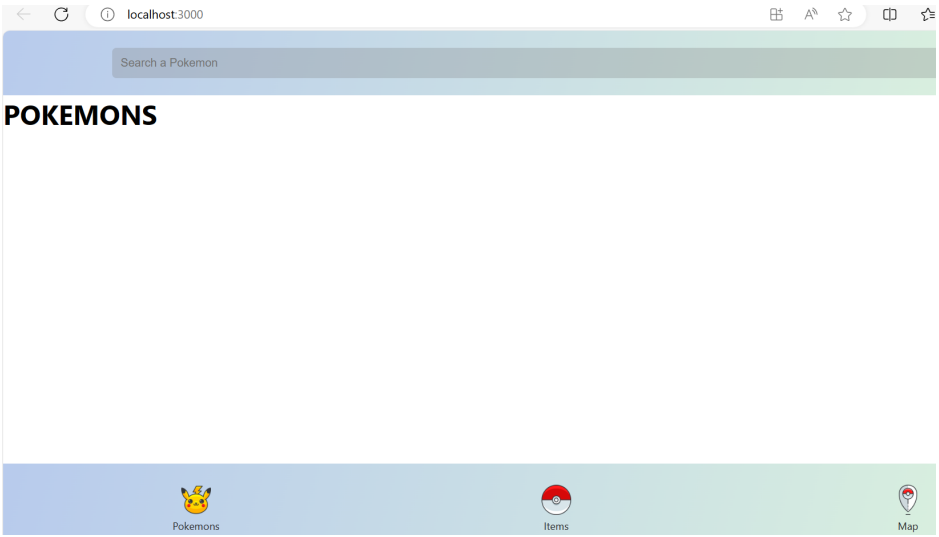


Figura 14: Vista de previa actual de la aplicación al introducir los nuevos cambios en los estilos de la aplicación y el *footer*

Como podemos observar, el *footer* ahora se encuentra en la parte inferior, quedando de esta forma más natural y profesional.

Y también, vamos a añadir en *App.css* el siguiente contenido para el *main*.

```
.app main {
        padding: 10px;
        height: 100%;
        overflow: auto;
}
```

Ahora, vamos a añadir algunos elementos del *main*. Por ello, vamos al fichero *Pokemons.tsx* y vamos a añadir el siguiente contenido.

```
import BulbasaurPic from '../assets/bulbasaur.gif';

const Pokemons = () => {
        const [query, setQuery] = useState('');
        return (

                (....)

```

```
                <main>
                        <nav>
                                <Link to="/"></Link>
                                <img src={BulbasaurPic} alt="bulbasaur"/>
                                <div>
                                        <span>Bulbasaur</span>
                                        <span>001</span>
                                </div>
                        </nav>
                </main>

                (.....)
        );
};
```

Como podemos observar, hemos añadido una navegación y dentro de ésta, el link de cada uno de los Pokémon, en primer lugar, de momento este link no irá a ningún sitio. En primer lugar, vamos a hacer la prueba con solo Bulbasaur, por ello, importamos la imagen, y hacemos referencia desde el *main* a la imagen y rellenamos con dos *span* el nombre del Pokémon y su número en la Pokédex. Con estos cambios, abrimos la aplicación y vemos cómo se han efectuado correctamente estos cambios (ver Figura 15).

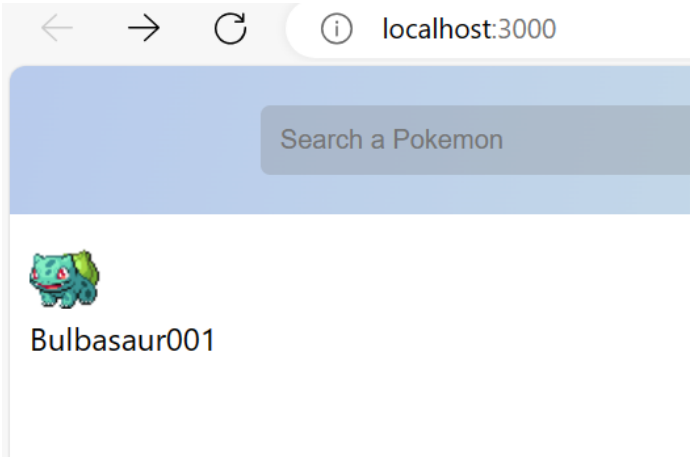


Figura 15: Vista de previa actual de la aplicación al introducir los nuevos cambios en el *main* del Bulbasaur

Ahora, vamos a tratar de añadir estilos a los Pokémon, para ello, creamos un fichero dentro del directorio de *pages* llamado *pokemons.module.css*. Y en él, añadimos el siguiente contenido.

```
.listItem {
        display: flex;
        border-bottom: 3px solid #e4e4e4;
        padding: 10px;
        text-decoration: none;
        align-items: center;
}

.listItem > :first-child {
        margin-right: 15px;
}

.listItem > :last-child {
        margin: 0;
}

.listItemIcon {
        width: 2rem;
        height: 2rem;
        object-fit: contain;
}

.listItemText {
        flex-grow: 2;
}

```

```css
.listItemText > :first-child {
        background: -webkit-linear-gradient(
        rgba(124, 166, 252, 1) 25%,
        rgba(119, 235, 129, 1) 100%
        );
        -webkit-background-clip: text;
        -webkit-text-fill-color: transparent;
        font-size: 1rem;
        display: block;
        margin-bottom: 11px;
        font-weight: bold;
}

.listItemText > :last-child {
        color: grey;
        font-weight: bolder;
        font-size: 0.8rem;
        display: block;
}
```

Y con estos nuevos cambios, volvemos al fichero *Pokemons.tsx* y añadimos la *className* y el *import*.

```tsx
import styles from './pokemons.module.css';

const Pokemons = () => {
        const [query, setQuery] = useState('');
        return (

        (....)

                <main>
                        <nav>
                                <Link className={styles.listItem} to="/">
                                        <img className={styles.listItemIcon} src={BulbasaurPic} alt="
    bulbasaur"/>
                                        <div className={styles.listItemText} >
                                                <span>Bulbasaur</span>
                                                <span>001</span>
                                        </div>
                                </Link>
                        </nav>
                </main>

        (.....)
        );
};
```

Y de esta forma, en la aplicación quedaría de la siguiente forma (ver Figura 16).

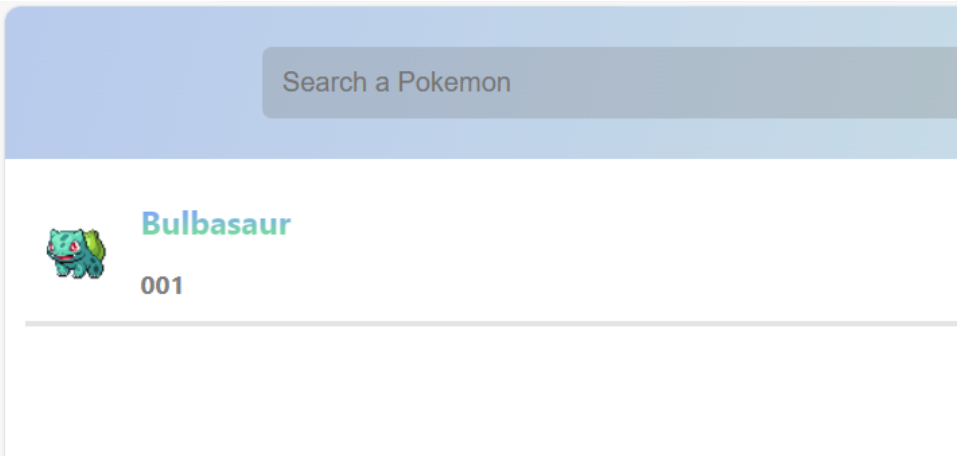


Figura 16: Vista de previa actual de la aplicación al introducir los nuevos cambios en el estilo de los Pokémon

Como podemos ver, poco a poco va cogiendo forma y el estilo que buscamos de la Pokédex.

2.7. Añadiendo el fetch de todos los Pokémon

Ahora lo que haremos será cargar nuestras funciones de *fetch*, exactamente la de todos los Pokémon. Lo primero que haremos será cambiar el nombre al fichero *fetchPokemons.js* por *fetchPokemon.tsx* para que de tipo Typescript. Y ahora, introducimos el siguiente contenido.

```
export async function fetchPokemons() {
        const response = await fetch('https://unpkg.com/pokemon@1.1.0/pokemons.json');

        if (!response.ok) {
                throw new Error("Failed to fetch pokemons");
        }

        const results = await response.json();

        console.log(results);

        const pokemons = results.results.map((pokemon: any) => ({
                name: pokemon.name,
                id: pokemon.national_number,
                imgSrc: `https://img.pokemondb.net/sprites/black-white/anim/normal/${formatPokemonName(
    pokemon.name.toLowerCase())}.gif``,
        }));

        const uniquePokemons = pokemons.filter(
                (pokemon: any, index: number) =>
                        pokemons.findIndex((other: any) => other.id === pokemon.id) === index
        );

        return uniquePokemons;

}
```

Aquí lo que se hace es exportar el *fetch*, crear una respuesta que vayamos a hacer a la dirección de la API, si la respuesta no ha sido buena, nos lanzará un error indicando este hecho. Lo siguiente será crear los resultados, es decir, cuando tenemos esa respuesta, lo pasaremos a un JSON, para comprobarlo ahora vamos a verificarlo en el navegador. Ahora, le decimos que de los datos de un Pokémon en específico, no de todos, por ello, hacemos un mapeo, en donde tendremos en cuenta el nombre del Pokémon, su ID y su imagen. Y por último, le añadimos una funcionalidad, para que en caso de que haya habido algún problema con la API, que los Pokémons sean únicos.

Con esto, vamos a crear una carpeta llamada *utils* dentro de *src*, en esa carpeta creamos el siguiente *utils.tsx*. Esto lo hacemos para crear algunas funciones auxiliares que nos ayudarán, ya que, el problemas de estas APIs es que hay cuatro Pokémons que tienen un problema, que son: Nidoran ♀, Nidoran ♂, Farfetch'd y Mr. Mime, que por cómo lo estamos escribiendo ya podemos intuir a dónde se quiere llegar. Por ello, vamos a intentar solucionar esto, en el fichero *utils.tsx* escribimos lo siguiente. Lo que se pretende es hacer un *replace* de ciertos caracteres que de primeras, la aplicación no sabría interpretar.

```
export function formatPokemonName(name: String): String {
        if (name.includes('')){
                return name.replace('', '-f');
        }

        else if (name.includes('')){
                return name.replace('', '-m');
        }

        else if (name.includes('. ')){
                return name.replace('. ', '-');
        }
        else if (name.includes('farfetch'd)){
                return name.replace('farfetchd', 'farfetchd');
        }
        else return name;

}
```

Nota. No ha sido posible indicar los símbolos de macho y hembra, por ello, mencionar que en los primeros dos if's se verifica si el nombre contiene los símbolos ♀y ♂, pero aparece en blanco estos símbolos.

Entonces, llamamos a esta función desde el *fetch*, y añadimos el *import* correspondiente.

```
import { formatPokemonName } from "../utils/utils";


        (.....)


    const pokemons = results.results.map((pokemon: any) => ({
                name: pokemon.name,
                id: pokemon.national_number,
                imgSrc: `https://img.pokemondb.net/sprites/black-white/anim/normal/${formatPokemonName(
     pokemon.name)}.gif`,
        }));
```

Entonces, una vez tenemos hecho esto, nos vamos al fichero *Pokemons.tsx* y creamos unos states para casa caso de los Pokémon, y a continuación, un *useEffect*, en donde indicamos el *fetch* para todo los Pokémons asíncrono, ya que esto depende de la API y luego lo vamos a querer guardar. Con esto, importamos el *fetchPokemons* para que nos devuelva todos los Pokémons únicos, hacemos el *set* y llamamos a la función *fetchAllPokemons()*.

```
import { fetchPokemons } from '../api/fetchPokemons';




        const [query, setQuery] = useState('');
        const [pokemons, setPokemons] = useState([]);


        useEffect(() => {
                const fetchAllPokemons = async () => {
                        const allPokemons = await fetchPokemons();
                        setPokemons(allPokemons);
                }

                fetchAllPokemons();

        },[])
```

Entonces, con esto, ya podemos eliminar el link que teníamos del Bulbasaur, ya que ya no nos hace falta. En ese mismo sitio, hacemos un mapeo de casi todos los Pokémon, exactamente solo de la primera generación de Pokémon, es decir, los primeros 151 (más adelante añadiremos más).

```

        (.....)

        {pokemons?.slice(0, 151).map((pokemon) => (

                (....)

        )}
```

A partir de ahora, empezará habiendo diversos errores, pero no preocuparse, los iremos solucionando poco a poco. Para este primer error, nos dice que en el fichero *fetchPokemons.tsx* nos ha faltado escribir el tipo de dato. Por ello, lo añadimos de la siguiente forma.

```

        export async function fetchPokemons(): Promise<Pokemon[]> {

                        (......)

        }

```

Y también, vamos a crear una nueva carpeta dentro de *src* llamada *types*, y dentro creamos el fichero *types.d.ts*. En este fichero, añadimos la siguiente información.

```
export type Pokemon = {
        name: string;
        id: string;
        imgSrc: string;
};
```

Con esto, importamos este fichero en *fetchPokemons.tsx* y en *Pokemons.tsx*. Luego, en el fichero *Pokemons.tsx* modificamos el state por lo siguiente.

```
1
2         const [pokemons, setPokemons] = useState<Pokemon[]>([]);
```

Ahora, ya debería de solucionarse todo lo que fallaba. Con esta ejecución, veremos en la aplicación que nos aparecen 151 Bulbasaur's, pero porque en el *main* lo teníamos puesto de esta forma. Por ello, vamos a cambiarlo a lo siguiente para que nos aparezcan los primeros 151 Pokémons, en el *main* vamos a añadir la llave para que no nos advierta el programa de que son únicos, y en la página, vamos a redirigirla a */pokemons* más el nombre del Pokémon. Por lo demás, vamos a intercambiar donde decía Bulbasaur, por los ids, names e imágenes de los Pokémon en general.

```
1
2         (.....)
3
4
5         <main>
6                 <nav>
7                         {pokemons?.slice(0, 151).map((pokemon) => (
8                                 <Link key={pokemon.id} className={styles.listItem} to={`/pokemons/${
      pokemon.name.toLowerCase()}`}>
9                                         <img className={styles.listItemIcon} src={pokemon.imgSrc} alt={
      pokemon.name}/>
10                                        <div className={styles.listItemText} >
11                                                <span>{pokemon.name}</span>
12                                                <span>{pokemon.id}</span>
13                                        </div>
14                                </Link>
15                        ))}
16
17                </nav>
18        </main>
19
20        (.....)
21
```

Y ya con esto, si entramos en la aplicación, ya podremos ver que están todos los Pokémon, con sus nombres, ids e imágenes (ver Figura 17).

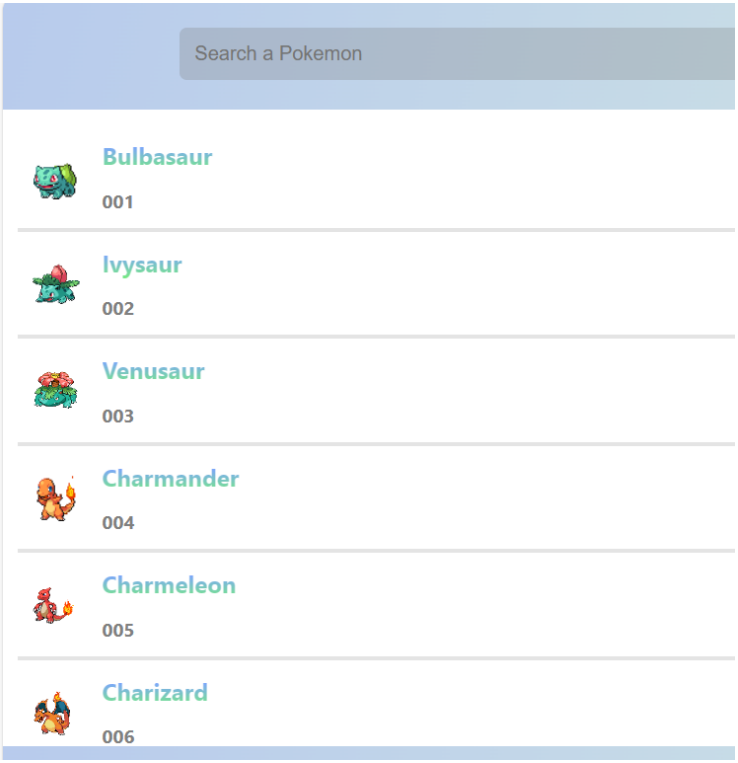


Figura 17: Vista de previa actual de la aplicación al introducir los nuevos cambios

Sin embargo, si bajamos hasta el final del todo, nos vamos a dar cuenta de que no podemos ver el al último Pokémon, en este caso Mew, ya que lo tapa el *footer*, por ello, vamos a ajustar esto. En el fichero de *pokemons.module.css* vamos a añadir la siguiente información.

```
1
2           .nav{
3                   margin-bottom: 100px;
4           }
5
6           (.....)
7
```

Y con esto, nos vamos al *main* y añadimos este estilo a la navegación. Y si volvemos a comprobar la aplicación, ya veremos que se ha corregido este fallo (ver Figura 18).

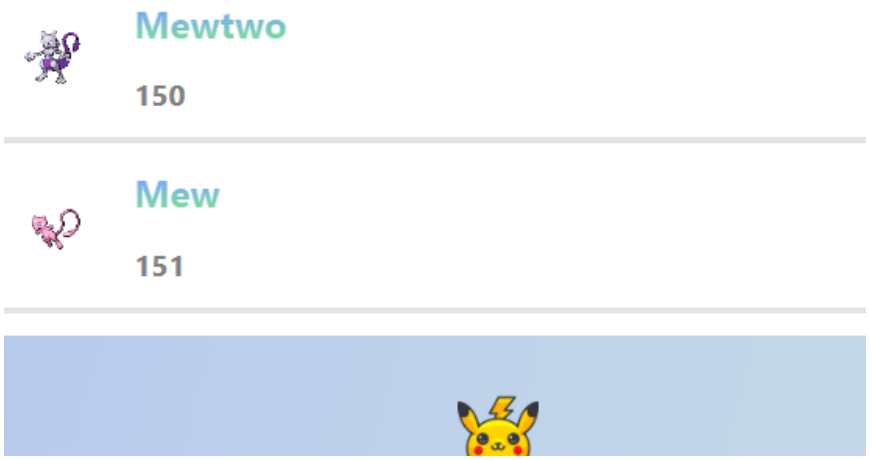


Figura 18: Vista de previa actual de la aplicación al corregir el fallo de navegación

2.8. Creando un Loading Screen

Para cuando estemos tratando con una mayor cantidad de datos, es una buena práctica hacer un loading screen de alguna forma. Para esto, nos vamos al fichero *Pokemons.tsx* y vamos a crear el siguiente estado, en donde al principio será falso. Cuando empeice el fetch lo hacemos verdadero, y cuando éste termine, lo volvemos a hacer falso.

```
1
2         const [isLoading, setIsLoading] = useState(false);
3         const [query, setQuery] = useState('');
4         const [pokemons, setPokemons] = useState<Pokemon[]>([]);
5
6
7         useEffect(() => {
8                 const fetchAllPokemons = async () => {
9                         setIsLoading(true);
10                        const allPokemons = await fetchPokemons();
11                        setPokemons(allPokemons);
12                }
13
14                fetchAllPokemons();
15
16        },[])
```

Entonces, ya con esto, creamos más abajo una condición, en donde si está cargando o no hay todavía Pokémons, devolverá un LoadingScreen.

```
1
2         if (isLoading || !pokemons) {
3
4                 return <LoadingScreen />
5
6         }
7
```

A continuación, creamos esto último, en *components* creamos el fichero *LoadingScreen.tsx*, y generamos en él, una plantilla inicial. A continuación, importamos este componente en *Pokemon.tsx*, y hacemos una prueba inicial (ver Figura 19).

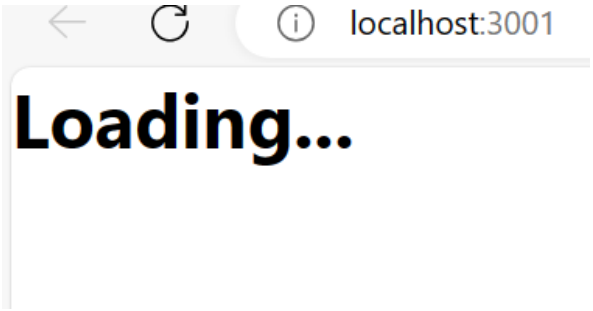


Figura 19: Vista de previa actual de la aplicación al introducir el Loading Screen

Como podemos observar, esto será lo que nos salga en el momento que lanzamos la aplicación, sin embargo, ahora mismo se queda así siempre, por ello, seguimos haciendo cambios. En primer lugar, cambiamos el template de *LoadingScreen.tsx* por lo siguiente.

```
import Pokedex from "../assets/pokedex.png";

const LoadingScreen = () => {
        return (
                <div>
                        <img src={Pokedex} alt="Pokedex" />
                </div>
        );
};

export default LoadingScreen;
```

Y a continuación, creamos el módulo de CSS (*loadingScreen.module.css*), e introducimos lo siguiente.

```
.loadingScreen {
        height: 100vh;
        display: grid;
        place-items: center;
        background: var(--bg-gradient);
}

.loadingScreenIcon {
        width: 30%;
        height: 30%;
        max-width: 200px;
        max-height: 200px;
        object-fit: contain;
}
```

Y en el componente añadimos el modulo de CSS en el div y en la imagen. Y con esto, ya estaría listo todo. Aunque bueno, casi no te deja verlo ya que pasa muy rápido, debido a que no son muchísimos datos los que tiene que buscar. Sin embargo, vamos a crear una función, en *utils.tsx* para que podamos verlo más tiempo. Esta función es la siguiente.

```
export function waitFor(time: number): Promise<void> {
        return new Promise((resolve) => setTimeout(resolve, time));
}
```

Esta función lo que hace es esperar el tiempo indicado para que podamos ejecutar algo con más retardo. Entonces, nos vamos al fichero *Pokemons.tsx*, realizamos el import correspondiente, y añadimos una llamada a esta función.

```
        (....)


        const fetchAllPokemons = async () => {
                setIsLoading(true);
                await waitFor(2000);
                const allPokemons = await fetchPokemons();
                setPokemons(allPokemons);
                setIsLoading(false);
        }
```

```
11
12          (....)
13
14
```

Como podemos observar, le hemos añadido un retardo de 2 segundos y que sea asíncrono. Ahora, lo probamos en la aplicación, y veremos la imagen de la Pokédex antes de cargar la aplicación y verificaremos está en pantalla 2 segundos (ver Figura 20).

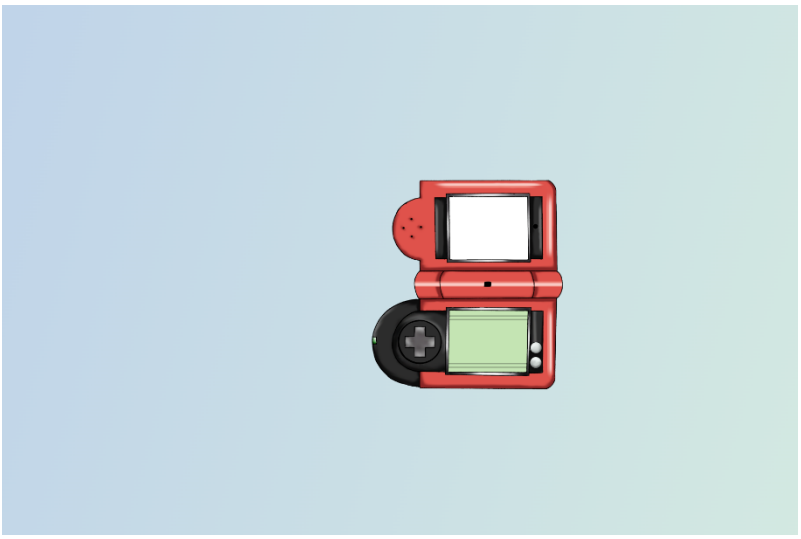


Figura 20: Vista de previa actual de la aplicación al introducir la imagen del Loading Screen y verificación de que tarda 2 segundos en cargar la aplicación

2.9. Arreglando la búsqueda

Ahora vamos a arreglar el buscador que habíamos creado previamente, ya que por ahora, solo nos quedamos con que podemos escribir texto en él, pero no podemos realizar ninguna búsqueda, entonces vamos a solucionar esto. En primer lugar, vamos al fichero *Pokemons.tsx* y creamos la siguiente función.

```
1 const filteredPokemons = pokemons?.slice(0, 151).filter((pokemon) => {
2         return pokemon.name.toLowerCase().match(query.toLowerCase());
3 });
```

Esta función lo que hará filtrar entre los 151 Pokémons y devolverá los Pokémons que cuando se haga *LowerCase* hagan *match* con la consulta también en *LowerCase*. A continuación, intercambiamos en el *main* la llamada a los Pokémon por el nombre de esta función, y veremos que ya funciona (ver Figuras 21 y 22).

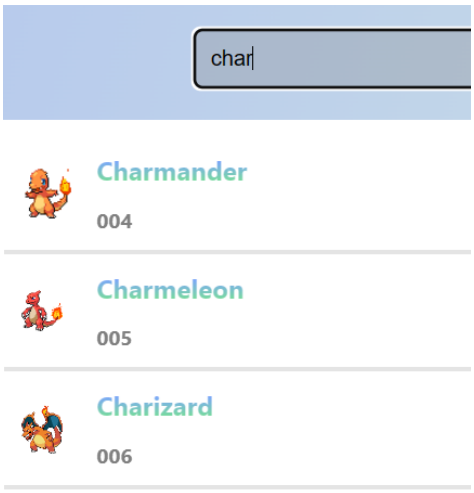


Figura 21: Vista de previa actual de la aplicación al introducir la búsqueda (I)

Figura 22: Vista de previa actual de la aplicación al introducir la búsqueda (II)

2.10. Editando los estilos de la página de los Pokémon

Ahora vamos a cambiar la página de cada de Pokémon y vamos a darle inicialmente estilos. Entonces, vamos a ir creando un fichero dentro de *pages* para el estilo llamado *pokemon.module.css*. Entonces, lo que queremos de esto es, una parte en donde tendremos todo acerca del Pokémon (HP, Attack, Defense), el *footer* y un botón para volver atrás. Entonces, en primer lugar, vamos a tratar de hacer el botón, luego, nos vamos al fichero *Pokemon.tsx*. Y vamos a introducir el siguiente contenido.

```
import { useNavigate, useParams } from "react-router-dom";
import PokeballImg from "../assets/pokeball.png";
import BulbasaurImg from "../assets/bulbasaur.gif";
import Footer from "../components/Footer";

const Pokemon = () => {
        const { name } = useParams();
        const navigate = useNavigate();


        return (
                <>
                        <button onClick={() => navigate(-1)}>
                                <img src={PokeballImg} alt="Pokeball" />Go Back
                        </button>

                        <div>
```

```
                        <main></main>

                        </div>

                        <Footer />

                </>
        );


};
```

Como podemos observar, hemos importado las imágenes correspondientes del botón, y se llama a esa imagen y se dice que ponga el texto "Go Back", también decimos que mantenga el *footer* y también indicamos que tendrá un *div* y un *main*, ahora los completamos. Para el botón, vamos a indicar la función *useNavigate()*, y definimos que el botón, dándole un click va a hacer que se avance para atrás.

Ahora, volvemos al *main*, dentro del *main* creamos otros *div's* en donde especificamos el nombre del Pokémon, su número, la imagen, y más cosas opcionales que podemos ir añadiendo.

```

        (....)

        <div>
                <main>
                        <div>{name?.toUpperCase()}</div>
                        <div>Nr. 001</div>
                        <div>
                                <img src={BulbasaurImg} alt="Bulbasaur" />
                        </div>
                        <div>HP: 00</div>
                        <div>Attack: 20</div>
                        <div>Defense: 40</div>
                </main>
        </div>

        (....)

```

Ahora con esto hecho, vamos a darle el estilo, por ello, vamos al fichero que habíamos creado, *pokemon.module.css* e introducimos, por ejemplo, el siguiente contenido.

```
.pokemon {
        height: 85vh;
}

.pokeballButton {
        border: none;
        cursor: pointer;
        outline: none;
        display: flex;
        align-items: center;
        background: none;
        padding: 50px 50px 0 50px;
        transform: translate(-10px, -10px);
}

.pokeballImg {
        width: 25px;
        height: 25px;
        margin-right: 5px;
}

.pokemonInfo {
        display: flex;
        flex-direction: column;
        align-items: center;
        margin: 100px 0;
        transform: translate(0px, -20px);
}

.pokemonInfoImg {
        width: 100px;
        margin: 0 0 30px;
}

```

```
.pokemonTitle {
        margin: 0 0 30px;
        font-size: 2em;
        font-weight: bold;
        transform: translate(0px, -20px);
}
```

Y luego, hacemos las respectivas llamadas desde el *main*.

```

        (....)

        return (
                <>
                        <button className={styles.pokeballButton} onClick={() => navigate(-1)}>
                                <img className={styles.pokeballImg} src={PokeballImg} alt="Pokeball" />Go Back
                        </button>

                        <div className={styles.pokemon}>
                                <main className={styles.pokemonInfo}>
                                        <div className={styles.pokemonTitle}>{name?.toUpperCase()}</div>
                                        <div>Nr. 001</div>
                                        <div>
                                        <img className={styles.pokemonInfoImg} src={BulbasaurImg} alt="Bulbasaur" />
                                        </div>
                                        <div>HP: 00</div>
                                        <div>Attack: 20</div>
                                        <div>Defense: 40</div>

                                </main>
                        </div>

                <Footer />

        </>

        (....)


```

Y ya con esto, estaría listo. Lo único, que como se ha visto, se ha hecho para que la foto y los datos sean todos iguales (menos el nombre) que es el Bulbasaur (ver Figura 23).

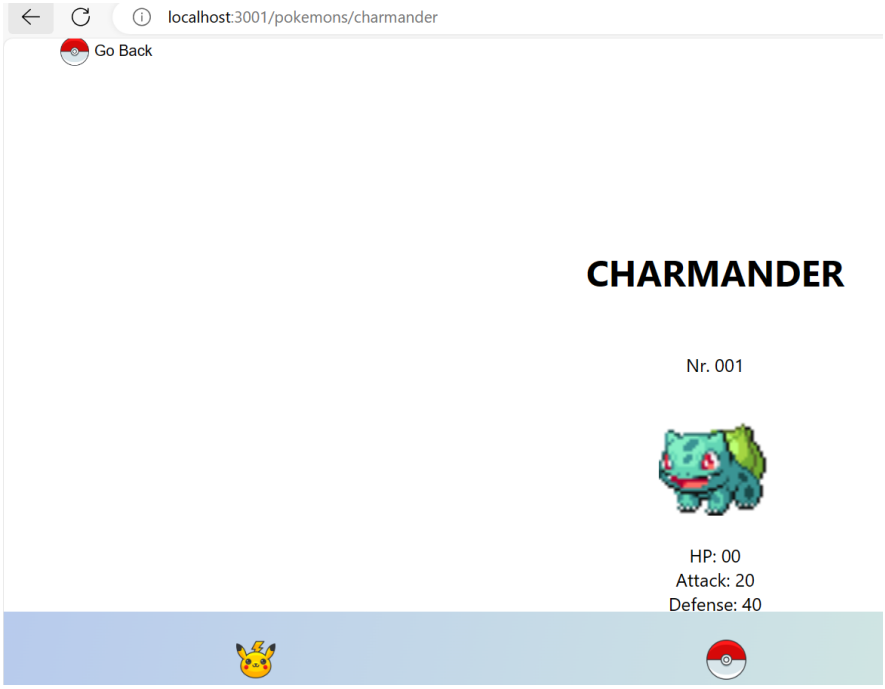


Figura 23: Vista de previa actual de la aplicación introduciendo el nuevo diseño de cada Pokémon

2.11. Fetch a la segunda API

Antes de introducir la segunda API, vamos a crear otro tipo de dato en *types.d.ts* que serán los que hemos introducido previamente.

```
export type PokemonDetails = {
        name: string;
        id: string;
        imgSrc: string;
        hp: number;
        attack: number;
        defense: number;
}
```

Ahora, con esto, hecho, nos vamos a ir a la carpeta *api* y vamos a cambiar el nombre del fichero *fetchPokemon.js* a *fetchPokemon.tsx*, y añadimos el siguiente contenido.

```
import { PokemonDetails } from "../types/types.d";
import { formatPokemonName } from "../utils/utils";


export async function fetchPokemon(name: string): Promise<PokemonDetails> {

        const response = await fetch(`https://pokeapi.co/api/v2/pokemon/${formatPokemonName(name)}`);

        if (!response.ok) {
                throw new Error(`Error fetching ${name}`);
        }

        const result = await response.json();

        const pokemon = {
                name: result.name,
                id: result.id,
                imgSrc: result.sprites.front_default,
                hp: result.stats[0].base_stat,
                attack: result.stats[1].base_stat,
                defense: result.stats[2].base_stat,

        };
```

```
24
25          return pokemon;
26
27  }
```

En donde accedemos a la API designada de PokeAPI, y sacamos la información mencionada de los Pokémon, ahora, vamos al fichero *Pokemon.tsx* y añadimos la siguiente información.

```
1
2           const [isLoading, setIsLoading] = useState(false);
3           const [pokemon, setPokemon] = useState<PokemonDetails>();
4           const { name } = useParams();
5           const navigate = useNavigate();
6
7           useEffect(() => {
8
9                   async function getPokemon(){
10                          setIsLoading(true);
11                          await waitFor(500);
12                          const fetchedPokemon = await fetchPokemon(name as string);
13                          setPokemon(fetchedPokemon);
14                          setIsLoading(false);
15
16                  }
17
18                  getPokemon();
19
20          }, [name]);
21
22          if (isLoading || !pokemon) {
23                  return <LoadingScreen />
24          }
```

Todo esto es básicamente lo que habíamos hecho anteriormente con el fichero *Pokemons.tsx*, en donde también hemos añadido el Loading Screen y en donde hacemos referencia esta vez a la segunda API. También intercambiamos en el cuerpo del *div*, los datos estáticos que habíamos declarado de Bulbasaur, por lo referentes en este caso a los nuevos tipos que habíamos declarado.

Y con esto, ya tendríamos los detalles de los Pokémons terminados, se vería de la siguiente forma (ver Figura 24).

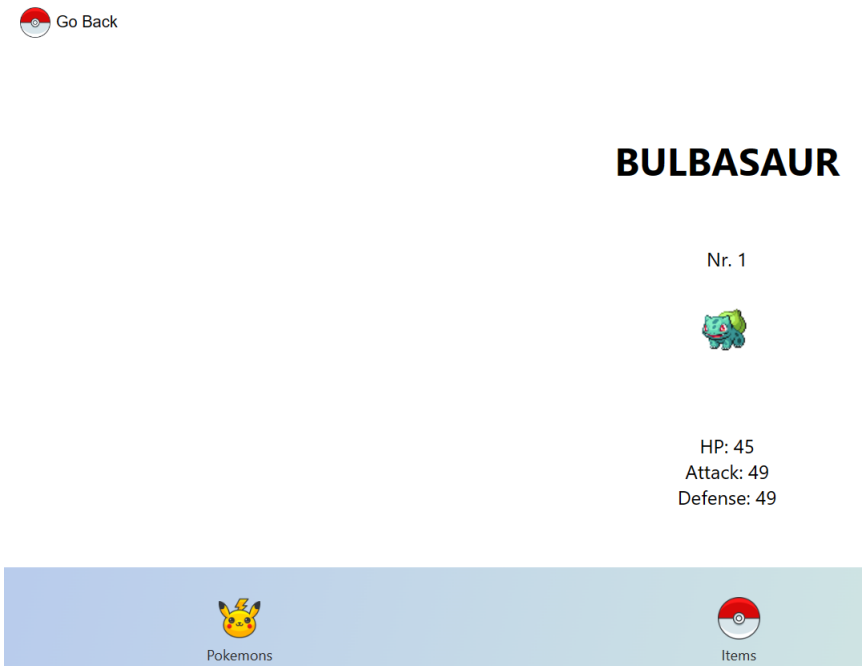


Figura 24: Vista de previa actual de la aplicación introduciendo la API para diseño de cada Pokémon

Y ya con esto, habríamos terminado esta Pokédex básica.

2.12. Deploy de la aplicación

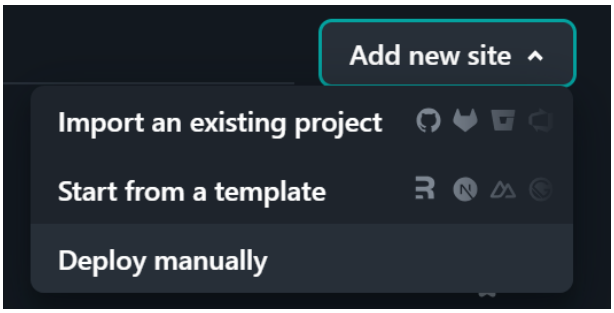
Ahora que ya hemos terminado la fase de programación de la aplicación vamos a tratar de realizar el Deployment de la aplicación. Para ello, nos vamos a una terminal y escribimos lo siguiente para construir el deploy.

```
$ npm run build
```

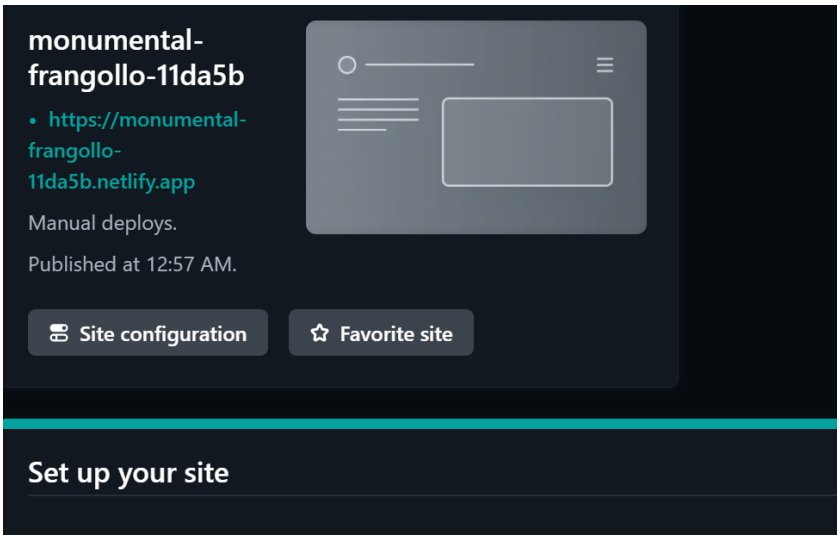
Y lo que hará será generar una carpeta llamada *build*, y es la carpeta que vamos a enviar para el deploy. Ahora vamos a entrar en la siguiente dirección.

https://www.netlify.com/

En esta página nos registramos, y vamos a la pestaña de *Team Overview*.



Y aquí subimos la carpeta *build*. Esperamos a que se procese, y ya tendríamos el proyecto subido públicamente.



Si pinchamos en el enlace, veremos cómo podemos acceder a nuestro app sin problemas.

3. Propuestas de mejora y conclusiones

Como propuestas de mejora se han ocurrido, por ejemplo, realizar más opciones de estilo para que la aplicación y se luzca un poco mejor, aunque ya ha cumplido mis expectativas con esta primera versión. De la misma forma, me gustaría poner otras imágenes que acorden mucho mejor a las Pokédex de algunos juegos.

Además, como ya habíamos mencionado, se pretendía tener una Pokédex en donde aparezcan todos los Pokémon de todas las generaciones, aunque esto no es muy complicado, ya que solo habría que cambiar dos valores de dos funciones, pero me parecía apropiado señalarlo. También, no se si es posible pero por si acaso lo menciono, me gustaría añadir un audio por cada Pokémon para reproducir su correspondiente grito, ya que eso es un factor muy común en las Pokédex de los videojuegos.

Por otra parte, también me gustaría prestar atención al comportamiento de la aplicación en dispositivos móviles. Actualmente, la aplicación se puede visualizar tanto en un navegador desde un PC como en uno de un móvil, pero aún así me gustaría añadir funciones que suelen ser propias para los dispositivos móviles, como deslizar la pantalla hacia la izquierda o derecha para pasar de un Pokémon a otro, como también es común en los videojuegos.

Por último, y como conclusión de este proyecto, he de decir que me ha encantado haber desarrollado esta aplicación. Se que se trata de un proyecto bastante básico y corto, pero al tratarse de mi primera aplicación web y la primera vez que trabajo con estos lenguajes de programación, como mi primera aproximación a todo este mundillo, me ha parecido fascinante y con ganas de aprender mucho más. Es posible que en algún momento vuelva a este proyecto para realizar las mejoras propuestas, cuando llegue ese momento se prepararía otro informe al respecto :) .

A. Posibles problemas con los módulos de CSS

Si en algún momento o directamente no nos funcionen bien los módulos de CSS empleados, es posible que se tenga que realizar la siguiente instalación, desde una terminal.

```
$ npm i -D typescript-plugin-css-modules
```

Y luego, en el fichero *tsconfig.json*, añadir este hecho de la siguiente forma.

```
1
2           (.....)
3
4           "isolatedModules": true,
5           "noEmit": true,
6           "jsx": "react-jsx",
7           "plugins": [{ "name": "typescript-plugin-css-modules" }]
8
9   },
10
11          (.....)
```

Y con esto, debería de funcionar.