Parte 1: Aplicaciones CRUD y Routing con React

UD5: Construcción de aplicaciones web con React

Después de este documento...

- Aplicación CRUD sencilla con datos en cliente
- Aplicación CRUD con varias páginas – usando rutas - y datos en cliente
- Aplicación CRUD con datos en servidor – API – y componentes de terceros

Aplicación CRUD

- Las operaciones **CRUD** hacen referencia a las operaciones básicas que se pueden hacer sobre los datos **Create/Read**, **Update** y **Delete**
  - En cuanto a los datos…
    - Pueden estar en cliente
      - Usando ***localStorage*** para persistencia
    - Pueden estar en servidor
      - Consumirlos usando una ***API***
  - Cada operación suele/puede ir en una página independiente
    - Necesitamos mecanismo de rutas

EJERCICIO PROPUESTO I: Veterinaria (1/4)

APP

HEADER

FORMULARIO

LISTA PACIENTES

PACIENTE

EJERCICIO PROPUESTO I: Veterinaria (2/4)

- Para los estilos usaremos **bootstrap**
  - En index.html
    - Cabecera

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min.css"
integrity="sha256-2TnSHycBDAm2wpZmgdi0z81kykGPJAkiUY+Wf97RbvY="
crossorigin="anonymous">
```

    - Cuerpo

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/bootstrap.min.js"
integrity="sha256-gOQJIa9+K/XdfAuBkg2ONAdw5EnQbokw/s2b8BqsRFg="
crossorigin="anonymous"></script>
```

EJERCICIO PROPUESTO I: Veterinaria (3/4)

- En la App principal tendremos la lista de pacientes
  - Se cargará desde *LocalStorage* al comenzar
  - Se actualizará cada vez que cambie la información
- Los pacientes, así como las funciones modificadoras se le pasarán al formulario y a la lista de pacientes usando lifting
  - En esta primera versión guardaremos el estado del paciente activo
- Usaremos el mismo **formulario** para editar/agregar:
  - Todos los campos son obligatorios
  - En el formulario el botón pondrá **Editar o Agregar** según el caso

EJERCICIO PROPUESTO I: Veterinaria (4/4)

- Estado booleano "error" y si está a true usar el componente Error:

```
const Error = ({children}) => {
    return (
        <div className="bg-danger text-white text-center p-3 text-uppercase
                        font-weight-bold mb-3">
            {children}
        </div>
    )
}

export default Error
```

SPA *(Single Page Application)*

- El contenido se crea a partir de los componentes
  - Se "inyecta" el contenido en el mismo index.html
- Para páginas "distintas", React usa un enrutador

React Router Dom

- Por defecto React no trae un mecanismo de rutas integrado
  - Mantiene las dependencias al mínimo
  - Será necesario instalar un módulo: **React Router**
    - **`npm install react-router-dom`**

- **React Router** permite definir cómo se va a navegar por la app

Usando routing en React

- Importar los módulos correspondientes :

```
import {BrowserRouter, Routes, Route, Link} from "react-router-dom";
```

- Envolver toda la app con el componente **BrowserRouter**
- A partir de donde empiezan las vistas añadir un elemento **Routes**
- Crear un **Route** para cada vista añadiendo el **path** (url) y su correspondiente **element** (el componente a mostrar)

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<Layout />}></Route>
    <Route path="home" element={<Home />}></Route>
    <Route path="pets" element={<Pets />}></Route>
  </Routes>
</BrowserRouter>
```

Página 404

- Sería una “ruta por defecto”
  - * para indicar cualquier opción

```
**<Route path="*" element={<h1>404</h1>}></Route>**
```

Rutas anidadas

- Permiten tener persistencia de los componentes

```
<BrowserRouter>
        <Routes>
          <Route path="/" element={<Layout />}>
            <Route path="home" element={<Home />}></Route>
            <Route path="pets" element={<Pets />}></Route>
          </Route>
        </Routes>
</BrowserRouter>
```

  - Si en Layout hay una navBar, esta persistirá en las vistas Home y Pets, podría contener algo así:

```
<nav style={{ display: "flex", justifyContent: "space-around" }}>
        <Link to="/home">Home</Link>
        <Link to="/pets">Pets</Link>
</nav>
```

useParams

- Hook que nos permite acceder a los parámetros de la ruta
  - En la ruta podemos especificar que vamos a pasar un parámetro:

```
<Route path='/update/:id' element={<UserUpdate/>} />
```

  - Y donde necesitemos acceder a él, usamos el hook (**importado** previamente):

```
const {id}= useParams( )
```

Navegación a través de la interacción del usuario

- Para navegar a una ruta vamos a usar el elemento **Link** (instalado con **React Router**) que NO recarga la app
  - **`<Link to="/create"><Button> Crear usuario</Button></Link>`**
    - No vamos a utilizar las etiquetas **`<a>`** con el atributo **href**
  - Al hacer clic en este elemento, el **BrowserRouter** se encarga de llevarnos al componente correspondiente según lo que le indicamos en el **Route**

Navegación sin que interactúe el usuario

- Si queremos hacer el cambio de ubicación sin interacción del usuario podemos usar:
  - **window.location="/update/"+id**
    - Produce refresco
  - **import {useNavigate} from "react-router-dom"**
    const navigate=useNavigate();
    - En la raíz del componente

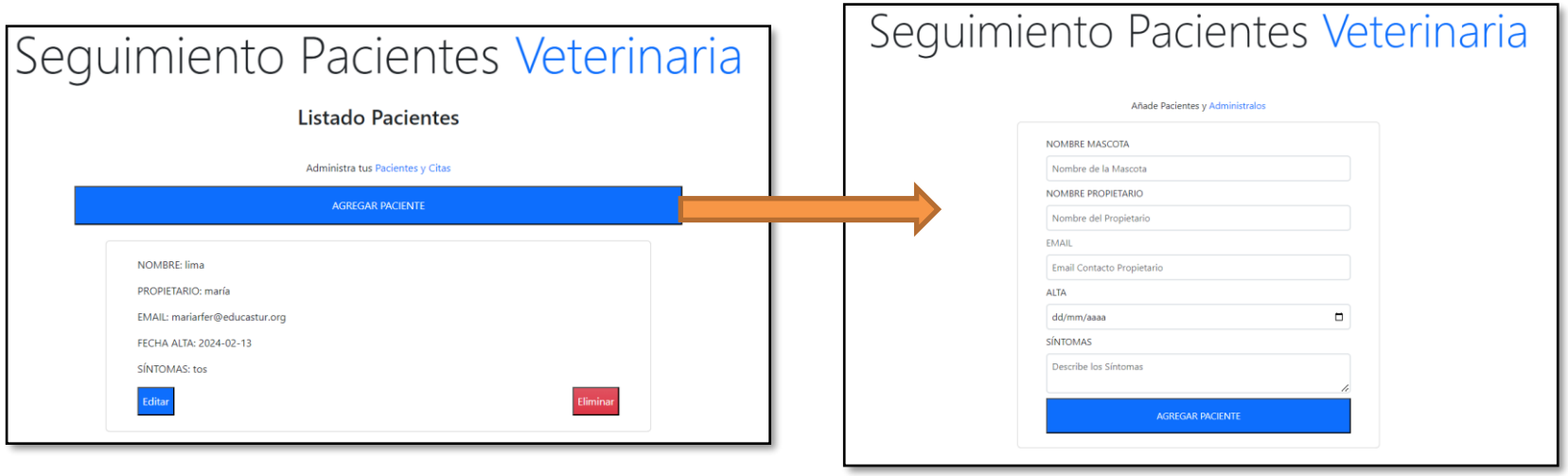
    navigate ("/")

    navigate (-1) - atrás

    …

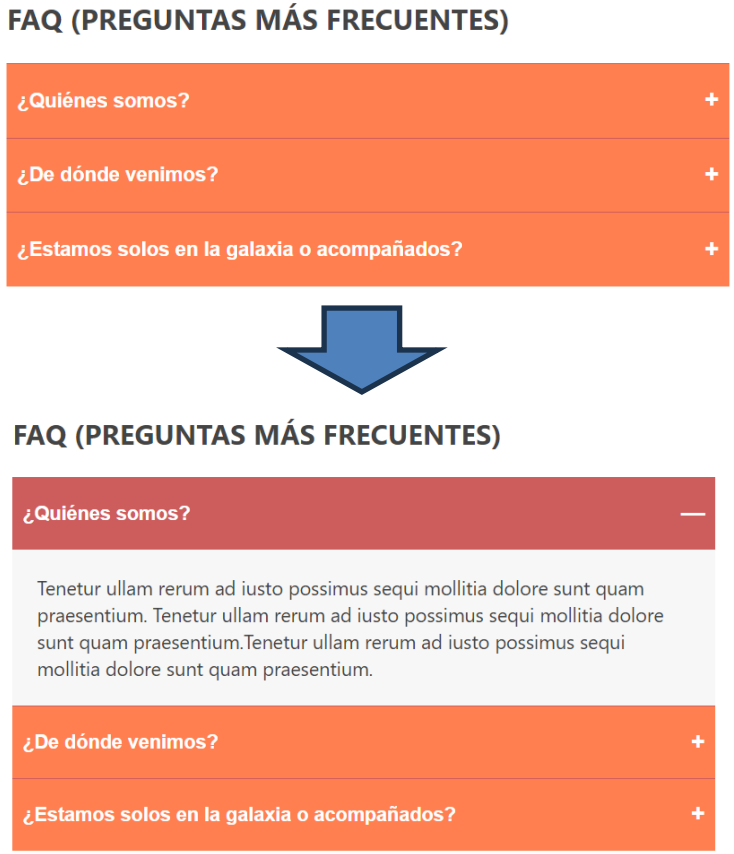
EJERCICIO PROPUESTO II: Pacientes con rutas

- Vamos a hacer una segunda versión del ejercicio, pero separando la parte del formulario de la parte del listado:



EJERCICIO PROPUESTO III (1/3): Acordeón

- Componentes:
  - **App**: Programa principal
    - **Header**: Muestra el título
    - **Accordion**: Contiene todo lo que no es título, compuesto de varios elementos iguales:
      - **AccordionItem**: Cada uno de los elementos desplegables
- Datos en fichero **data.js** *(array de objetos literales)*



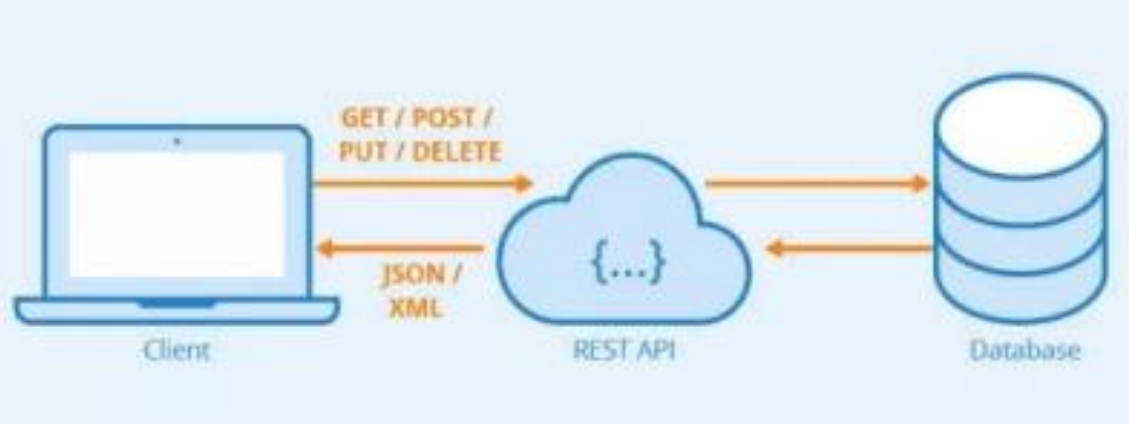
EJERCICIO PROPUESTO III (2/3): Acordeón

- Cada desplegable contiene una pregunta y una respuesta
  - La repuesta se muestra al hacer clic en la pregunta
- Guardar información de elemento activo en un estado
- Aplicar estilos correspondientes

```
▼<div id="root">
  ▼<div class="container">
     <h1 class="heading">3 chistes sobre programación</h1>
    ▼<ul class="accordion">
      ▼<li class="accordion_item active">
        ▼<button class="button">flex
            "¿Por qué C consigue todas las chicas y Java no tiene ninguna?"
            <span class="control">–</span> == $0
          </button>
        ▼<div class="answer_wrapper open">
            <div class="answer">Porque C no las trata como objetos.</div>
          </div>
        </li>
```

Consumo de APIs

- Ejemplos de APIs para ejemplos:
  - Harry Potter API
  - Star Wars API
  - JSON Placeholder
  - Pokeapi
  - ...



Usando fetch API en React

- Normalmente el fetch se incluye dentro de **useEffect**
- Podemos iniciar el estado de los datos como un array vacío que se rellenará cuando lleguen los datos
  - Para mostrar la información en el JSX usaremos map

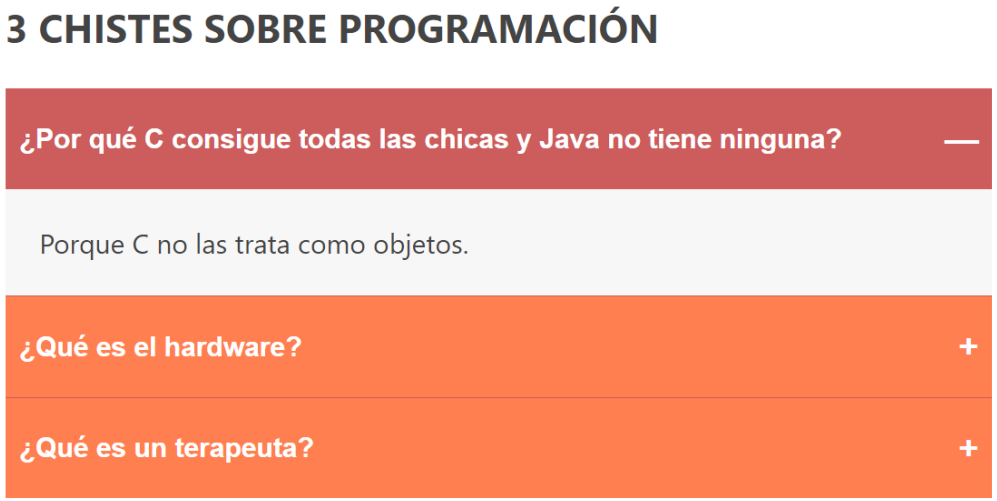
```
const [datos, setDatos] = useState([])
const[loading, setLoading]= useState(false)

useEffect(() => {
    fetch('https://api…..')
      .then(response => {return response.json()})
      .then(jsonObject => {
      const datosObj = jsonObject;
      setDatos(datosObj)
      })
}, [])
```

```
<div>
{ datos.map((dato) =>
                <p key={dato.id}>dato.name</p>)}
</div>
```

EJERCICIO PROPUESTO III (3/3): Acordeón

- Versionaremos el ejercicio del acordeón para que, en lugar de obtener los datos de un js, los saque de una llamada a la API **JokeAPI**:
  - Llamada para tres chistes sobre programación:
    - `https://v2.jokeapi.dev/joke/Programming?lang=es&type=twopart&amount=3`



RETO

- Añade un elemento que permita escoger el tipo de chiste, y adapta la llamada a la API

- Para ello tendrás que consultar la documentación
  - https://sv443.net/jokeapi/v2/