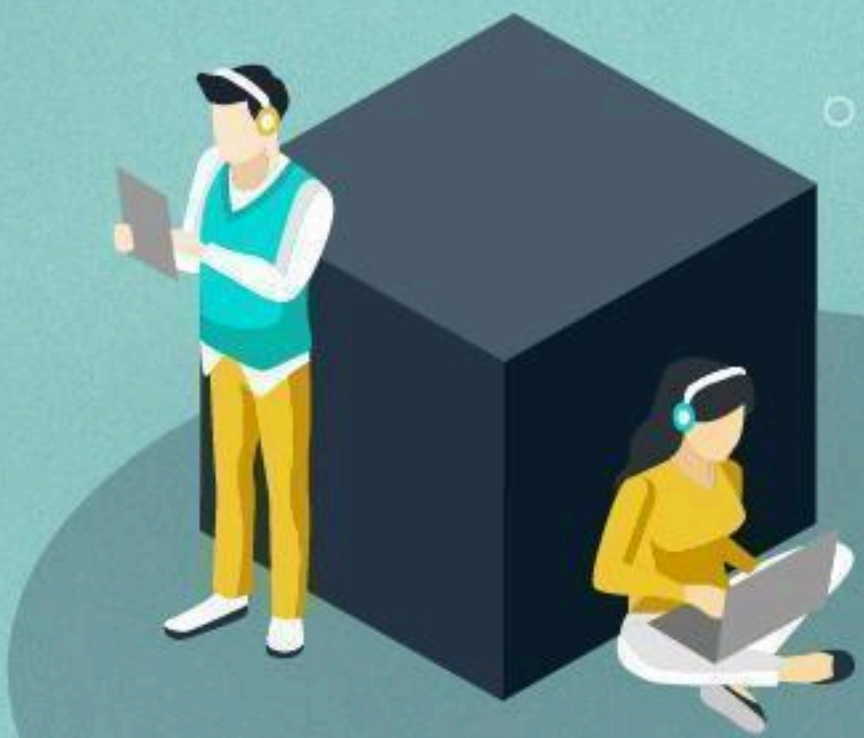
«Talento Tech»

Desarrollo Web 3

Clase 04



Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 4 | APIs y Forms en React

Temario:

- Métodos para Arrays
- Consumo de API renderizando array de objetos
- Formularios y Eventos en React

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Métodos de Arrays

Vamos a aprender algunos métodos básicos para trabajar con arrays de objetos en JavaScript. Los **arrays de objetos** son muy comunes al trabajar con datos provenientes de APIs. Comencemos:

1. Crear un Array de Objetos:

```
// Creamos un array de objetos (por ejemplo, datos de estudiantes)
const estudiantes = [
  { id: 1, nombre: 'Juan', edad: 16 },
  { id: 2, nombre: 'María', edad: 17 },
  { id: 3, nombre: 'Carlos', edad: 16 }
];
```

2. Filtrar Elementos en el Array:

```
// Filtramos estudiantes mayores de 16 años
const estudiantesMayores = estudiantes.filter(estudiante => estudiante.edad > 16);
console.log(estudiantesMayores);
```

3. Encontrar un Elemento en el Array:

```
// Encontramos al estudiante con id igual a 2
const estudianteEncontrado = estudiantes.find(estudiante => estudiante.id === 2);
console.log(estudianteEncontrado);
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

4. Modificar Elementos en el Array:

```
// Modificamos la edad de María a 18
estudiantes.forEach(estudiante => {
  if (estudiante.nombre === 'María') {
    estudiante.edad = 18;
  }
});
console.log(estudiantes);
```

5. Agregar un Nuevo Elemento al Array:

```
// Agregamos un nuevo estudiante
const nuevoEstudiante = { id: 4, nombre: 'Laura', edad: 17 };
estudiantes.push(nuevoEstudiante);
console.log(estudiantes);
```

6. Eliminar un Elemento del Array:

```
// Eliminamos al estudiante con id igual a 1
const estudiantesFiltrados = estudiantes.filter(estudiante => estudiante.id !== 1);
console.log(estudiantesFiltrados);
```

7. Ordenar el Array por una Propiedad:

```
// Ordenamos los estudiantes por edad de menor a mayor
const estudiantesOrdenados = estudiantes.sort((a, b) => a.edad - b.edad);
console.log(estudiantesOrdenados);
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Estos son algunos métodos básicos para trabajar con arrays de objetos en **JavaScript**. Ahora, cuando estemos listos, pasaremos a la parte de cómo consumir APIs que proporcionan datos en formato de array de objetos. ¡Sigamos aprendiendo!

Consumo de API renderizando array de objetos

Para consumir una **API** en **React,** primero debemos hacer una **solicitud HTTP** a la API. Vamos a utilizar el método **fetch**, que es parte de JavaScript, para hacer esta solicitud. **fetch** es una función en JavaScript que se utiliza para realizar solicitudes HTTP (como GET, POST, etc.) desde el navegador hacia un servidor web. Se utiliza comúnmente para interactuar con APIs y recuperar o enviar datos desde y hacia el servidor.

Imaginemos que estamos trabajando con una API ficticia que proporciona información de estudiantes. Vamos a utilizar **JSONPlaceholder**, que es un servicio de prueba para simular solicitudes a una **API REST**.

```
import React, { useState, useEffect } from 'react';

const App = () => {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Endpoint para obtener las publicaciones de JSONPlaceholder
    const apiUrl = 'https://jsonplaceholder.typicode.com/posts';

    // Hacer la solicitud a la API
    fetch(apiUrl)
      .then(response => response.json())
      .then(data => setPosts(data))
      .catch(error => setError(error.message))
      .finally(() => setLoading(false));
  }, []);
```

```
  return (
    <div>
      <h1>Posts</h1>
      {loading ? (
        <p>Loading...</p>
      ) : (
        <ul>
          {error ? (
            <p>Error: {error}</p>
          ) : (
            posts.map(post => (
              <li key={post.id}>
                <h3>{post.title}</h3>
                <p>{post.body}</p>
              </li>
            ))
          )}
        </ul>
      )}
    </div>
  );
};

export default App;
```

Ahora, cuando utilices este componente en tu aplicación React, obtendrás la lista de estudiantes desde la API en lugar de tener datos estáticos. ¡Espero que esto te ayude a empezar a trabajar con APIs en React!

Formularios y Eventos en React

Vamos a explorar cómo trabajar con eventos y formularios en una aplicación React utilizando un ejemplo sencillo de contador.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Creación del Componente del Contador

Primero, crearemos un componente funcional llamado “**Contador**” que tendrá un estado para almacenar el valor del contador y dos botones para incrementar y decrementar ese valor:

```
import React, { useState } from 'react';

const Contador = () => {
  // Estado para almacenar el valor del contador
  const [contador, setContador] = useState(0);

  // Manejar el evento de incremento
  const incrementar = () => {
    setContador(contador + 1);
  };

  // Manejar el evento de decremento
  const decrementar = () => {
    setContador(contador - 1);
  };

  return (
    <div>
      <h2>Contador: {contador}</h2>
      <button onClick={incrementar}>Incrementar</button>
      <button onClick={decrementar}>Decrementar</button>
    </div>
  );
};

export default Contador;
```

En este ejemplo:

Utilizamos el estado (useState) para almacenar el valor del contador.
Definimos dos funciones (incrementar y decrementar) que actualizan el estado del contador.
En los botones, utilizamos el atributo ”onClick” para asignar las funciones a los eventos de clic.

onClick

Este evento se activa cuando haces clic en un elemento. Es útil para manejar acciones como clics en botones o elementos interactivos.

En ocasiones, al momento de controlar un formulario, además de necesitar capturar los datos que el usuario completa, podríamos necesitar que nos indique si el formulario ha sido enviado y renderizando nuevamente a la página.

Notaremos algunas cuestiones:

Hay 2 estados establecidos: uno para almacenar los datos que fueron llenados en el formulario y otro para indicar si el formulario ha sido llenado.

Atención en el manejo de eventos: tanto en el cambio de texto del campo (input) como en el envío del formulario

- **onChange:** mediante la función “manejoCambioTexto” indicamos junto con el spread operator lo que ya se haya llenado del objeto y así se le suman los datos nuevos.

- **onSubmit:** mediante la función “enviarDatos” se indica que prevenga el comportamiento predeterminado y cambie el estado de sí, el form, ha sido completado. No queda claro cuál es el estado que no deberia cambiar (si, el form, ha sido completado?).

En este ejemplo veremos lo siguiente:

```
import React, { useState  } from "react";

const Formulario = () => {

    // Estado para almacenar como objeto los datos del formulario
    const [datos, setDatos] = useState({
        nombre: '',
        apellido: ''
    })
    // Estado para indicar si el formulario se ha llenado
    const [formLleno, setFormLleno] = useState(false)

    // onChange
    const manejoCambioTexto = (event) => {
        // event.target.name = Hace referencia al atributo
        // event.target.value = Hace referencia al valor del atributo
        setDatos({
            ...datos,
            [event.target.name] : event.target.value
        })
    }

    // onSubmit
    const enviarDatos = (event) => {
        event.preventDefault()
        setFormLleno(true)
    }
    // En el condicional con el operador ternario, si el formulario se ha llenado, se mostrará el mensaje
    return (
        <>
            {
                formLleno ?

                <h1>¡Gracias! responderemos a la brevedad</h1>
```

```
        :

        <form className="row" onSubmit={enviarDatos}>
          <div className="col-md-4">
            <input
            type="text"
            placeholder="Nombre"
            className="form-control"
            onChange={manejoCambioTexto}
            name="nombre" />
          </div>
          <div className="col-md-4">
            <input
            type="text"
            placeholder="Apellido"
            className="form-control"
            onChange={manejoCambioTexto}
            name="apellido" />
          </div>
          <div className="col-md-4">
            <button type="submit" className="btn btn-primary">Enviar</button>
          </div>
        </form>
      }
    </>
    )
}
export default Formulario
```

1. Importaciones:

Importamos **useState desde React** para manejar el estado local del componente.

2. Definición del Componente:

Creamos un componente funcional llamado **Formulario.** Desarrollar y enlazar con estados.

Estados:

Utilizamos useState para crear dos estados: datos (almacena los datos del formulario) y formLleno (indica si el formulario se ha llenado).

3. Manejo del Cambio de Texto (onChange):

Definimos la función “manejoCambioTexto” que se llama cuando cambia el contenido de los campos del formulario.
Utilizamos “event.target.name” para obtener el nombre del campo y “event.target.value” para obtener su valor.
Actualizamos el estado datos utilizando el operador de propagación (...datos) y estableciendo el nuevo valor según el campo.

onChange
Se utiliza principalmente en elementos de formulario y se activa cuando cambia el valor de un campo de entrada (input). Es útil para realizar acciones en respuesta a cambios en campos de texto, casillas de verificación, selectores, etc. (Subí este párrafo, check))

4. Envío del Formulario (onSubmit):

Definimos la función “enviarDatos” que se llama cuando se envía el formulario.
Prevenimos el comportamiento por defecto del formulario utilizando “event.preventDefault()”.
Establecemos el estado “formLleno” en true para indicar que el formulario se ha llenado.

5. Renderizado Condicional:

Utilizamos un condicional con el operador ternario (formLleno ? ... : ...) para mostrar diferentes elementos en función del estado formLleno.
Si formLleno es true, se muestra un mensaje de agradecimiento.
Si formLleno es false, se muestra el formulario con campos de nombre y apellido, y un botón de envío.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ahora, puedes utilizar estos componentes en tu aplicación React para experimentar con eventos, formularios y estados. ¡Espero que encuentres útiles estos ejemplos!

Desafío #2

Vamos a continuar con el mismo desafío #1. Ahora lo siguiente es crear los siguientes componentes para que por fin puedas renderizar la vista que tenga las tareas en pantalla 😁 ¿Cómo vamos a hacerlo?

Paso 1: Creación de Componentes y Manejo de Estado con `useState`

Crea un componente `**TaskList.jsx**` en la carpeta `**src/components**` para mostrar la lista de tareas y manejar el estado con `useState`.

```
// src/components/TaskList.jsx
import React, { useState } from 'react';

const TaskList = () => {
  const [tasks, setTasks] = useState([]);

  // Simulación de carga de datos desde una API con useEffect
  React.useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/todos')
      .then((response) => response.json())
      .then((data) => setTasks(data))
      .catch((error) => console.error('Error fetching tasks:', error));
  }, []); // La dependencia vacía asegura que se ejecute solo una vez al montar el componente

  return (
    <div className="container mt-5">
      <h2>Lista de Tareas</h2>
      <ul>
        {tasks.map((task) => (
```

```
        <li key={task.id}>{task.title}</li>
      ))}
    </ul>
  </div>
 );
};

export default TaskList;
```

Paso 2: Integración en la App Principal

Actualiza `**src/App.jsx**` para utilizar el componente `**TaskList**`.

```
// src/App.jsx
import React from 'react';
import TaskList from './components/TaskList';

const App = () => {
 return (
  <div id="app">
   <TaskList />
  </div>
 );
};

export default App;
```

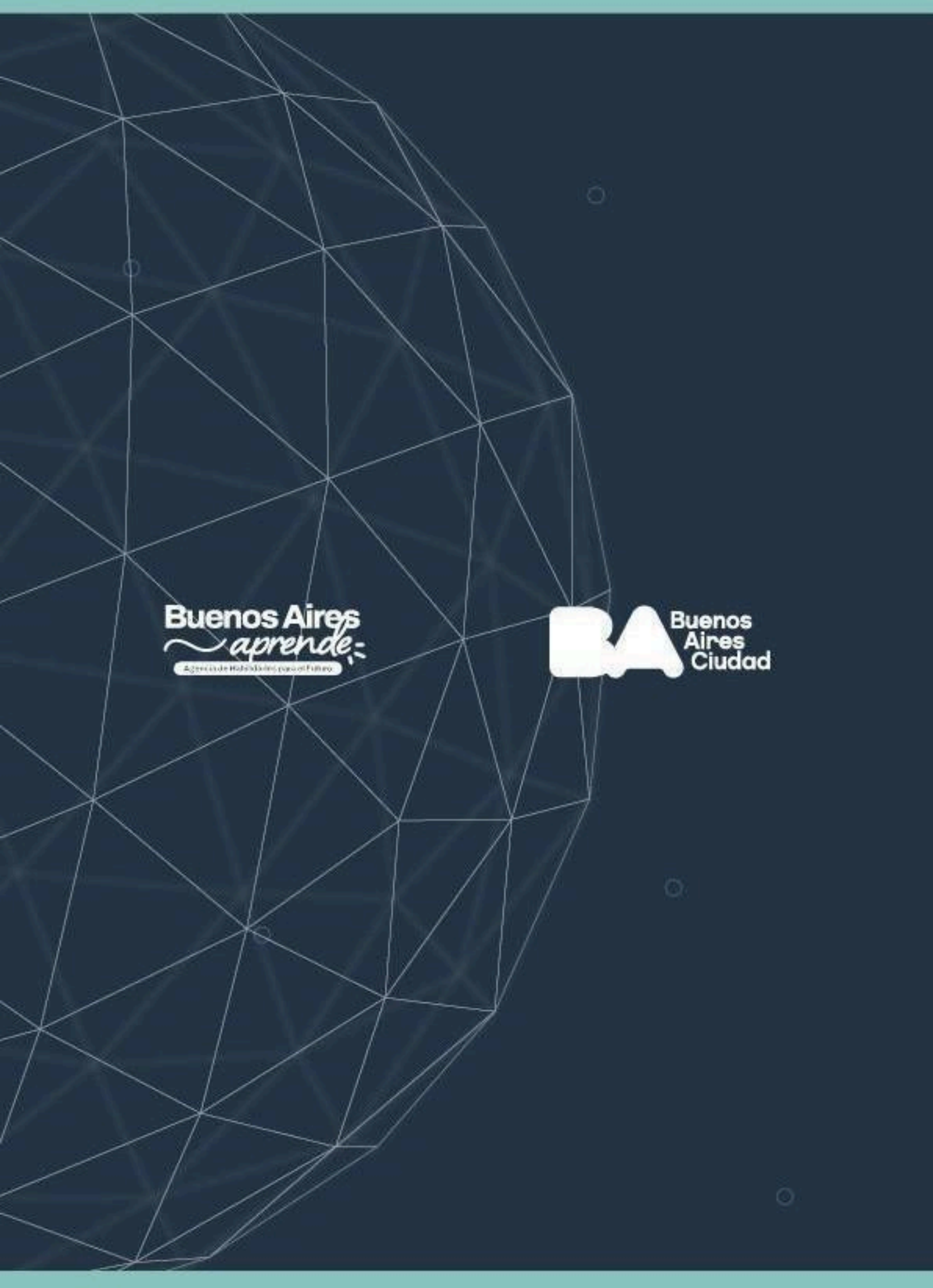
Paso 3: Ejecución de la Aplicación

Ejecuta la aplicación **Vite**.

```
npm run dev
```

En la consola del editor de código deberás ver la url donde estará corriendo la app. Haz **Ctrl + Click** en el enlace y se abrirá el navegador mostrando tu proyecto recién creado. Deberías ver la lista de tareas cargada desde la API y estilizada con Bootstrap y Bootswatch. Este ejemplo integra los conceptos de las clases mencionadas.

La entrega de este desafío será la visualización de tu proyecto en el navegador. Haz una captura de pantalla y podrás subir esa imagen a la plataforma.


Buenos Aires
aprende
Buenos
Aires
Ciudad