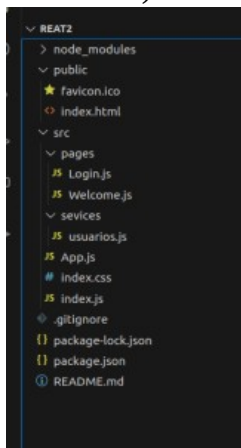


1-Creación del proyecto: abrimos la terminal y tecleamos el siguiente comando `npx create-react-app <nombre del proyecto>`. Una vez creado el proyecto, lo abrimos en Visual Studio Code y eliminaremos aquellos archivos que no utilizaremos, como `logo.svg` y `app.css`, crearemos el directorio `src` y de nuevo en la terminal, instalamos **React Router** con el comando `npm install react-router-dom@6`.

Una vez instalada la librería, para implementación en nuestro proyecto basta con importar e introducir el componente `<BrowserRouter>` en el archivo principal `index.js`; al envolver el renderizado de manipulación del DOM ese componente inicial implementa en el resto de los componentes, los métodos y propiedades que necesitaremos posteriormente. También importamos el archivo raíz `app.js` y las dependencias necesarias de **React y React Router DOM**.

```
rc > JS index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5
6  import { BrowserRouter } from 'react-router-dom';
7  ReactDOM.render(
8    <React.StrictMode>
9      <BrowserRouter>
10
11        <App />
12
13      </BrowserRouter>
14    </React.StrictMode>,
15    document.getElementById('root')
16  );
17
```

2-Creamos un directorio llamado `'pages'` en el directorio `'src'`; dentro de este dos archivos uno llamado `'login.js'` y otro llamado `'welcome.js'` que contienen componentes funcionales que de momento no tendrán la lógica de ejecución de datos, (uno es un formulario y otro con mensaje de bienvenida).



```
const Welcome = () => {
  return (
    <div>
      <div>
        <h2>Bienvenido, <span>{userData.name}</span></h2>
        <p>Edad: <span>{userData.age}</span></p>
        <p>Email: <span>{userData.email}</span></p>
        <Link to="/" />
        <button>Regresar a Inicio</button>
      </div>
    </div>
  );
};

export default Welcome;
```

```
const LoginForm = () => {
  return (
    <div>
      <h2>Iniciar Sesión</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label htmlFor="email">Email:</label>
          <input autoComplete="email" type="email" id="email" name="email" value={x} onChange={handleOnChange} required />
        </div>
        <div>
          <label htmlFor="password">Contraseña:</label>
          <input autoComplete="current-password" type="password" id="password" name="password" value={x} onChange={handleOnChange} required />
        </div>
        <button type="submit">Iniciar Sesión</button>
      </form>
    </div>
  );
};

export default LoginForm;
```

3-A continuación creamos dentro de `'src'` una carpeta llamada `'Services'`, y dentro de ella un archivo `'usuarios.js'` en la cual crearemos las funciones para consumir los datos de `'usuarios'`, estas serán incorporadas en la lógica de los componentes. En este sentido hemos creado una api con nodeJS y Express que correrá localmente, contiene una ruta GET y un array con datos ficticios de `usuarios` que nos servirá para que dichas funciones realicen las llamadas necesarias para la obtención de los datos necesarios para el propósito de este trabajo de evaluación.

```

15 server.js > @ users
1  const express = require('express');
2  const app = express();
3  const PORT = process.env.PORT || 5000;
4  const cors = require('cors');
5  app.use(cors());
6  // Array de usuarios ficticios
7  const users = [
8    {id: 'us1', name: 'Usuario1', email: 'usuario1@example.com', age: 25, password: 'Usuario1123' },
9    {id: 'us2', name: 'Usuario2', email: 'usuario2@example.com', age: 30, password: 'Usuario2123' },
10   {id: 'us3', name: 'Usuario3', email: 'usuario3@example.com', age: 35, password: 'Usuario3123' },
11   {id: 'us4', name: 'Usuario4', email: 'usuario4@example.com', age: 40, password: 'Usuario4123' },
12   {id: 'us5', name: 'Usuario5', email: 'usuario5@example.com', age: 45, password: 'Usuario5123' },
13 ];
14
15 // Ruta GET para obtener la lista de usuarios
16 app.get('/api/users', (req, res) => {
17   console.log(users)
18   res.json(users);
19 });
20
21 // Iniciar el servidor
22 app.listen(PORT, () => {
23   console.log(`Servidor corriendo en el puerto ${PORT}`);
24 });
25

```

Volviendo a la carpeta 'Services' creamos en el archivo 'usuarios.js' la función 'getUsuario' que es responsable de realizar una solicitud a la API para que en la lista de usuarios, busque un usuario específico en función de las credenciales proporcionadas, (password e email) y manejar adecuadamente cualquier error que pueda surgir durante el proceso.

```

export const getUsuario = (email, password) => {
  return fetch('http://localhost:5000/api/users')
    .then(response => {
      if (!response.ok) {
        throw new Error('Error al obtener los usuarios');
      }
      return response.json();
    })
    .then(users => {
      const user = users.find(user => user.email === email && user.password === password);
      if (user) {
        return user.id;
      } else {
        throw new Error('Usuario no encontrado');
      }
    })
    .catch(error => {
      console.error('Error al obtener el usuario:', error.message);
      throw new Error('Error al obtener los usuarios');
    });
};

```

getUsuario acepta dos parámetros: email y password. Utiliza fetch para hacer una solicitud GET a http://localhost:5000/api/users y obtener la lista de usuarios tambien utilizamos then/catch para el manejo de asincronía antes de aplicar los métodos para el manejo de las respuestas.

Una vez que la solicitud es realizada, el código maneja la respuesta de la siguiente manera: Primero, verifica si la respuesta fue exitosa (response.ok). Si la respuesta no es exitosa, se lanza un error con el mensaje "Error al obtener los usuarios".

Si la respuesta es exitosa, se convierte la a formato JSON utilizando response.json(). Una vez que se obtiene la lista de usuarios en formato JSON, se realiza una búsqueda dentro de esa lista para encontrar un usuario que coincida con el email y password proporcionados. Esto se hace mediante el método 'users.find(user => user.email === email && user.password === password)', si se encuentra un usuario que coincide con el correo electrónico y la contraseña proporcionados, la función devuelve el id del usuario encontrado. Si no se encuentra ningún usuario que coincida, se lanza un error con el mensaje "Usuario no encontrado" Finalmente otro bloque catch para capturar cualquier error que pueda ocurrir durante el proceso de solicitud, como errores de red o problemas en la API. Si se produce un error, se imprime un mensaje de error en la consola y se lanza un nuevo error con el mensaje "Error al obtener los usuarios".

La función getUsuarioName se encarga de obtener información completa de un usuario específico a partir de su id. En líneas generales el funcionamiento es igual al de getUsuario con la diferencia de que toma como argumento un 'id', identificador único para cada usuario, si se encuentra un usuario cuyo id coincida, la función devuelve al componente que hace la llamada el objeto con toda la información del usuario especificado.

```

export const getUsuarioName = (id) => {
  return fetch('http://localhost:5000/api/users')
    .then(response => {
      if (!response.ok) {
        throw new Error('Error al obtener los usuarios');
      }
      return response.json();
    })
    .then(users => {
      const user = users.find(user => user.id === id);
      if (user) {
        return user;
      } else {
        throw new Error('Usuario no encontrado');
      }
    })
    .catch(error => {
      console.error('Error al obtener el usuario:', error.message);
      throw new Error('Error al obtener los usuarios');
    });
});

```

4- Añadimos la lógica en el componente `app.js`, (componente raíz), que utiliza `React Router DOM` para manejar las diferentes rutas dentro de la aplicación.

Importamos las dependencias necesarias de React y `React Router DOM` que nos da acceso a la biblioteca que permite utilizar las herramientas para la navegación y el enrutamiento en aplicaciones React. (`Routes`, `Routes`)

Se define un componente funcional llamado `App`. Este componente actúa como el componente principal de la aplicación. Dentro del componente `App`, se utiliza el componente `Routes de React Router DOM` para definir las diferentes rutas de la aplicación. El componente `Routes` actúa como el contenedor principal para todas las rutas definidas en la aplicación.

Dentro de `<Routes>`, utilizamos `<Route>` para especificar una URL y el componente que debe renderizarse cuando se accede a esa URL.

La primera ruta (`path="/"`) especifica que cuando la URL coincida con la raíz de la aplicación, se renderizará el componente `LoginForm`.

La segunda ruta (`path="/welcome/:Id"`) especifica que cuando la URL coincida con `"/welcome/:Id"`, se renderizará el componente `Welcome`. El `:Id` en la ruta indica un parámetro dinámico que puede ser accesible desde el componente `Welcome`.

Cada ruta utiliza la prop `'element'` para especificar el componente que debe renderizarse cuando la ruta coincida con la URL actual. El componente `App` se exporta como el componente principal de la aplicación, lo que significa que será el punto de entrada principal de la aplicación React.

```

src > JS App.js > ...
1
2 import React from 'react';
3 import { Routes, Route } from 'react-router-dom';
4 import LoginForm from './pages/Login';
5 import Welcome from './pages/Welcome';
6
7
8 const App = () => {
9   return (
10     <Routes>
11       <Route path="/" element={<LoginForm />} />
12       <Route path="/welcome/:Id" element={<Welcome/>} />
13     </Routes>
14   );
15 };
16
17 export default App;
18
19
20
21

```

5-Lógica de componentes hijos:

El componente **LoginForm** es un formulario de inicio de sesión que maneja la autenticación de usuarios mediante la llamada a la API por medio de **getUsuario(email, password)** y redirigirá al usuario a una pagina de bienvenida si sus credenciales son las correctas.

import React, { useState, useEffect } from 'react';: Importa los módulos necesarios de React, incluyendo **useState** y **useEffect**, que son hooks para manejar el estado y los efectos secundarios respectivamente.

import { getUsuario } from '../seviles/usuarios';: Importa la función **getUsuario** desde un archivo de servicios (probablemente un archivo que maneje la lógica para obtener información de usuario).

import { useNavigate } from 'react-router-dom';: Importa **useNavigate** de 'react-router-dom', que permite la navegación programática en **React Router**.



```
index.js JS App.js JS usuarios.js JS Login.js X #
> pages > JS Login.js > LoginForm
1 import React, { useState, useEffect } from 'react';
2 import { getUsuario } from '../seviles/usuarios';
3 import { useNavigate } from 'react-router-dom'
4
```

const LoginForm = () => {: Declara el componente de función **LoginForm**

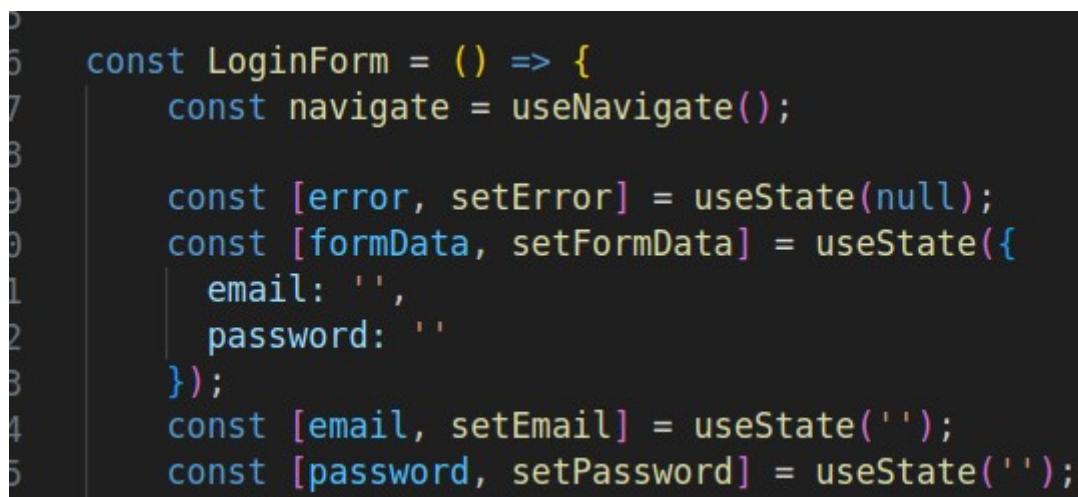
const navigate = useNavigate();: Pasamos el hook **useNavigate** para obtener la función de navegación programática, mas a delante utilizaremos esta constante y le pasaremos entre paréntesis la ruta a la que queremos dirigirnos.

const [error, setError] = useState(null);: Define un estado para almacenar mensajes de error.

const [formData, setFormData] = useState({ email: "", password: "" });: Define un estado para almacenar los datos del formulario.

const [password, setPassword] = useState("");: Define un estado para la contraseña del usuario.

const [email, setEmail] = useState("");: Define un estado para el correo electrónico del usuario.



```
5
6 const LoginForm = () => {
7   const navigate = useNavigate();
8
9   const [error, setError] = useState(null);
10  const [formData, setFormData] = useState({
11    email: '',
12    password: ''
13  });
14  const [email, setEmail] = useState('');
15  const [password, setPassword] = useState('');
```

useEffect(() => { ... }, [email, password]) Este efecto se ejecuta cada vez que email o password cambian. Verifica si ambos campos tienen valores .Llama a la función **getUsuario** para autenticar al usuario utilizando el correo electrónico y la contraseña y utiliza el manejo de asincronía **'then/catch'**

Si la autenticación es exitosa, limpia el error y redirige al usuario a la página de bienvenida utilizando **navigate(`/welcome/\${id}`)**, al mismo tiempo que pasa el Id del usuario como parámetro para que pueda ser utilizado por el componente **'Welcome'**.

Si la autenticación falla, muestra el mensaje de error devuelto por la API, `setError(error.message)`.

```
useEffect(() => {
  if (email && password) {
    getUsuario(email, password)
      .then(id => {
        setError(null);
        navigate(`/welcome/${id}`)
      })
      .catch(error => {
        setError(error.message);
      });
  }
}, [email, password]);
```

handleChange: Esta función se activa cuando los valores de los campos del formulario cambian, actualizando el estado de `formData` con los nuevos valores del campo.

handleSubmit: Esta función se activa cuando se envía el formulario. Establece los valores de email y password, `setEmail`, `setPassword`, a partir del estado actual de `formData`. También previene el comportamiento predeterminado del formulario.

```
const handleChange = e => {
  setError(null);
  const { name, value } = e.target;
  setFormData({
    ...formData,
    [name]: value
  });
};

const handleSubmit = (e) => {
  e.preventDefault();
  setEmail(formData.email)
  setPassword(formData.password)
};
```

Por ultimo se presenta un formulario con dos campos: **correo electrónico y contraseña**. Los campos de entrada están vinculados a los estados correspondientes (`formData.email` y `formData.password`). Cuando hay un error de autenticación, se muestra un mensaje de error en rojo arriba del formulario. `{error && <p style={{ color: 'red', marginLeft:'50px'}}>>{error}</p>}`. Al enviar el formulario, se activa la función **handleSubmit**.


```

return (
  <div>
    <h2>Iniciar Sesión</h2>
    {error && <p style={{ color: 'red', marginLeft: '50px'}}>{error}</p>}
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="email">Email:</label>
        <input autoComplete="email"
          type="email"
          id="email"
          name="email"
          value={formData.email}
          onChange={handleOnChange}
          required
        />
      </div>
      <div>
        <label htmlFor="password">Contraseña:</label>
        <input autoComplete="current-password"
          type="password"
          id="password"
          name="password"
          value={formData.password}
          onChange={handleOnChange}
          required
        />
      </div>
      <button type="submit">Iniciar Sesión</button>
    </form>
  </div>
);

export default LoginForm;

```

El componente 'Welcome' se encarga de mostrar la información del usuario una vez que se ha obtenido del servidor, mientras proporciona una interfaz de carga en caso de que la solicitud esté en curso. Utiliza el hook `useParams` para obtener el Id del usuario de la URL y el hook `useEffect` para manejar la lógica de obtención de datos del usuario.

Importamos `useParams` de `react-router-dom` para extraer parámetros de la URL que hemos pasado al enviar el formulario. Importamos `React`, `useEffect` y `useState` de `React` para crear el componente y manejar el estado y los efectos secundarios, también Importamos `getUsuarioName` para hacer el llamado a la API.

```

c > pages > JS Welcome.js > ...
1  import { useParams } from 'react-router-dom';
2  import { Link } from 'react-router-dom';
3  import React, { useEffect, useState } from 'react';
4  import { getUsuarioName } from '../services/usuarios';
5
6  const Welcome = () => {
7    const [error, setError] = useState(null);
8    const {Id} = useParams();
9    const [userData, setUserData] = useState(null)
10
11    useEffect(() => {
12      getUsuarioName(Id)
13        .then(userData => {
14          setError(null);
15          setUserData(userData);
16        })
17        .catch(error => {
18          setError(error.message);
19        });
20    }, [Id]);
21

```

`const [error, setError]` : Almacena cualquier error que pueda ocurrir durante la obtención de datos del usuario.

`const {Id} = useParams()` : Utiliza `useParams` para extraer el parámetro `Id` de la URL.

`const [userData, setUserData]` : Almacena los datos del usuario una vez que se han recuperado del servidor.

Utilizamos el hook `useEffect` que se ejecuta cada vez que `Id` cambia. Dentro del efecto, llamamos a `getUsuarioName(Id)` para obtener los datos del usuario correspondientes al `Id` proporcionado. Si la solicitud es exitosa, actualizamos el estado `userData` con los datos del usuario y restablecemos `error` a `null`. Si la solicitud falla, actualizamos el estado `error` con el `mensaje de error`. Finalmente en el valor de retorno renderizamos la información del usuario si `userData` está disponible.

Si `userData` está presente, mostramos el nombre, la edad y el correo electrónico del usuario. Si `userData` aún no está disponible, mostramos un mensaje indicando que los datos del usuario se están cargando. También incorpora un `<boton>` envuelto en un componente `<Link>` que nos regresa al componente inicia `'LoginForm'`

```
c > pages > JS Welcome.js > ...
6  const Welcome = () => {
21
22    return (
23      <div>
24        {userData ? (
25          <div>
26            <h2>Bienvenido, <span>{userData.name}</span></h2>
27            <p>Edad: <span>{userData.age}</span></p>
28            <p>Email: <span>{userData.email}</span></p>
29            <Link to="/">
30              <button>Regresar a Inicio</button>
31            </Link>
32          </div>
33        ) : (
34          <p>Cargando datos del usuario...</p>
35        )}
36      </div>
37    );
38  };
39
40
41  export default Welcome;
42
```

6-CSS

body: Establece reglas para el elemento `body` del documento HTML.

margin: 0; Elimina los márgenes externos del cuerpo del documento.

padding: 0; Elimina el relleno interno del cuerpo del documento.

font-family: sans-serif; Establece la fuente predeterminada del texto como una fuente sans-serif.

color: blue; Establece el color del texto del cuerpo del documento como azul.

background-color: #d3d3d357; Establece el color de fondo del cuerpo del documento como un tono de gris claro con un poco de transparencia.

padding-top: 10%; Establece un relleno en la parte superior del cuerpo del documento que es el 10% del alto del viewport.

#root: Establece reglas para un elemento HTML con un **ID de "root"**.

display: flex; Establece el modelo de caja flexible en el elemento, permitiendo el uso de flexbox para su contenido.

flex-direction: row; Establece la dirección principal del contenedor flex como una fila, lo que significa que sus elementos secundarios se colocarán uno al lado del otro horizontalmente.

justify-content: center; Centra los elementos secundarios horizontalmente dentro del contenedor flex.

form: Establece reglas para los elementos form del documento HTML.

text-align: center; Centra el texto dentro del formulario.

border: 3px solid white; Establece un borde sólido de 3 píxeles de grosor y color blanco alrededor del formulario.

padding: 10px; Agrega un relleno interno de 10 píxeles alrededor del formulario.

#password: Establece reglas para un elemento con el ID "password".

margin-right: 45px;; Establece un margen derecho de 45 píxeles para este elemento.

h2: Establece reglas para los elementos h2 del documento HTML.

text-align: center;; Centra el texto dentro de los elementos h2.

button: Establece reglas para los elementos button del documento HTML.

margin: 10px;; Establece un margen de 10 píxeles alrededor de los botones.

background: white;; Establece el color de fondo de los botones como blanco.

color: blue;; Establece el color del texto de los botones como azul.

input, button: Establece reglas comunes para los elementos input y button.

border: none;; Elimina los bordes de estos elementos.

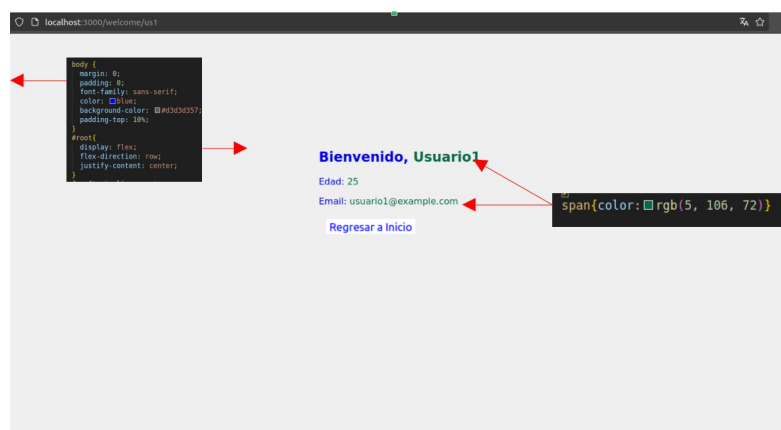
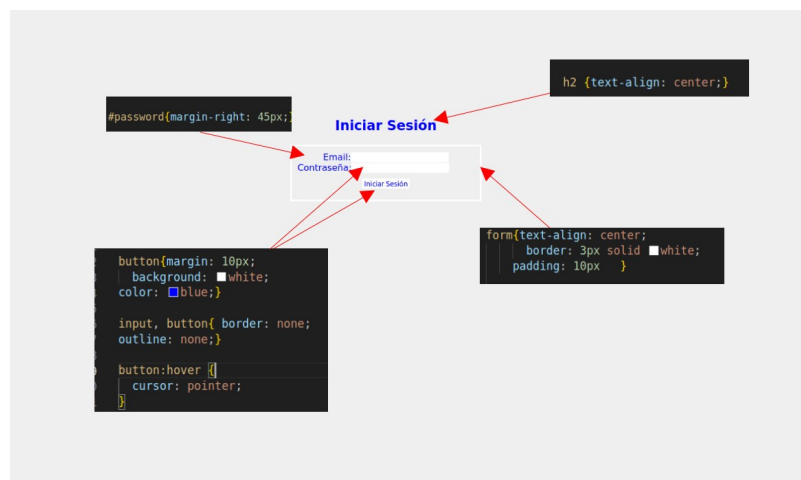
outline: none;; Elimina el contorno alrededor de estos elementos cuando están enfocados.

button:hover: Establece reglas para los botones cuando el cursor se desplaza sobre ellos.

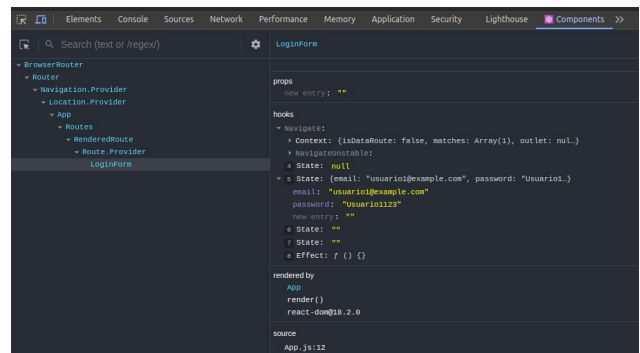
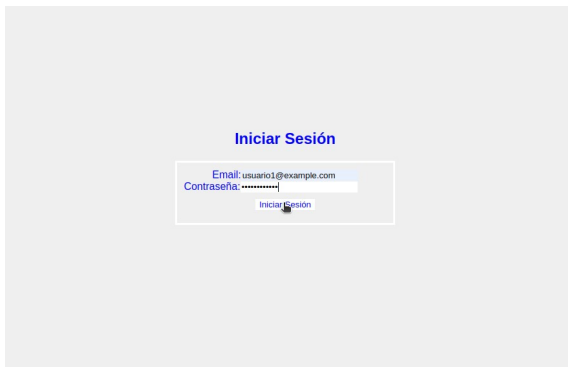
cursor: pointer;; Cambia el cursor del mouse a una mano, indicando que el botón es interactivo.

span: Establece reglas para los elementos span del documento HTML.

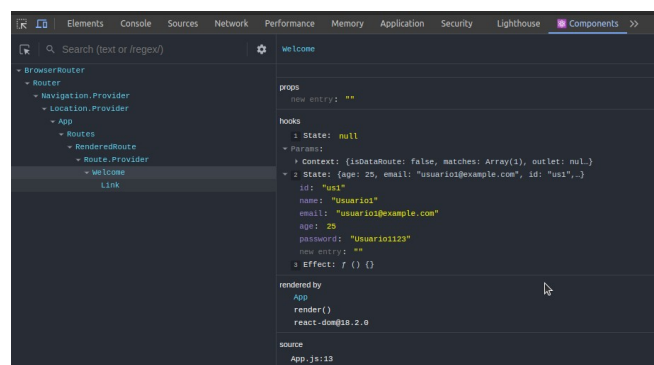
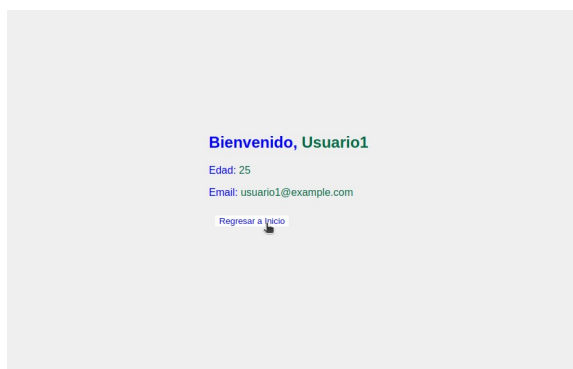
color: rgb(5, 106, 72);; Establece el color del texto de los elementos span como un tono de verde oscuro.



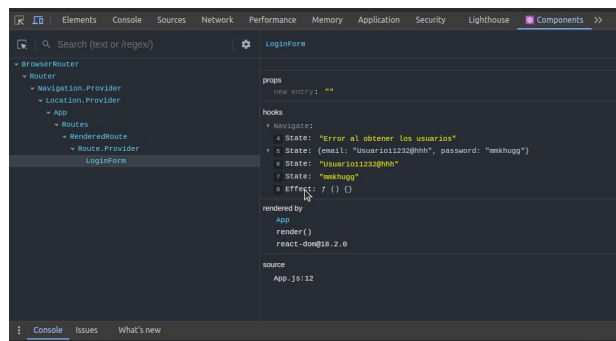
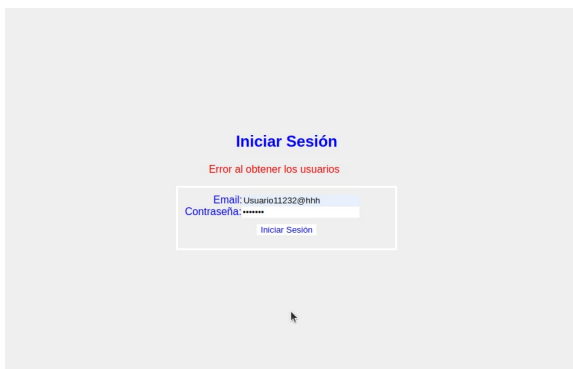
7- Por último comprobaremos la app en el navegador



* Ingresamos los datos en el formulario y enviamos



* La navegación programática nos dirige a la pagina de bienvenida



* Intentamos enviar el formulario pero las credenciales de usuario no coinciden con las registradas en la base de datos, por tanto el mensaje de error aparece.