

React Js

Guía de ejercicios complementarios

Sobre los ejercicios



Podemos identificar los elementos en:

Ejercicios: son propuestas de ejercitación práctica complementaria, basadas en problemáticas comunes del desarrollo de aplicaciones. Algunas consignas son específicas, pero otras son un poco más abiertas, con la intención de familiarizarnos con formatos narrativos en la solicitud de funcionalidades a programar.

Notas: cada ejercicio presenta una nota, la cual nos brinda más información o tips para desarrollarlo (nota: modo hardcore hacerlo sin leer las notas).

Resoluciones propuestas: encontrarás las resoluciones para que las compares con tus propias soluciones. Es propuesta por que **no es perfecta**, en desarrollo de software ninguna solución es infalible, y mucho depende del contexto, por lo cual si tu resolución presenta un funcionalidad similar o mejor a la propuesta ¡felicidades, resolviste el ejercicio!

Repositorio en Github: a partir de los ejercicios de clase 4 podrás encontrar las resoluciones alojadas en [Github](#).

Clase 1

Introducción a React Js

Actividad 1

Avatar

Crear un componente Boton, utilizando funciones del DOM.

Cada elemento debe estar contenido en una función, siendo un div con id="root" el punto de entrada en el DOM.



Notas actividad 1

Es posible hacerlo de varias maneras como, por ejemplo, utilizando la función createElement o innerHTML.

Actividad 2

Avatar con React

Continuando con la Actividad 1, replicar el mismo componente pero utilizando React, siguiendo la guía básica de prueba, donde agregamos React a un sitio html.

[Agregar React a un sitio web.](#)



Notas actividad 2

Este es un ejercicio para conocer lo puro de React, más adelante agregaremos mejoras que facilitarán mucho más la creación de nuestras interfaces

Resoluciones

Ejemplo de solución actividad 1

```
</head>
<body>
  <div id="root"></div>
  <script>
    const root = document.getElementById("root")

    const drawContainer = () => {
      const html = []

      html.push('<div class="container">')
      html.push(drawButton())
      html.push('</div>')

      return html.join('')
    }

    const drawButton = () => {
      const html = []

      html.push('<button class="btn">')
      html.push('Test')
      html.push('</button>')

      return html.join('')
    }

    root.innerHTML = drawContainer()
  </script>
```

Ejemplo de solución actividad 1

```
</head>
<body>
  <div id="root"></div>
  <script>
    const root = document.getElementById("root")

    const createContainer = () => {
      const div = document.createElement('div')
      div.classList.add('container')

      return div
    }

    const createButton = () => {
      const button = document.createElement('button')
      const label = document.createTextNode('Test')

      button.appendChild(label)

      button.classList.add('btn')

      return button
    }

    const container = createContainer()
    const button = createButton()

    container.appendChild(button)
    root.appendChild(container)
  </script>
</body>
```


Ejemplo de solución actividad 2

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>REACT</title>
    <script
      src="https://unpkg.com/react@18/umd/react.development.js"
      crossorigin
    ></script>
    <script
      src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
      crossorigin
    ></script>
  </head>
  <body>
    <div id="root"></div>
    <script>
      const e = React.createElement // Funcion para crear elementos de React

      const Container = () => {
        return e('div', { className: 'container' }, Button())
      }

      const Button = () => {
        return e('button', { className: 'btn' }, 'Test')
      }

      const domContainer = document.querySelector('#root');
      const root = ReactDOM.createRoot(domContainer);
      root.render(e(Container));
    </script>
  </body>
</html>
```

Clase 2

Instalación y configuración del entorno

Actividad 1

Avatar con React y JSX

Siguiendo la guía de prueba de React, vemos que podemos utilizar JSX (una extensión de javascript) que nos permite que sea mucho más fácil escribir nuestros componentes.

[Agregar React a un sitio web.](#)



Notas actividad 1

JSX nos permite escribir el llamado de funciones como si fueran elementos HTML, pero NO son elementos HTML. Ahora la función `React.createElement` es llamada de forma implícita.

Resoluciones

Ejemplo de solución actividad 1

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>REACT</title>
    <script
      src="https://unpkg.com/react@18/umd/react.development.js"
      crossorigin
    ></script>
    <script
      src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
      crossorigin
    ></script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/babel">
      const e = React.createElement // Funcion para crear elementos de React

      const Container = () => {
        return (
          <div className="container">
            <Button />
          </div>
        )
      }

      const Button = () => {
        return (
          <button className="btn">Test</button>
        )
      }

      const domContainer = document.querySelector("#root");
      const root = ReactDOM.createRoot(domContainer);
      root.render(<Container />);
    </script>
  </body>
</html>
```

Clase 3

JSX y transpiling

Actividad 1

Transformando Arrays

Implementar un polyfills del método map de los arrays.

Debemos suponer que los navegadores no son compatibles con este método y debemos implementarlo para que todos puedan usarlo en nuestra aplicación.



Notas actividad 1

Es necesario utilizar el objetos this, que en este caso va a hacer referencia al dueño de la función map que es ejecutada. Por eso mismo, necesitaremos de una función regular y no una arrow function. Recordar que necesitamos RETORNAR el array transformado.

Actividad 2

Filtrando Arrays

Del mismo modo podemos implementar un polyfill para el método filter de los Arrays.



Notas actividad 2

Entender el funcionamiento de estos métodos nos ayudará a comprender los principios de inmutabilidad de la programación funcional.

Esto utiliza React para poder hacer la manipulación del DOM de forma performante.

No olvidar que necesitamos RETORNAR un array con los elementos que cumplen con la condición del filtro, por lo que dentro necesitaremos de un IF.

Resoluciones

Ejemplo de solución actividades 1 y 2

```
<script>

  Array.prototype.customForEach = function (callback) {
    for(let index = 0; index < this.length; index++) {
      const elemento = this[index]
      callback(elemento, index)
    }
  }

  const numeros = [1, 2, 3]

  numeros.customForEach((num, index) => console.log(`el numero ${num} esta en el indice ${index}`))

</script>
```

Clase 4

Componentes I

Actividad 1

Botón Multiuso

Implementar un componente Button que reciba por props un color, un label y una función de callback a ejecutar en el evento onClick.

Este botón debe mantener los estilos cada vez que se lo utilice, con la opción de cambiar su color de fondo a través de props.



Notas actividad 1

Pueden agregarse más props, por ejemplo una para deshabilitarlo.

Actividad 2

Botón Multiuso

En un componente botón no es tan cómodo establecer su label como si fuese un atributo, sino que sería más fácil de entender si el texto lo escribo como hijo del componente. Aplicar la prop children para esto.



Notas actividad 2

Es importante recordar que en React todo se trata como elementos de React por lo que puedo escribir un texto como componente hijo.

Resoluciones

Ejemplo de solución actividad 1

```
import styles from './Button.module.css'

const Button = ({ color, label, callback }) => {
  return (
    <button style={{ backgroundColor: color }} onClick={callback} className={styles.button}>
      {label}
    </button>
  )
}

export default Button
```

Ejemplo de solución actividad 2

```
import styles from './Button.module.css'

const Button = ({ color, children, callback }) => {
  return (
    <button style={{ backgroundColor: color }} onClick={callback} className={styles.button}>
      {children}
    </button>
  )
}

export default Button
```


Clase 5

Componentes II

Actividad 1

ToDo List

Implementar un componente `TaskListContainer` que liste tareas por incompletas, cuando se haga click sobre una tarea esta debe cambiar a estado completado. Estos cambios de estado pueden representarse con estilos en cada uno de los elementos.



Notas actividad 1

El estado inicial debe un Array de objetos, y estos objetos deben tener las siguientes propiedades: `id`, `text`, `completed`.
Recordar que la actualización de un array puede hacerse con el método `map`.

Actividad 2

Refactorización de ToDo List

Ahora es momento de separar responsabilidades. Debe haber un componente contenedor que se encargue de la lógica de estado y otros componentes que se encarguen de la presentación de los datos.



Notas actividad 2

Los componentes de presentación no deben contener ningún tipo de lógica que esté relacionada directamente al estado de los datos.

Resoluciones


Ejemplo de solución actividad 1

```
const ToDoList = () => {  
  const [tasks, setTasks] = useState([  
    { id: "1", text: "Hacer la tarea 1", completed: false },  
    { id: "2", text: "Hacer la tarea 2", completed: false },  
    { id: "3", text: "Hacer la tarea 3", completed: false }  
  ]);  
  
  const handleClick = (id) => {  
    const newTasks = tasks.map(task => {  
      if(task.id === id) {  
        return { ...task, completed: !task.completed }  
      } else {  
        return task  
      }  
    });  
    setTasks(newTasks);  
  }  
  
  return (  
    <div>  
      <h1>Lista de tareas</h1>  
      <ul>  
        {tasks.map((task) => (  
          <li  
            key={task.id}  
            style={{ textDecoration: task.completed ? "line-through" : "none" }}  
            onClick={() => handleClick(task.id)}  
          >  
            {task.text}  
          </li>  
        ))}  
      </ul>  
    </div>  
  );  
}
```

Puedes acceder a la resolución de esta actividad en el [repositorio de Github](#). 🚀

Ejemplo de solución actividad 2

```
const ToDoList = () => {  
  const [tasks, setTasks] = useState([  
    { id: "1", text: "Hacer la tarea 1", completed: false },  
    { id: "2", text: "Hacer la tarea 2", completed: false },  
    { id: "3", text: "Hacer la tarea 3", completed: false }  
  ]);  
  
  const handleClick = (id) => {  
    const newTasks = tasks.map(task => {  
      if(task.id === id) {  
        return { ...task, completed: !task.completed }  
      } else {  
        return task  
      }  
    });  
    setTasks(newTasks);  
  }  
  
  return (  
    <div>  
      <h1>Lista de tareas</h1>  
      <TaskList tasks={tasks} onCompleted={handleClick}/>  
    </div>  
  );  
}
```

Puedes acceder a la resolución de esta actividad en el [repositorio de Github](#). 

Clase 6

Promises, asincronía y MAP

Actividad 1

ToDo List Asíncrono

Ya con el componente funcionando, podemos empezar a pensar en recibir este listado de un backend, pero como todavía no conocemos de backend, vamos a simularlo con una función sencilla que retorne una promesa que al resolverse devuelva el listado de tareas.



Notas actividad 1

Debemos retornar una promesa de esta función que realiza un fetch porque debemos simular el retardo de una request con un setTimeout.

Actividad 2

Mejorando la UX

Recuerda que va a haber un momento en el que la pantalla va a estar vacía, hasta que se resuelva la promesa. Para mejorar la experiencia del usuario podemos agregar un Loader o Spinner que le indique al usuario que su solicitud está siendo procesada. No nos olvidemos tampoco de manipular los errores.



Notas actividad 2

El estado de la promesa lo podemos sincronizar con un estado de loading, con este mismo validaremos si debemos o no mostrar el Loader.

Resoluciones

Ejemplo de solución actividad 1

```
const TodoList = () => {  
  const [tasks, setTasks] = useState([]);  
  
  useEffect(() => {  
    fetchTasks().then(tasks => {  
      setTasks(tasks);  
    });  
  }, []);  
  
  const handleClick = (id) => {  
    const newTasks = tasks.map(task => {  
      if(task.id === id) {  
        return { ...task, completed: !task.completed }  
      } else {  
        return task  
      }  
    });  
    setTasks(newTasks);  
  }  
  
  return (  
    <div>  
      <h1>Lista de tareas</h1>  
      <TaskList tasks={tasks} onCompleted={handleClick} />  
    </div>  
  );  
}
```

```
const fetchTasks = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve([  
        { id: "1", text: "Hacer la tarea 1", completed: false },  
        { id: "2", text: "Hacer la tarea 2", completed: false },  
        { id: "3", text: "Hacer la tarea 3", completed: false }  
      ]);  
    }, 1000);  
  });  
}
```

Ejemplo de solución actividad 2

```
const TodoList = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    fetchTasks().then(tasks => {
      setTasks(tasks);
    }).catch(error => {
      setNotification('error', 'hubo un error')
    }).finally(() => {
      setLoading(false)
    });
  }, []);

  > const handleClick = (id) => { ...
  }

  return (
    <div>
      <h1>Lista de tareas</h1>
      {
        loading ? (
          <h3>Cargando tareas...</h3>
        ) : (
          <TaskList tasks={tasks} onCompleted={handleClick} />
        )
      }
    </div>
  );
}
```

Clase 9

Routing y navegación

Actividad 1

Naveguemos

Es momento de darle un orden a nuestra aplicación. Podemos crear una ruta de bienvenida a nuestra app y una ruta donde podamos ver las task. No nos olvidemos de agregar un Navbar.



Notas actividad 1

El login debe sincronizar 2 inputs con su estado correspondiente. Al hacer click en login por el momento solo debería mostrar en consola los datos ingresados.

Actividad 2

Hola Anónimo

Podemos pensar que en un futuro necesitamos autenticar a los usuarios, para darle a cada uno sus tareas.

Creemos una ruta /login y mostremos un formulario.



Notas actividad 2

El componente Login debe sincronizar 2 inputs con su estado correspondiente. Al hacer click en login por el momento solo debería mostrar en consola los datos ingresados.

Resoluciones

Ejemplo de solución actividad 1

```
function App() {  
  return (  
    <div className="App">  
      <BrowserRouter>  
        <Navbar />  
        <Routes>  
          <Route path="/" element={<Home />}/>  
          <Route path="/tasks" element={<ToDoList />} />  
        </Routes>  
      </BrowserRouter>  
    </div>  
  );  
}
```

Ejemplo de solución actividad 2

```
function App() {  
  return (  
    <div className="App">  
      <BrowserRouter>  
        <Navbar />  
        <Routes>  
          <Route path="/" element={<Home />}/>  
          <Route path="/tasks" element={<ToDoList />} />  
          <Route path="/login" element={<Login />} />  
        </Routes>  
      </BrowserRouter>  
    </div>  
  );  
}
```

Clase 10

Eventos

Actividad 1

¿Qué debo hacer primero?

Cuando tenemos muchas tareas quizás no sepamos por dónde empezar.

¿Por qué no le damos la opción al usuario de elegir el grado de urgencia a través de un dropdown?



Notas actividad 1

Este dropdown puede ser un componente select con opciones.

Actividad 2

Refactoricemos

La lógica del dropdown puede ser un poco compleja, y en caso volver a necesitarlo, no queremos volver a escribirla, así que refactoricemos este select y creemos un componente orientado al evento que ejecuta.




Notas actividad 2

La función principal del componente que tendrá esta lógica compleja será la de ejecutar un evento y devolver un valor al componente padre.

Resoluciones

Ejemplo de solución actividad 1

```
const Task = ({ id, text, completed, priority }) => {  
  const options = [  
    { id: '1', text: 'Baja', value: 1 },  
    { id: '2', text: 'Media', value: 2 },  
    { id: '3', text: 'Alta', value: 3 }  
  ]  
  
  const { handleChangeTaskState, handleChangeTaskPriority } = useContext(TasksContext)  
  
  const handleOnSelect = (value) => {  
    handleChangeTaskPriority(id, value)  
  }  
  
  return (  
    <article  
      className={styles.task}  
    >  
      <h3>{text}</h3>  
      <select  
        onChange={(e) => handleOnSelect(e.target.value)}  
        value={priority}  
      >  
        {options.map(option => <option key={option.id} value={option.value}>{option.text}</option>)}  
      </select>  
      <Button  
        color={completed ? 'green' : 'red'}  
        callback={() => handleChangeTaskState(id)}  
      >  
    </article>  
  )  
}
```

Puedes acceder a la resolución de esta actividad en el [repositorio de Github](#). 

Ejemplo de solución actividad 2

```
const Select = ({options, defaultValue, onSelect}) => {  
  return (  
    <select  
      onChange={(e) => onSelect(e.target.value)}  
      value={defaultValue}  
    >  
      {options.map(option => <option key={option.id} value={option.value}>{option.text}</option>)}  
    </select>  
  )  
}  
  
export default Select
```


Clase 11

Context

Actividad 1

¡Hola, Usuario!

Si nuestra idea es mostrarles información distinta a los usuarios sin mostrarle tareas a los anónimos, toda nuestra aplicación deberá estar validando con un estado donde estén guardados los datos del usuario.

¿Por qué no implementamos un Custom Context que lo almacene e inyecte las funciones correspondientes?



Notas actividad 1

Este Custom Context (AuthContext) debe inyectar la función de login que setea el estado de usuario y la de logout que limpia el estado de usuario. Utilizar la función de login en el componente Login.

Actividad 2

Privacidad de rutas

Ahora tenemos dos tipos de usuarios, los autenticados y los anónimos, pero los anónimos no deberían poder acceder a la ruta task y los usuarios ya autenticados no deberían poder acceder a la ruta de login. Vamos a protegerlas.

Para esto vamos a necesitar de la librería react-router-dom y de nuestro AuthContext. Necesitamos dos componentes: un ProtectedRoute y un PublicRoute



Notas actividad 2

Podemos utilizar los componentes:

[Navigate](#) y [Outlet de react-router-dom](#)

Resoluciones

Ejemplo de solución actividad 1

```
import { createContext, useState } from "react"; 4.4k (gzipped: 1.9k)


export const AuthContext = createContext({ tasks: []})

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null)

  const login = (data) => {
    setUser({ user: data.email })
  }

  const logout = () => {
    setUser(null)
  }

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      { children }
    </AuthContext.Provider>
  )
}
```

Puedes acceder a la resolución de esta actividad en el [repositorio de Github](#). 

Ejemplo de solución actividad 2

```
import {
  Navigate,
  Outlet,
  useLocation
} from 'react-router-dom'; 2.4k (gzipped: 1.1k)

const ProtectedRoute = ({
  user,
  redirectPath = '/',
  children,
}) => {
  const location = useLocation()

  if (!user) {
    return <Navigate to={redirectPath} state={{ from: location }} replace />;
  }

  return children ? children : <Outlet />;
};

export default ProtectedRoute
```

```
import {
  Navigate,
  Outlet
} from 'react-router-dom'; 2.4k (gzipped: 1.1k)

const PublicRoute = ({
  user,
  redirectPath = '/',
  children,
}) => {
  if (user) {
    return <Navigate to={redirectPath} replace />;
  }

  return children ? children : <Outlet />;
};

export default PublicRoute
```

Ejemplo de solución actividad 2

```
const AppRouter = () => {  
  const { user } = useContext(AuthContext)  
  
  return (  
    <Routes>  
      <Route path="/" element={<Home />} />  
  
      <Route element={<PublicRoute user={user} redirectPath="/profile"/>>  
        <Route path="/login" element={<Login />} />  
      </Route>  
  
      <Route element={<ProtectedRoute user={user}/>>  
        <Route path="/tasks" element={<TodoList />} />  
      </Route>  
    </Routes>  
  )  
}
```

```
function App() {  
  return (  
    <div className="App">  
      <AuthProvider>  
        <BrowserRouter>  
          <Navbar />  
          <AppRouter />  
        </BrowserRouter>  
      </AuthProvider>  
    </div>  
  );  
}
```

Clase 13

Firestore I

Actividad 1

Nuestro Backend

Es momento de migrar nuestra simulación de backend (mock) a un backend real. Debemos reemplazarlo por Firebase y utilizar el servicio de base de datos Firestore.



Notas actividad 1

Debemos crear una colección 'tasks' donde pasaremos nuestras tareas y luego debemos obtenerlas en nuestro componente ToDoList.

Actividad 2

Modelando desde la base de datos

Es posible que aquellos datos que muestra nuestra aplicación cambien, como por ejemplo las clases de urgencias de nuestras tareas. Podemos también guardarlas en una colección en firestore y obtenerlas para llenar el select.



Notas actividad 2

Podemos crear una colección 'priority' y crear documentos que tengan los datos correspondientes.

Resoluciones

Ejemplo de solución actividad 1

```
export const getTasks = () => {  
  const tasksRef = query(collection(db, 'tasks'), orderBy('priority', 'desc'))  
  
  return getDocs(tasksRef).then(snapshot => {  
    const tasksAdapted = snapshot.docs.map(doc => createTaskAdaptedFromFirestore(doc))  
    return tasksAdapted  
  }).catch(error => {  
    return error  
  })  
}
```

Ejemplo de solución actividad 2

```
export const getPriorities = () => {  
  const prioritiesRef = query(collection(db, 'priorities'), orderBy('value', 'desc'))  
  
  return getDocs(prioritiesRef).then(snapshot => {  
    const prioritiesAdapted = snapshot.docs.map(doc => createPriorityAdaptedFromFirestore(doc))  
    return prioritiesAdapted  
  }).catch(error => {  
    return error  
  })  
}
```

Clase 14

Firestore II

Actividad 1

Actualicemos el estado de nuestras tareas

Es hora de modificar el estado de nuestras tareas y poder guardarlo. Para eso debemos actualizar el campo completed de la tarea en firestore.



Notas actividad 1

Debemos hacer un `updateDoc` del documento al cual le queremos cambiar el estado.

Actividad 2

Actualicemos la prioridad de nuestras tareas

Es posible que a lo largo del tiempo la prioridad de nuestras tareas cambien, por eso debemos poder actualizarlas y guardarlas en nuestra base de datos.



Notas actividad 2

Ahora debemos hacer un updateDoc del campo priority

Resoluciones


Ejemplo de solución actividad 1

```
export const updateTask = (taskId, data) => {  
  const taskRef = doc(db, 'tasks', taskId)  
  
  return updateDoc(taskRef, data).then(() => {  
    return true  
  }).catch(error => {  
    return error  
  })  
}
```

```
const handleChangeTaskState = (id) => {  
  const updatedTasks = tasks.map(task => {  
    if(task.id === id) {  
      updateTask(task.id, { completed: !task.completed })  
      return { ...task, completed: !task.completed }  
    } else {  
      return task  
    }  
  })  
  
  setTasks(updatedTasks)  
}
```

Ejemplo de solución actividad 2

```
const handleChangeTaskPriority = (id, value) => {  
  const updatedTasks = tasks.map(task => {  
    if(task.id === id) {  
      updateTask(task.id, { priority: value })  
      return { ...task, priority: value }  
    } else {  
      return task  
    }  
  })  
  
  setTasks(updatedTasks)  
}
```

Puedes acceder a la resolución de esta actividad en el [repositorio de Github](#). 

Clase 15

Workshop final

Actividad 1

Firebase Auth

Nuestra aplicación ya estaría casi lista, ahora necesitamos poder registrar usuario de verdad, para eso podemos utilizar Firebase Auth. Este tiene diferentes métodos de login, puedes elegir el que te resulte mejor.

Recorda guardar los datos del usuario en una coleccion para utilizarlos en las demas colecciones y en la aplicacion.



Notas actividad 1

El login debería realizarse dentro del AuthContext y al obtener el objeto del usuario guardarlo en el estado de nuestro context.

Deberás usar el ID del usuario en los campos user de cada task para darle un propietario. (primero deberias agregarlo manualmente para probar)

Actividad 2

Creando tareas para el usuario autenticado

Ahora que tenemos al usuario identificado, podemos dejarlo crear task para sí mismo.

Deberíamos implementar una nueva ruta /createtask con un formulario de creación de task y a esta task agregarle el ID del usuario logueado que tenemos en nuestro AuthContext.



Notas actividad 2

Es importante no olvidarse el ID del usuario para así poder filtrarlas al mostrarlas en ToDoList.

Resoluciones

Ejemplo de solución actividad 1

```
const signInWithGithub = async (callback) => {  
  const provider = new GithubAuthProvider()  
  
  return signInWithPopup(auth, provider)  
    .then(async (userCredencial) => {  
      const user = userCredencial.user  
      const userData = await handleUser(user)  
      callback()  
      return userData  
    })  
    .catch(error => {  
      console.log(error)  
    })  
}
```

```
const handleUser = async (rawUser) => {  
  if(rawUser) {  
    const user = formatUser(rawUser)  
    const { token, ...userWithoutToken } = user  
    const createdUser = await createUser(user.uid, userWithoutToken)  
    setUser(createdUser)  
    return user  
  } else {  
    setUser(null)  
    return false  
  }  
}
```


Ejemplo de solución actividad 2

```
export const createTask = (newTaskWithUser) => {  
  const tasksRef = collection(db, 'tasks')  
  
  return addDoc(tasksRef, newTaskWithUser)  
}
```

```
<Route element={<ProtectedRoute user={user}/>}>  
  <Route element={<TasksProvider />}>  
    <Route path='/tasks' element={<TaskListContainer />} />  
    <Route path='/createtask' element={<CreateTask />} />  
  </Route>  
</Route>
```

```
const createTaskAndRefresh = (newTask) => {  
  const newTaskWithUser = {  
    ...newTask,  
    user: user.uid  
  }  
  
  return createTask(newTaskWithUser).then(snapshot => {  
    return getTasks(user.uid)  
  }).then(tasksUpdated => {  
    setTasks(tasksUpdated)  
    return true  
  }).catch(error => {  
    return error  
  })  
}
```