React App: Login with Formik, Yup and Context API

This application serves as a Proof of Concept (PoC) for:

* Authentication
* Form validation
* Page routing

1. POC 1: Authentication with Context API

This section explains how user authentication is implemented using React's built-in Context API, ensuring that the authentication state (login/ logout) is globally accessible throughout the application. How to use it:

* Setup: It creates a new context called AuthContext to share authentication data globally.

import { createContext, useContext, useState } from "react";

const AuthContext = createContext();

* AuthProvider Component: It holds the authentication state (isAuthenticated), it provides login() and logout() methods and wraps the app, so that all children components can access authentication-related data.

export const AuthProvider = ({ children }) => {

  // State to track if the user is authenticated

  const [isAuthenticated, setAuthenticated] = useState(false);

  // Define login and logout functions

  const login = () => setAuthenticated(true);

  const logout = () => setAuthenticated(false);

  return (

    // Provide the authentication state and functions to all child components

    <AuthContext.Provider value={{ isAuthenticated, login, logout }}>

      {children}

    </AuthContext.Provider>

  );

};

* useAuth Custom Hook: A hook to access AuthContext

// Custom hook to easily access the AuthContext in any component

export const useAuth = () => useContext(AuthContext);

// Allows any component to use:

const { isAuthenticated, login, logout } = useAuth();

* Private routing: It prevents unauthorized users from accessing restricted pages. If the user is not authenticated, they are redirected to the login page.

function PrivateRoute({ children }) {

  const { isAuthenticated } = useAuth();

  return isAuthenticated ? children : <Navigate to="/" />;

}

This component is used to wrap private pages within the routing setup:

<Route

          path="/dashboard"

          element={

            <PrivateRoute>

              <Dashboard />

            </PrivateRoute>

          }

 />

1. POC 2: Form Validation with Formik and Yup

This section explains how the login form validation is implemented using Formik to manage form state and Yup to define and enforce validation rules. These tools simplify form handling and ensure robust, declarative validation. Formik simplifies form handling by automatically managing:

* Field values and changes
* Validation status and errors
* Form submission lifecycle

Yup provides a declarative and reusable schema for validation rules, supporting:

* Basic validations: required fields, min/max length, email format
* Complex validations: regex patterns, conditional validation, nested objects, custom tests

Example validations supported by Yup:

Yup.string().matches(/regex/) for pattern matching

Yup.number().min(0).max(100) for numeric ranges

Yup.boolean().isTrue() for required boolean flags

Using Formik + Yup together promotes cleaner, maintainable code by separating form logic and validation rules, improving user experience with immediate feedback and preventing invalid submissions. How to use it:

* Setup

import { Formik, Form, Field, ErrorMessage } from "formik";

import \* as Yup from "yup";

* Defining the Validation Schema with Yup:
* email: Must be a valid email string and is required
* password: Must be at least 6 characters, must contain at least one uppercase letter, one number, and one special character and is required
* Custom error messages will be displayed if validation rules are not met

const LoginSchema = Yup.object().shape({

email: Yup.string().email("Invalid email").required("Required"),

password: Yup.string()

  .min(6, "At least 6 characters")

  .matches(

      /^(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]+$/,

      "Password must contain at least one uppercase letter, one number, and one special character"

    )

    .required("Required"),

});

* (?=.\*[A-Z]) – ensures at least one uppercase letter
* (?=.\*\d) – ensures at least one digit (number)
* (?=.\*[@$!%\*?&]) – ensures at least one special character (from the set @$!%\*?&)
* [A-Za-z\d@$!%\*?&]+ – allows only letters, numbers, and those special characters
* The LoginPage Component

function LoginPage() {

  const navigate = useNavigate();

  const { login } = useAuth();

  return (

    <div className="login-container">

      <div className="login-card">

        <h2 className="login-title">Login</h2>

        <Formik

          initialValues={{ email: "", password: "" }}

          validationSchema={LoginSchema}

          onSubmit={(values, { setSubmitting, setErrors }) => {

            if (

              values.email === "test@example.com" &&

              values.password === "123456A\*"

            ) {

              login();

              navigate("/dashboard");

            } else {

              setErrors({ password: "Invalid email or password" });

            }

            setSubmitting(false);

          }}

        >

          {({ isSubmitting }) => (

            <Form>

              <div className="form-group">

                <label htmlFor="email">Email:</label>

                <Field

                  type="email"

                  name="email"

                  id="email"

                  className="form-control"

                />

                <ErrorMessage

                  name="email"

                  component="div"

                  className="error-message"

                />

              </div>

              <div className="form-group">

                <label htmlFor="password">Password:</label>

                <Field

                  type="password"

                  name="password"

                  id="password"

                  className="form-control"

                />

                <ErrorMessage

                  name="password"

                  component="div"

                  className="error-message"

                />

              </div>

              <button

                type="submit"

                disabled={isSubmitting}

                className="submit-button"

              >

                {isSubmitting ? "Signing in..." : "Sign In"}

              </button>

            </Form>

          )}

        </Formik>

      </div>

    </div>

  );

}

export default LoginPage;

* How it works:
* Formik manages the form state, including input values, touched fields, and submission status (isSubmitting)
* The validationSchema prop connects the Yup schema to Formik, which automatically runs validation whenever values change or on form submission
* On submit:
  + If the email and password match (test@example.com and 123456A\*), the login() method from authentication context is called and the user is redirected to /dashboard
* Otherwise, a custom error message is set on the password field using Formik's setErrors
* The <Field> components represent controlled inputs managed by Formik, automatically wiring up value and change handlers.
* The <ErrorMessage> components display validation or submission errors directly beneath each field, improving user feedback
* The submit button is disabled during form submission (isSubmitting ), preventing multiple submissions

1. POC 3: Page Routing with react-router-dom

This section demonstrates how page navigation is handled in the React app using the react-router-dom library. It enables seamless navigation between components like the login page and the dashboard, while also supporting protected routes based on authentication status. React-router-dom provides a robust and declarative way to handle routing in a single-page application How it works:

* Route configuration: The app defines two main routes:
  + / : LoginPage
  + /dashboard : Dashboard (accessible only when the user is authenticated)

<Route path="/" element={<LoginPage />} />

<Route path="/dashboard" element={ <PrivateRoute> <Dashboard /> </PrivateRoute>}/>

* Global Provider: The entire routing structure is wrapped in the AuthProvider, making the authentication state accessible throughout the app via the Context API.
* In addition to declarative routing, the app uses programmatic navigation via the useNavigate hook from react-router-dom. This allows redirecting users in response to actions like a successful login. Inside LoginPage, once the credentials are validated, the user is redirected to the dashboard like this:

const navigate = useNavigate();

...

login(); // updates auth state

navigate("/dashboard"); // navigates to the dashboard

This improves user experience by automatically guiding them to the appropriate page after authentication without requiring manual input.