POLITECHNIKA ŚWIĘTOKRZYSKA

Advanced frontend applications – lecture 3

Routing, forms and navigation management

mgr inż. Mateusz Pawełkiewicz

1.10.2025

Lecture Objective: To demonstrate how to create complex user interfaces using routing (multi-page navigation in SPAs) and interactive forms with validation. We'll discuss **the React Router DOM** (including createBrowserRouter , useNavigate , and useParams), front-end code organization (pages / , layouts / , and components / directories), dynamic and nested navigation, form handling with the **react-hook-form library** and validation (e.g., using the **Zod** or **Yup libraries**), the controlled vs. uncontrolled component pattern, data passing between pages (e.g., via **state** in useNavigate), and view authorization using private routes (**ProtectedRoute**). Finally, we'll walk through a sample mini-app with the **/login** , **/ dashboard** , and **/profile/:id routes** , containing a login form with validation and redirection after login.

React Router DOM – createBrowserRouter , useNavigate , useParams

React Router DOM is a library for handling routing in React applications . The latest versions (v6+) introduce a new API that simplifies route definition and data management. One of the key features is **createBrowserRouter** , recommended for creating a router for web applications. createBrowserRouter leverages the browser's History API for history navigation and enables the use of *Data APIs* (e.g., *loaders* , *actions*) introduced in React Router v6.4. A router created with createBrowserRouter is typically defined outside the React tree (e.g., in a separate module), passing an array of route configuration objects. Then, in the application's rendering, we use the <RouterProvider> component with the created router. For example, we can define a main route with a layout component and nested child routes (children), which we will discuss in the section on nested navigation.

useNavigate is a React Router hook that enables programmatic navigation (redirection) in response to user events or other actions. It returns a navigate (to, options) function , which can be called, for example, after a form submit or in a click handler to change the application page. In React Router v6 , useNavigate replaced the earlier hook. useHistory and has become the primary method of imperative navigation. As described in the documentation, useNavigate allows you *to programmatically navigate the browser in response to user interactions* . The most common use case is redirecting after some action—for example, after a successful login or after clicking the "Next" button. This hook is often used *after form submissions or in click events* . By calling navigate ("/ target-url ", { state: {...} }) , you can optionally pass a state object to a new location (more on this in the data passing section). The first argument to navigate is the target URL (path), and the second is an optional configuration object, which contains, among other things, the state key to pass the data.

useParams is a hook that returns a key-value pair object with dynamic parameters extracted from the current URL (for the component rendered in the context of a given route). We use it in page components to retrieve, for example, an identifier from a URL (such as :id in a routing path). For example, if we have a route defined as /profile/:id , then within the component corresponding to that route, calling const { id } = useParams (); will allow us to retrieve the value of the :id parameter from the URL and, for example, use it to retrieve data for a user with a given ID. In practice, useParams is used to retrieve dynamic fragments of a URL path and use them, for example, for API queries or data filtering.

View structure: pages /, layouts /, components / directories

When developing larger React applications , it's important to ensure code readability and organization through appropriate directory and component structure. A common approach is to divide the code into directories: **pages /** (pages), **layouts /** (layouts), and **components /** (reusable components).

- **Pages /** : Here we place components corresponding to individual pages/routes of the application. Each file in pages / typically exports one component representing a screen (view) accessible at a specific URL. Pages are the entry points for user interaction with the application, so it is logical to organize the project around the pages / folder . In SPAs (client-driven React , e.g., using Vite), files in pages / are not automatically mapped to routes (unlike, e.g., Next.js), but we still maintain page components there, and we define the mapping to paths manually in the router. This division increases readability – you can immediately see which views (subpages) exist.
- **Layouts /** : This directory holds layout components, which are common interface sections for many pages, such as main navigation, headers, footers, page templates, etc. A layout is a component that usually contains fixed UI elements (e.g., menu bar, header) and **an < Outlet />** from React Router, through which matching nested routes are rendered . Thanks to layouts, we can define nested routes in the router – for example, we can have a main route with a layout and nested routes for subpages inheriting this layout within it . For example, we can create a MainLayout component with a header and a menu, in which we call < Outlet /> in place of variable content. In the route definition, we then set < Route path ="/" element={< MainLayout />}> and inside it another < Route path="name " element={< GivePage />}/> for subpages . This pattern allows you to maintain a consistent look between pages and clearly separates the logic: the layout handles fixed elements (e.g., navigation), and the pages provide unique content. In React Router v6 , the syntax is as follows: in <Routes> , we define a parent route with a layout element and *children* as nested <Route> with only relative paths and page components. In the layout component itself , we import the Outlet from react-router-dom and in JSX, we place the < Outlet /> where the content changing depending on the current subpage should appear . This ensures that all nested routes will render in a common layout skeleton (e.g., under the header and menu defined in the layout).
- **Components /** : Here we place smaller, **reusable presentation components** , which aren't full pages, but rather snippets of the interface used in various places. These could be buttons, form fields, tabs that display data, etc. These components are usually stateless or contain their own logic but don't know anything about routing . By separating them into a separate directory, it's easier to maintain order and share code between pages. For example, if we have a login and registration form, we can have a common TextInput component in components / that handles displaying the <input> with a label and error message, instead of duplicating this code across both forms.

This modular structure (pages, layouts , components) facilitates project scalability and team collaboration – developers can easily locate the code corresponding to a given subpage or

functionality. This is similar to the *page-driven approach* known from Next.js (where files in the pages folder automatically define routing). In pure React + React Router, we create the mapping ourselves, but it's also worth using the pages / folder as an entry collection. points (views). In terms of layouts , some divide them even further, for example, separate layouts for the public and logged-in sections of the application. It's important to maintain consistency in organization.

Dynamic and nested navigation

Dynamic navigation means supporting routes with variable parameters, so a single route definition can handle many similar subpages . An example is a user profile at /profile/:id – instead of defining separate paths for each ID, we use the :id notation as **a route parameter** . React Router allows you to easily read this parameter inside a component via the aforementioned hook. useParams . Dynamic routing allows you to create components that render for different data depending on a URL parameter. For example, a separate view for each product, article, or user, but using a single template component. In practice, this means you can build a single page, e.g., a ProfilePage , that displays a user profile with an ID retrieved from the URL. When the user navigates to /profile/7 or /profile/42 , the same page will be rendered , and in useParams () you'll get { id: '7' } or { id: '42' } , respectively . **Dynamic routing allows you to create components for each element based on a URL parameter (e.g., slug or ID)** —this is especially useful when displaying data collections (lists and details).

Nested navigation refers to situations where one route is **inside another** —that is, we have a view hierarchy. In React Router, this is achieved by defining *child routes* (children) inside the *parent route* . As described earlier, a typical use is layout : for example, the main route "/" contains a common layout with a menu, and underneath it are nested routes "/ dashboard ", "/profile/:id", etc., which display inside this layout . In the route definition (e.g., in createBrowserRouter or in JSX with < Routes >), the parent route has an element= < LayoutComponent > attribute and a children field (in the case of createBrowserRouter) or a nested < Route > inside (in the case of declarative JSX). Each child route renders wherever the < Outlet /> is in the layout . Example in JSX code (React Route v6):

```
< Routes >
  < Route path = "/" element = { < Layout />}>
    < Route index element = { < HomePage />} />
    < Route path = "about" element = { < AboutPage />} />
    < Route path = "profile/:id" element = { < ProfilePage />} />
  </ Route >
</ Routes >
```

Here < Route path ="/" element={< Layout />}> surrounds the remaining definitions – this means that for each path defined inside it, it will be rendered first Layout , and its <Outlet> will contain the appropriate page element (HomePage , AboutPage , ProfilePage , depending on the subpath). It's also possible to delve more than one level (layout within a layout) – React Router supports this. Nested navigation simplifies the creation of multi-page applications, and also enables **partial**

navigation (only a portion of the page changes) and the behavior of parent components in the background when changing children. This is convenient, for example, in administration panels or tabbed dashboards .

It's worth mentioning **programmatic vs. declarative navigation** : Programmatic is, for example, using navigate () in component logic (imperative), while declarative is using <Link to="..."> links in JSX. Nested routes work the same in both approaches – the link generates the appropriate URL, and the router renders the layout+child hierarchy . Navigation links (e.g., menus) should use the <Link> or <NavLink> component instead of the usual <a> – this allows us to use the SPA mechanism without reloading the page. Example link: < NavLink to="/ about " > About </NavLink> renders as an anchor, but clicking switches the route within the React app. In the layout snippet above , we saw the use of Home in the menu, although in practice it's better to use <Link> there

In summary, dynamic routes allow you to handle *variable URL fragments* (like IDs), and nested routes allow you to keep repeatable elements *dry and create a hierarchy of views. These two concepts often coexist – for example,* / users /: userId / settings can be a nested route (settings) inside a user profile route.

Controlled Pattern vs Uncontrolled Components

Before discussing form handling, it's helpful to understand the two ways of handling form elements in React : controlled and **uncontrolled components** . They differ in their approach to storing and updating the state of form fields.

- **Controlled Components :** In this approach, React (the component's state) **controls the input value** . The form element (e.g., <input>) has its value set via the value attribute associated with the state (useState or state from the parent), and each change (onChange event) triggers a handler that updates the state in React . In other words, the form state is stored in React memory , and each update triggers a render (synchronizing the UI with the new state). **A controlled component therefore has a single source of truth— the React state** . The advantage is predictability and centralized control over input data: at any time, the application state accurately reflects the field values, which facilitates live validation, reacting to changes, etc. Furthermore, this means, for example, resetting a form is simply a state change, and formatting or restrictions can be easily applied (e.g., preventing letters from being entered into a numeric field). The drawbacks can be performance—each keystroke causes the component to render —and more code (you have to write value = {...} onChange = {...} for each field).
- **Uncontrolled components :** Here, **the state of the form element is managed by the browser (DOM)** , not directly by React . Form fields work like in pure HTML – they have an internal state (a pre-filled value) that React doesn't interfere with on an ongoing basis. To read values, we use a reference (ref) or event read. submit . For example, we can have < input ref={ myRef } defaultValue="start "> and only read myRef.current.value when the form is submitted . **In uncontrolled components, the component manages its state**

internally, instead of relying on React state . React doesn't force every update through its render cycle , so changes to the field don't cause the parent component to re-render on every character. This can provide performance improvements for large forms. It's closer to native form behavior and can be simpler when we don't need to react to every change immediately.

Key differences: In controlled components, each input change triggers onChange and updates React's state , ensuring *full control* and one-way data flow (the source of truth is state, and the UI reflects it). In uncontrolled components, the state is *independent of React* , read only on submit , for example , which means no immediate synchronization but *less overhead* (React doesn't render continuously).

When to use which? Controlled components are preferred when you need on-the-fly validation, dynamically enabling/disabling buttons based on input , or generally tight control over interaction (they're more predictable and easier to test in these scenarios). Uncontrolled components, on the other hand, are suitable for simpler forms where you don't need to react while someone is typing—you can let the user fill out the form and then collect the data after submitting . They're also used when integrating with external libraries that don't work with React state (e.g., some UI libraries can manipulate the DOM). **React Hook Form** (more on that in a moment) uses an uncontrolled approach under the hood for optimization, keeping the number of renders to a minimum even in complex forms.

In practice, the controlled approach is often used *for single fields or small forms* , while for larger or more complex forms it is worth reaching for libraries that effectively manage uncontrolled fields to avoid overloading the application.

Form handling: React Hook Form and Validation (Zod / Yup)

Writing forms in pure React can be time-consuming – you have to deal with the state of each field, validation, error display, and so on. Fortunately, there are libraries that simplify these tasks. One of the most popular is **React Hook Form (RHF)** , which allows you to easily build forms using **hooks** and an uncontrolled component approach. For validation, it often integrates with external data schema libraries like **Yup** or **Zod** to declaratively describe form validation rules.

React Hook Form (RHF) was designed to **simplify form validation as much as possible** and **minimize the amount of code and re-renders** needed to handle the form. It uses internally **uncontrolled components** (native inputs) and refs for managing field values, so that updating values does not cause constant refreshing of the React state . The creators of RHF noticed that with large forms, the controlled approach can cause performance degradation, hence their library provides a hook- based API for handling forms. Using RHF typically involves:

- Calling the hook useForm () in the form component, which returns methods and objects for managing the form (e.g. register , handleSubmit , formState : { errors }).

- Each form field (e.g. < input > , < select >) is registered using the register function obtained from useForm () . Example: < input {...register("email")} /> – this connects the field named "email" to the RHF mechanism.
- handleSubmit (onSubmit) provides a function that we can provide, for example, in the onSubmit of a form. RHF will automatically collect values from registered fields and pass them to our onSubmit function (only if validation passes).
- Any validation errors are available in the errors object (e.g. errors.email), so we can easily display the message: { errors.email && { errors.email.message } } .

Validation : RHF supports validation in several ways. The simplest is directly in the register. to give rules (e.g. . register("email", { required: "Email is required", pattern: {...} }))). However, it is much clearer to use **a validation schema** from a library like Yup or Zod . These libraries allow you to define the object schema of the entire form (fields and their rules) in one place, and then connect it to RHF using a so-called **resolver** (e.g. yupResolver or zodResolver from the @ hookform / resolvers package).

Yup is a popular library for building validation schemes in JavaScript . It allows you to declaratively describe data types and rules (e.g., the email field must be a required string , matching the email format, with a custom error message). Inspired by the Joi library, Yup is *lightweight and optimized for client-side validation* . **Zod** is a newer library, gaining popularity especially in TypeScript projects . It is *TypeScript-first* , meaning it can generate TS types from schema definitions and vice versa, ensuring better integration of types with validation logic. Zod is sometimes preferred in modern TS projects, while Yup is equally widely used in JavaScript/ React environments due to its maturity and simplicity. In practice, both fulfill a similar role – providing declarative, reusable form validation rules.

RHF integration with Yup / Zod : we use a *resolver* that connects the validation schema with the form engine. For example :

```
import { useForm } from "react-hook-form" ;
import { yupResolver } from "@hookform / resolvers/yup" ;
import * as yup from "yup" ;

const schema = yup.object ( {
  email : yup. string (). email ( " Enter valid email" ).required( "Email is required " ),
  password : yup. string (). min ( 6 , "Min 6 characters " ). required( " Password is required " )
}).required();

const { register, handleSubmit , formState : { errors } } = useForm ({
  resolver : yupResolver ( schema )
});
```

Here, we defined a schema using Yup and passed it to useForm via the resolver . This way, **upon submission,** RHF will automatically validate the data against the Yup schema . If a field fails validation, the errors object will be populated with appropriate messages (e.g., errors.password.message will contain "Min 6 characters" if the password is too short). This approach

centralizes the validation logic – the rules are defined in one place (the Yup / Zod schema), and the form component only displays any error messages. **The advantages of using Yup / Zod with RHF** include: maintaining the purity of the component (validation is not distributed throughout the JSX code), the ability to reuse the same schema, e.g., for server-side validation, and readability (the schema is read almost like a field specification). Yup offers a wealth of features: type checking, chained validators (min, max, regex , etc.), conditional validations dependent on other fields, and custom error messages. Zod is similar, and also works great with TS.

React Hook Form + Yup / Zod combine to create a powerful, **efficient, and concise** form solution. This saves you from having to manually write tons of error handling and conditional checking. Combining Yup with React Hook Form allows you to build **efficient forms with minimal code, providing users with immediate feedback on incorrectly filled fields** . This is because RHF takes care of optimization (including minimal re -rendering of form components) and automatically invokes schema validation only when needed. As a result, developers focus on application logic rather than low-level form handling.

Displaying errors to the user : RHF makes this simple – as mentioned, simply check if errors [fieldName] exists and display something like { errors.fieldName ?. message } . These messages come either from the schema definition (e.g., the message from Yup) or from manual rules. It's crucial that the form provides clear information, e.g., red borders around error fields, messages like "Field Required," etc.

Transferring data between pages (state in useNavigate)

Often, during navigation, we want to pass certain data from one page to another. For example, a user fills out a form, and after submitting , we want to redirect him to another page and pass, for example, information about the newly created object or a success message in the state. React Router provides a mechanism **for passing state in navigation** , which can be used both with the <Link> component and the navigate function (from the Hook useNavigate).

How does it work? Both the <Link> component and the navigate () function accept an optional state parameter , which can contain any serializable object (e.g., { message : " Success " }). This state will be included in the navigation history but **will not be visible in the URL** . For example, using:

```
< Link to= "/ dashboard " state={{ user : username } }> Go to dashboard</ Link >
```

or equivalently imperatively:

```
navigate ( "/ dashboard " , { state : { user : username } });
```

will cause the component handling the route / dashboard to have access to the { user : username } object via hook useLocation . Specifically, at the destination we call:

```
import { useLocation } from " react-router-dom " ;
...
```

```
const location = useLocation ();
console . log ( location.state ) ;
```

location.state contains the passed object (if we reached this page via the above mechanism). This way, we can pass, for example, information like " *User John logged in successfully* " and use it on the target page (e.g., display a welcome message).

Note that the passed state **must be serializable** (React Router saves it in the session history). This means: it can be a primitive type, a JS object, a string , a number, etc. – but not a function or a class instance, as these will not be recreated correctly. Attempting to pass, for example, a function will result in an error or its loss. In particular, **you cannot pass React components in state** (it's better to use, for example, the Context API). A simple object or array – absolutely.

We showed how to read this state above using useLocation ().state . We can also use TypeScript to type this, but conceptually, the important thing is that **the data in state is volatile** . It's important to remember that navigation state is stored in the browser's session history (History API). This means it survives a page refresh (F5), but is lost if the user opens a link in a new tab or clears the history . Therefore, this mechanism is suitable for passing temporary information (e.g., "Toast: operation completed successfully" or an object we don't need after refresh). If the data must survive a page reload or be available regardless of navigation, it's better to use, for example, a parameter in the URL (which will remain in the address) or a global store (Context / Redux) or LocalStorage .

Despite this transience, passing state between pages is very convenient. This allows us to avoid global variables or artificially inserting data into queries. params . Example use-case : after logging in, we want to pass the username to the dashboard – instead of keeping it in the URL, we can do navigate ("/ dashboard ", { state: { user : "Jan" } }) . On the dashboard page , we check location.state.user and, for example, greet the user by name. Another example is navigating from a list to a details page with information about which tab to open or which element to highlight – this can also be passed via state (as long as it's not critical after refresh).

To summarize: **React Router allows you to pass your own data during navigation** – both declaratively via <Link state={...}> and imperatively via navigate (..., { state }) . On the target page, we retrieve it using useLocation () from the react-router-dom package (location.state property). This is useful for communicating between views without the need for global state, but there are some limitations to keep in mind (state disappears after refresh, it must be serializable).

View authorization – private routes (ProtectedRoute)

Many applications require that access to certain subpages be restricted to logged-in users only. This is achieved using so-called **private routes** , i.e. private routes, also called protected routes routes). **Protected A route** is a concept that checks whether a user is authorized to view a given page (e.g., authenticated) before rendering it. If so, we display the page; if not, we redirect the user, for example, to a login page.

React Router implementation is typically done by wrapping protected paths in a special component that performs this check. This can be done in several ways. One way is to create a < ProtectedRoute > component that accepts a child component(s) and use it in the route definition: e.g., < Route path ="/ dashboard " element={< ProtectedRoute><Dashboard /></ ProtectedRoute >} /> . Such a ProtectedRoute component inside can use, for example, the authentication context to check the user's state.

First, however, you need to have **a source of truth about your login status** . This is often achieved using **the Context API.** React or a state manager. For example, we can create an AuthContext that stores information about the logged-in user, as well as login() and logout () functions . This context encompasses the entire application (a provider placed high up, e.g., in the App or router). Login() sets the user object's state (and optionally in localStorage to survive refresh), while logout () clears the user's state. In the login code, after successfully verifying the password, we call login(userData) – this will mark us as logged in and, for example, redirect us (useNavigate can be used within login() to navigate to, for example, the / dashboard).

Given the context, our **ProtectedRoute component** can do something like this: get the current user from the context (const { user } = useAuth ();) and if the user is *n't* logged in (! user), instead of returning children, it returns a < Navigate to="/login" replace /> component – which immediately redirects the browser to the login screen. Otherwise (if the user exists), it returns children, i.e., the actual protected subpage . Thanks to this, any route wrapped in ProtectedRoute will automatically be inaccessible without logging in. This mechanism is based on the < Navigate > component from React Router – a component that renders nothing but the effect of redirecting to the specified URL (similar to the navigate () call , but used in JSX).

It is worth emphasizing that **protected Routes are simply a concept for restricting access, implemented manually** – React Router doesn't have any built-in "mark this route as private" magic beyond using common constructs. There used to be examples in React Router v5 where you could compose < Route > and check the condition in the prop. render , in v6 due to the simplification of < Routes > this approach has changed – hence the pattern with the wrapper component . We can also use it differently: for example, conditionally render various elements in the route definition (but with ProtectedRoute the component is cleaner).

In practice, our ProtectedRoute can be very simple, as described above, or extended – for example, it can display the message "Verification in progress..." while authorization status is still being determined (e.g., a JWT token is being validated), or it can support *roles* (checking whether the user has the appropriate role/level of permissions for a given resource). In its simplest form, however, it's a simple if / else : **if no authorization => redirect** . This prevents unauthorized users from even briefly seeing the target page's content.

When implementing private routes, remember to:

- Security also from the backend side (routing React is only in the frontend layer – it should be supplemented with access control on the API, otherwise a clever user could retrieve data via the API anyway, bypassing the frontend).
- Providing the possibility of redirecting back: e.g. the user wanted to enter a more deeply protected page - after logging in, he can be sent back to where he wanted (React Router Navigate can pass in state, e.g. the target address before redirection , or use the query parameter redirect =).

For the purposes of this lecture, it is worth remembering that **private Routes** are a mechanism for restricting access to **pages only to authorized users** (e.g., logged in users). This is typically achieved through a ProtectedRoute component that checks authentication status and uses <Navigate> to redirect unauthorized users to /login . We wrap these components in the route definition, and a context or other global state stores information about whether the user is logged in. This provides a simple way to authorize views on the frontend .

Example: Routing and Login Form in Practice

To illustrate the above concepts, consider a simple application with several pages and a login mechanism. Our routes are: **/login** , **/ dashboard** , **/profile/:id** . Assumption: The Login page is publicly accessible, while the Dashboard and Profile/:id are private (requiring login). Upon entering the login page, the user fills out a form and is redirected to the dashboard . From the dashboard , they can navigate to their profile (e.g., by clicking on their name), which loads the profile page with a dynamic ID parameter.

Project Structure: In the pages / folder, we have the following files: LoginPage.jsx , DashboardPage.jsx , and ProfilePage.jsx . In layouts /, let's place, for example, MainLayout.jsx , which contains the header with navigation. In components /, we could have, for example, ProtectedRoute.jsx and minor UI components (buttons, form fields, but let's simplify this – we'll create the form without field extraction).

Route definition: We are using React Router DOM v6.4+, so in the main router file (e.g. router.jsx) we create a router:

```
import { createBrowserRouter , RouterProvider } from "react-router- dom " ;
import MainLayout from " ./layouts/MainLayout " ;
import ProtectedRoute from " ./components/ProtectedRoute " ;
import LoginPage from " ./pages/LoginPage " ;
import DashboardPage from " ./pages/DashboardPage " ;
import ProfilePage from " ./pages/ProfilePage " ;

const router = createBrowserRouter ([
{
   element : < MainLayout />, // common layout
   path : "/" ,
   children : [
{ index : true , element : < h2 >Welcome page </ h2 > }, // optional side home
```

```
{ path : "login" , element : < LoginPage /> },
{
    path : "dashboard" ,
    element : < ProtectedRoute > < DashboardPage /> </ ProtectedRoute >
},
{
    path : "profile/:id" ,
    element : < ProtectedRoute > < ProfilePage /> </ ProtectedRoute >
}
]
}
]);

// ... in rendering application :
< RouterProvider router = {router} />
```

In the above configuration , /login is outside the ProtectedRoute (i.e., accessible to everyone), while / dashboard and /profile/:id are wrapped in <ProtectedRoute> . We assumed that the ProtectedRoute component itself checks the authentication context. If the user isn't logged in, it will < Navigate to="/login" replace /> , and if the user is present , it will return children (i.e. , DashboardPage or ProfilePage).

Login Form: Let's implement a form with "login" and "password" fields in LoginPage . We'll use **react-hook-form** for handling and **Yup** for validation to demonstrate what we've been discussing:

- We set the schema Yup : e.g. login required (min. 3 characters), password required.
- We initialize RHF: const { register, handleSubmit , formState : { errors } } = useForm ({ resolver: yupResolver (schema) }); .
- At JSX we build <form onSubmit ={ handleSubmit (onSubmit)}> , and in it <input {...register("username")} /> , <input type="password" {...register("password")} /> .
- For each field we add the display of an error if it occurs: e.g. { errors.username && { errors.username.message } } .
- button : <button type="submit"> Log in </button> .

Submit handling : The onSubmit (data) function will receive an object with the username and password fields only if they pass schema validation . For demonstration purposes, we can simply compare whether the data matches any assumptions (e.g., if (data.username === " admin " && data.password === "123")). If so, we call our authentication context: login(data) . As mentioned earlier, login() in our context sets, for example, user in state and executes navigate ("/ dashboard ") . Alternatively, we can call navigate ("/ dashboard ", { state: { user : data.username } }) in onSubmit – but it's better to keep the navigation logic in one place, so let's assume that login() handles this. In case of invalid data, we can set, for example, an error message in the local state and display it. However, Yup validation will already catch, for example, a missing password or a too short login.

Navigation after login: When the user successfully submits the form, login() will be executed, which will redirect him to **the / dashboard** . This way, the user now sees the DashboardPage . This

component can display some general information (e.g., "Hello, { username }!"), for which it needs to know the username. There are two possibilities: we passed it via navigate in state (e.g., state: { user : " admin " }), then in DashboardPage we can get the const location = useLocation (); const { user } = location.state || {}; and use it. However, if we're using Context for authentication state, it's more convenient to simply retrieve the current user from the context. Let's assume that useAuth () returns a user object , e.g., with a username field , then in the DashboardPage we write: const { user } = useAuth (); and we have access. This demonstrates another method of passing data between pages – through a global context – which is persistent and doesn't disappear on refresh (if the context is based on localStorage , for example).

Dashboard and Profiles as private routes: Thanks to ProtectedRoute, an unlogged user can't manually access / dashboard or /profile/123 . If they tried, they would be immediately redirected to /login . After logging in, when they land on the dashboard , they can, for example, click "My profile." The link in the dashboard might look like: <Link to={ /profile/${ user.id } }>My profile</Link> . We navigate to /profile/7 (for example) – this route is also wrapped in ProtectedRoute , but since we're logged in, it will pass. ProfilePage calls const { id } = useParams (); and, for example, retrieves details of a user with that ID from the API or – if the data is already in the context – displays them immediately. For simplicity, let's assume that the id is only used to display or retrieve from some list of users . ProfilePage can show, for example: "User profile {id}" as a header, and below it, e.g. email, registration date, etc. (in a real application, we retrieve this data, e.g. after loading the component, using id).

Logout: There will likely be a "Logout" button on your dashboard or profile. Managing it is as simple as calling logout () from the context. In our context, logout () clears the user and executes navigate ("/", { replace : true }) or /login . Using replace : true will prevent the user from "going back" to the protected page from their browser history after logging out (we're replacing the history entry). After logging out, because the user state is null , ProtectedRoute will redirect to login everywhere , protecting our pages.

Example summary: In the above application we saw in practice: routing definition with layout and nested routes (MainLayout with Outlet for dashboard /profile), dynamic route (profile/:id), use of useParams to retrieve the ID parameter, use of useNavigate for redirection after logging in, passing user data via context (or possibly via navigation state, which we also illustrated as an option), validation of the login form using React Hook Form + Yup (forcing correct data before logging in) and a private route mechanism (ProtectedRoute) that protects the dashboard and profile from unauthorized access. This set of functionalities forms the basis of many modern SPA applications: routing allows the application to be divided into views, forms enable interaction and data collection from the user, and navigation and authorization ensure proper user guidance throughout the application (from login, through various screens, to eventual logout).

Of course, in more complex projects there are additional issues, such as lazy module loading (React Router allows delayed loading of route components, which improves performance), handling routing errors (paths not found – so-called route * with 404 component), passing query

parameters (query string) via useSearchParams , or e.g. role-based access (where ProtectedRoute checks not only whether the user is logged in, but also whether they have the admin role , for example). However, the foundation is understanding the concepts presented in this lecture. Armed with this knowledge, we can create React applications with multiple views, forms, and smooth navigation that are both user-friendly and well-organized from the code perspective.

Literature

1. https://react.dev/ (Access date: 1/10/2025)
2. https://vitejs.dev/guide/ (Access date: 1/10/2025)
3. https://www.typescriptlang.org/docs/ (Access date: 1/10/2025)
4. https://nextjs.org/docs/getting-started/react-essentials (Access date: 1/10/2025) - Next.js documentation that greatly expands on rendering topics (SSR, SSG, ISR).
5. https://tailwindcss.com/docs/installation (Access date: 1/10/2025)
6. https://ant.design/docs/react/getting-started (Access date: 1/10/2025)
7. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (Accessed: 1/10/2025) - MDN documentation for ES modules (import/export).