POLITECHNIKA ŚWIĘTOKRZYSKA

Advanced frontend applications – lecture 2

State management and communication within the application

mgr inż. Mateusz Pawełkiewicz

1.10.2025

Lecture Objective: To demonstrate how a React application manages local and remote data – from component state, through global state, to retrieving data from the server.

Status levels: local, global, remote

Local state is data maintained within a single component. It's used to handle UI within the component—for example, controlling form field values or modal visibility . Such state is typically managed via Hooks. React (useState , useReducer) and is not shared with other components. In modern applications, a lot of data can remain local – there's no need to put everything in a global store (a common mistake from the days of excessive Redux use) .

Global (shared) state is data shared between different, not necessarily closely related, components. Examples include information about a logged-in user, a shopping cart in an e-commerce application, or general application settings (e.g., a color theme). In React , global state can be passed through props (so-called *drilling*) or, more conveniently, using the Context API, which allows you to " *broadcast* " data to all child components without passing it through each level of the tree . Context solves the problem of cumbersome props passing through multiple component layers – for example, when many nested components need the same information. However, it should be used with caution; with very extensive global state, it is better to reach for external libraries, as Context can cause frequent rebuilds of many components at once. The rule of thumb is: first, try passing data through props , for larger scales, use Context , and only if that is not enough – consider a dedicated state management library .

Remote state (server state) is data coming from an external source – database, REST/ GraphQL API , etc. It is characterized by asynchronous nature and the fact that it is maintained outside the application (e.g., on the server) . Managing remote state is complex because, in addition to the value itself, we must handle **loading** (waiting for a response), **network errors** , **refreshing** (data may change on the server), and **sharing** this data between components. The traditional approach was based on useEffect + fetch () / axios and custom state variables to track these aspects, which led to a large amount of repetitive, error-prone code. Nowadays, it is recommended to use specialized libraries for handling remote state (e.g., **React). Query** or **SWR**), which solve the problems of caching , retrying, and refreshing data much better than manual solutions. According to the latest practices, *"splitting state into different categories provides better solutions than a single universal library"* - maintain local state in React , share it with simple techniques or Context , and entrust remote state to dedicated tools.

React Context API – architecture , Provider, Consumer

The Context API allows you to pass global data within a React application without having to pass props through the component tree . It consists of several elements:

- **Context** – Created using React.createContext (defaultValue) . Specifies a "channel" for passing data in the component tree.

- **Provider** – a component that provides a context value. We wrap it around a part of the React tree ; all child components have access to the provided value. Provider accepts a prop value with the value being provided. Providers can be nested , overriding values in lower branches of the tree.
- **Consumer** – a component that receives a value from the nearest Provider . In the older version of Context API, Consumer required the use of render prop (a function as a child of the component) to retrieve the context value. Since React 16.8, it is easier to use the hook useContext (MyContext) , which returns the current value . **Note:** In modern code, useContext is used instead of < MyContext.Consumer > , which is more readable and recommended .

The Context API architecture works by placing a Provider somewhere high in the application (e.g., in the main component or a specific module) with a certain value (e.g., the currently logged-in user, the application theme). All child components can retrieve this value using useContext —regardless of the nesting level. This provides great convenience when **sharing global data** , such as language preferences, color theme, or user data . There's no need to pass these props down through each intermediate component (which used to be called " props "). drilling ") . For example, instead of passing prop currentUser through subsequent levels, we can create a UserContext and read it in any component in the tree, simplifying the code.

However, please note that **overusing Context** can hinder optimizations. Changing the value in Provider causes re-rendering. **all** components using context. For large applications with a lot of frequently changing data, it is better to divide contexts into smaller scopes or – if this is not enough – reach for lightweight global state libraries (e.g. Zustand , Jotai). Nevertheless, the Context API is perfect for passing configuration, topics, or individual global state objects within a moderate scope.

Managing HTTP Requests – React Query (TanStack Query)

React Query (renamed **TanStack from version 4) Query**) is a library for managing *remote state* , i.e., data retrieved asynchronously from servers. Its main goal is **to simplify communication with APIs** and eliminate repetitive code for handling fetches , errors, and loading. Unlike Redux or Context , React Query **does not deal with the global UI state** – it focuses only on asynchronous data that needs to be downloaded or sent to the backend .

Key Features of React Query includes:

- **Declarative data fetching:** Using a hook useQuery defines a query by providing a unique key and a getter function. React Query itself handles calling this function, storing the result, and emitting statuses (loading/success/error). Example: const { data, isLoading , isError } = useQuery ([' posts '], fetchPosts) – the library will return data (if available), and booleans isLoading and isError reflecting the status of the query.
- **Automatic caching :** Once fetched, results are cached by default. If the component queries the same key again (e.g., the user returns to the list tab), React A query can

immediately return data from the cache instead of re-querying the server. This provides a faster and more efficient user experience. By default, the cache stores data for a specified period of time (stale time), after which it is marked as "stale."

- **Background data refresh:** The library can automatically *refetch* data, e.g. when the component is shown again or the user returns to the application window (refetch on window) . focus). This allows the user to usually see up-to-date information without manually refreshing the page.
- **Deduplication and synchronization of multiple queries:** React Query ensures that the same query isn't sent twice. If two components simultaneously need the same data (the same query key), the second query will wait for the first query's result and use it. This avoids the so-called " waterfall " of multiple requests to the same endpoint . Furthermore, the library provides the useQueries mechanism for executing multiple queries in parallel, as well as easy cache invalidation — for example, after a mutation, we can refresh related queries to update the data across all queries .
- **Mutation (write) support:** Hook useMutation allows you to easily send data to the server (POST/PUT/DELETE) and automatically update the cache upon success. It also supports **optimistic updates** – immediate changes to data in the UI before receiving confirmation from the server, which improves application responsiveness .
- **Easy loading and error states:** Each query has defined states like isLoading , isError , isSuccess , etc., so we don't need to write our own logic for setting spinners or error messages – just use these values in the renderer . The library also supports retrying queries on error, with configurable intervals and number of attempts.
- **Global Configuration and DevTools :** We can set global options (e.g., default refresh time, cache strategies) via the QueryClient object . Additionally, React Query offers *Devtools* – a developer panel that displays all active queries, caches , and mutations, making debugging easier .

React Query has gained popularity because it significantly **reduces the amount of code needed to handle remote data** (we no longer have to write useEffect + useState + try / catch for each fetch) and solves common problems like caching and data synchronization. In short: it's *" React 's missing piece "* for communicating with the server – a specialized layer for managing state that we don't control (because it's external). It's worth emphasizing that React Query **does not replace** Context or Redux – it is not used to store, for example, UI state – but it works great with them, complementing the ecosystem with API data management.

Axios vs Fetch API – Differences and Interceptors

For HTTP communication in React applications, the built-in **Fetch API** or the popular **Axios library** are most commonly used . Both allow sending GET/POST requests, etc., but they differ in convenience and functionality . Below is a comparison of key features:

- **Availability: Fetch** is built into browsers (and, since Node.js 18, also available on the backend), so it doesn't require installation. **Axios** is an external library based on XMLHttpRequest that we install via npm .

- **Response syntax and processing:** Fetch returns a Response object and requires a manual call to response.json () to extract JSON data from the response. Axios automatically *parses* the JSON – in the response object , the data is immediately available under the response.data field. This often results in shorter Axios code – the data can be used immediately, without any additional processing.
- **Error handling:** By default, Fetch recognizes errors only in the case of network problems (e.g., connection failure). HTTP responses with error codes (e.g., 404, 500) **do not result in Promise rejection** – you need to check response.ok or status yourself and manually throw an error. Axios, on the other hand, automatically throws an exception (rejects the Promise) if the status is outside the 2xx range, which simplifies global error handling. In Axios , we can use .catch () for all errors (network and HTTP), whereas with Fetch, we have to handle statuses separately.
- **Interceptors (request/response interception):** This is one of the biggest advantages of Axios . **Interceptors** allow you to plug into the process of sending a request or receiving a response, for example, to add an authorization header to each request, log errors, or globally handle token refreshes . You can define functions that will execute before each request (e.g., axios.interceptors.request.use (config => { /*...*/; return config })) or after receiving the response (axios.interceptors.response.use). This allows you to centralize the logic common to all HTTP calls – once set up, interceptors automatically execute on every request . **It doesn't have built-in interceptors** . While you can achieve a similar effect by globally overriding the fetch function or writing your own call layer, it's additional work. For many developers, the lack of interceptors and cumbersome error handling are the main reasons for choosing Axios .
- **Cancelling inquiries:** Fetch supports cancellation via **AbortController** – you need to create a controller , pass a signal to fetch () , and call controller.abort () if necessary. Axios offered its own CancelToken mechanism , and in newer versions it also works with AbortController . In Axios, setting a timeout is simpler – you can specify a timeout in the configuration, and the library will automatically abort the query.
- **Browser support:** Axios also works in older browsers (e.g., IE11) using XHR. Fetch requires a polyfill for IE and is only available by default in modern browsers (although this is now standard). In practice, compatibility is no longer a major issue, as the environments are converged (Fetch even works on the backend).
- **Size and performance:** Fetch doesn't add any size to the app (it's built in). Axios is an additional library (~5kB min+gzip), which isn't usually a significant overhead for today's apps, but it's worth mentioning.

To summarize: For simple queries in small projects, Fetch is sufficient – it's built-in and works well. However, **Axios offers conveniences** like automatic JSON processing, better error trapping, and powerful **interceptors** , which allow, for example, globally adding tokens to headers or handling session refreshes. Many developers choose Axios precisely for these additional capabilities, avoiding writing repetitive code to handle each request individually. If necessary, you can also use both – for example, Axios for communicating with your own API (where interceptors are needed), and Fetch for occasional queries to other services.

Error and loading status handling

When communicating remotely, it's essential to ensure **good UX in intermediate states** —when data is loading or an error occurs. Users shouldn't be left without information, for example, by staring at a blank screen while data is being fetched . Therefore, components should render appropriate messages depending on the state:

- **Loading state :** Typically represented by an activity indicator—e.g., a spinner , progress bar, or a placeholder that imitates a data structure. We show it **immediately after initiating a query and keep it there until a response arrives. In practice, you can have the** isLoading state variable set to true before fetch and to false in finally (after completion, regardless of success or error). Using React Query this is simplified – hook useQuery returns us isLoading automatically.
- **Error status:** If the data retrieval fails (e.g., network error, server error), the user should be informed. Typically, an error message is displayed where the data should be, possibly with an option to retry the action (e.g., a "Try again" button). For example, you can show an alert component with the text "An error occurred. Please try again later." In React code, the easiest way is to check the isError flag (from React Query) or catch the exception with fetch () and set the error message in the component state, then conditionally render e.g. < ErrorMessage /> .
- **Empty state:** Often, handling states also includes situations where the query returned **successfully** , but there's no data (e.g., the list is empty). It's good practice to distinguish this from the loaded state—displaying, for example, a "No Results" message or other placeholder , instead of simply an empty list. In code, this is shown as the condition if (data? .length === 0) return < EmptyState /> .

Example approach in a React component (pseudo-code):

```
const { data, isLoading , isError } = useQuery ([ 'posts' ], fetchPosts );

if ( isError ) {
  return < Alert variant = "error" > An error occurred while loading data. </ Alert >;
}
if ( isLoading ) {
  return < Spinner />; // or skeleton UI
}
if ( data. length === 0 ) {
  return < p >No results to display. </ p >;
}
// Main content when we have data:
return < PostsList posts = {date} />;
```

This structure ensures that all scenarios are handled – first the error (it has priority, because if there is an error, there is often no point in showing the spinner or old content), then loading (only if there is no data yet), then the empty state, and finally the normal data rendering .

Best practice: It's a good idea to separate common UI elements for loading/error states into separate, reusable components (e.g., < Loader /> , < ErrorState />). This will help maintain consistency throughout the application – users will see a similar spinner or error message everywhere. In larger projects, you can even build a higher-level abstraction: for example, a higher-level component or hook that handles *loading / error logic* for any query, so as not to duplicate it in every function or component.

React Query also allows you to define refresh strategies: for example, set staleTime to treat data as fresh for a certain period of time and not show the spinner on every navigation (instead, you can show the previous data and refresh it in the background). This allows the application to react more smoothly to view changes (we avoid spinner flickering when data is cached).

In summary: **handling asynchronous states** is crucial to the user experience. An application should always signal when it's loading (so the user knows something is happening) and catch any errors (so the user knows something has gone wrong, instead of just seeing a blank page). Well-designed loading/error states make an application more predictable and user-friendly.

Synchronizing multiple queries and refreshing data

In a real-world application, we often execute **multiple HTTP requests** —some independent, sometimes dependent. It's crucial to ensure proper synchronization of these requests to ensure consistent and up-to-date data across the entire interface.

Parallel Queries: If different data can be retrieved independently, it's best to execute queries **concurrently** (competitively) rather than sequentially. In React , we can use multiple useQuery calls in a single component – React Query will manage the queuing itself. Alternatively, the standard Promise.all () in useEffect will run fetches in parallel. This way, we avoid unnecessary extensions of page load time (the so-called *waterfall). request*). Example: when a page requires downloading a product list and a category list, we can download them simultaneously because they are not mutually dependent.

Dependent queries: More often, one query **needs the result of another** —for example, we first fetch a list and then the details of the selected item, or fetch uses an ID from the previous response. In such cases, synchronization means executing the second query only after the first one succeeds. React Query offers an enabled option for the query – for example, we can call useQuery ([' details ', id], fetchDetails , { enabled : !!id }) , which will cause the details query to execute only when the id (e.g., the selected item) is available. Another approach is to place logically related fetches into a single useEffect (for manual fetches) or merge them in the backend (if we have such control over the API). It's crucial to react to changes – for example, when the user selects a different item, we should cancel or ignore the previous details query and send a new one for the new ID.

Data Refresh (Refetch): Remote state can be dynamic – data can change on the server. Therefore, it is necessary to refresh the cache periodically or on demand. React By default,

Query treats data as "old" immediately after fetching (unless we set staleTime), which results in refreshing each time we revisit the data screen. We can adjust this strategy: for example, set staleTime : 5 * 60 * 1000 (5 minutes) to avoid querying the server too often, or use refetchInterval to automatically query the backend periodically in the background. Furthermore, after performing a data-modifying operation (mutation), **we invalidate the cache** of related queries (e.g., queryClient.invalidateQueries ([' todos ']) after adding a new "to-do"), which will refresh the data and ensure the view is consistent. React Query solves many such problems out of the box – for example, it automatically **merges identical queries** (deduplication), so if two components request the same data, the library executes them only once and provides the result to both. It can also refresh the displayed data **in the background , without the need to show the** loading state again (so-called stale-while-revalidate).

Synchronization examples: Let's say we have a screen where we need to retrieve user data and their notification list simultaneously. We can do this in the component to do :

```
const userQuery = useQuery ([ 'user' , userId ], fetchUser );
const notificationsQuery = useQuery ([ 'notifications' , userId ], fetchNotifications );
```

Both of these functions will run in parallel. However, if notifications require user data (e.g., role) first, you can use:

```
const userQuery = useQuery ([ 'user' , userId ], fetchUser );
const notificationsQuery = useQuery ([ 'notifications' , userId ], fetchNotifications , {
  enabled : !! userQuery.data , // wait for user data
});
```

This way notificationsQuery will only start when userQuery has a result.

In case of very many queries at once, React Query also offers a useQueries method , which allows you to pass an array of queries to execute. This returns an array of results that can be queried collectively. However, often, calling multiple useQueries side by side is sufficient.

Sharing query results: Sometimes a single result is needed in multiple places (e.g., product details in different tabs). Instead of fetching multiple times, you can either: a) fetch data higher in the hierarchy and pass it down (prop drilling or context), or b) use **the cache** React Query – if a component below calls useQuery with the same key, it will retrieve the cached data (or refresh it if the timer expires) without any additional code. This is a huge advantage – we don't need to manually synchronize states across different locations; simply use the same query keys consistently for the same resources, and the library will take care of consistency (e.g., after a mutation, it will refresh all components using a given key).

In summary, **query synchronization** boils down to: executing independent requests in parallel for better performance, executing dependent requests sequentially (via conditional calls), and refreshing data to prevent it from becoming stale. Tools like React Queries simplify most of these tasks—from deduplication of multiple identical requests, to automatic refreshes, to easy

reloading of data after changes. This allows developers to focus on application logic rather than low-level asynchronous state management.

Typing API responses in TypeScript

Using TypeScript , we can ensure **API data is typed** , which increases security and simplifies workflow. When we receive JSON from the server, it is treated as an arbitrary object by default – with no guarantees regarding its structure. In TS, it's useful to define **interfaces or types** that describe the expected response structure. For example, for an API returning a list of posts:

```
interface Post {
  userId : number ;
  id : number ;
  title : string ;
  body : string ;
}
```

With such an interface, we can use TypeScript mechanisms for fetching :

- **Fetch API:** After receiving the response and calling response.json () , we can cast the result to our type, e.g. const posts = data as Post[] . Fetch doesn't know directly what type it returns, but by casting or declaring the variable as Post[] we can make sure that posts [0].title will be recognized as a string , etc., later in the code . Alternatively, you can use generics with your own fetching function , but TS doesn't infer the type from fetch itself (you have to manually ensure the type match).
- **Axios :** Axios provides generics in its methods – we can call axios.get<Post []>('/ posts ') . This will allow the response object to have response.data as a Post[] array (instead of any or unknown). This is convenient because the entire pipeline becomes typed. It's worth noting that there are libraries/types that extend Axios (e.g., the extended Axios type), but in practice, regular Axios with generics is sufficient in 90% of cases.
- **React Query :** Here the typing is also simple - if the fetchPosts function is annotated , e.g. function fetchPosts (): Promise<Post []> { ... } , then the result of useQuery ([' posts '], fetchPosts) will be inferred as Post[] . We can also explicitly pass generics to useQuery<Post []> to specify the data type. Then the data will be of type Post[] and TS will force us, for example, to check that the data is not undefined before using it (because it can be undefined as long as isLoading is true).

Benefits of typing API responses:

- We have **autocomplete** and typo detection for object fields. For example, if our Post has a title : string field , we'll immediately see the available fields and types in the IDE.
- TypeScript will catch if we try to use a field that isn't in the data, or if we confuse the type (e.g., treating a number as an array). Without types, such errors would only be displayed at runtime .

- **refactor** more easily – for example, changing the name of a field in the interface will allow us to find all the places in the code where it is used.
- Clear documentation for other developers – you can immediately see in the definition what the response structure looks like.

The disadvantage is that **TypeScript does not provide runtime data validation** – if the API returns something that doesn't meet the type, TS won't stop it (it only works at compile time). Therefore, when integrating with external APIs, additional validation is sometimes used (e.g., libraries like `zod` to validate the data shape). However, in most cases, especially when we control the API or use a well-known API, **the contract itself typing** is enough and helps significantly.

When working with REST APIs, it's also common to use tools that generate types from API definitions (e.g., OpenAPI / Swagger generators) – in large projects, this can save work and ensure compatibility with the backend . In our components, response typing allows us to use React Query or axios with the full power of TS: e.g. `useQuery<Post []>(...)` will make `isError` have the associated error type (default `unknown` or as specified) and `data` will be `Post[] | undefined` , which forces checking the existence of data before use.

In short: **typing API responses** adds an extra layer of security. With minimal effort (interface definition + use of generics), we gain confidence in the data structure of the entire application. This helps avoid many potential errors and inconsistencies between frontend and backend , and makes the code more readable (it's known that `posts : Post[]` is an array of Post objects, instead of `any`). This is especially important in enterprise applications where API contracts are complex – TypeScript can then catch inconsistencies before deployment.

Pattern: " Container + Presentational components "

React architecture often uses a division between container components and **presentational components . This design pattern** promotes the separation of logic from the presentation layer:

- **Presentation components** (sometimes called "child" or "dummy " *Components* : They are solely responsible for displaying data. They accept data and callbacks via props and render the UI – e.g., a table, list, or form, which don't know where the data comes from or what is being done with it. Features: They are usually pure functional components, they don't have their own global state (at most, local state for internal UI needs, e.g., controlling password visibility in a field) , they have no side effects. This makes them easy to test and reuse. Example: the `<PostList>` component accepts props `posts` and generates a list of elements based on it – it only deals with **how** to display the data.
- **Container components** (*smart components*): are responsible for providing data to presentation components. They can retrieve data from an API, subscribe to a store (Redux), handle user events, and update state. The container "knows what" to display and how to respond to interactions, and the presentation component "knows how" to display it. Containers often render nothing of their own except a nested presentation component, to which they pass data and functions as props . For example,

<PostListContainer> can use useQuery to retrieve a list of posts, store the ID of the selected post in its state , handle clicks (set state), and in return render < PostList posts ={data} onSelect ={ handleSelect } /> . It doesn't generate HTML itself (outside of the border), it just passes on the logic and data.

Advantages of this pattern: It allows for better separation of responsibilities. Presentation components are simple and can often be used in different parts of the application (because they don't depend on a specific data source). Container components aggregate logic – for example, a single container can use both Context and React. Query , and compose data for the presentation child. This allows for changing the way data is retrieved (e.g., switching from fetch to axios or from Redux to React Query) requires only the container to be modified – the presentation components remain untouched.

This pattern was especially popular with Redux (where the container is a component connected to the store via connect (mapStateToProps) and the presentational renderer). Nowadays, with the advent of Hooks , the line is blurring somewhat – we can both retrieve data with a Hook and display it in a single component. However, with more complex views, it's still worth considering this separation for readability. You can also achieve a similar effect by writing **your own Hooks** – for example, usePostsData as the data retrieval logic (this replaces the container), and the presentational < PostList > uses this Hook within itself. This Hook approach is an alternative to literally splitting the component into two components, but the idea of separation remains the same.

For example, in our context we can distinguish:

- PostsContainer – a container component that retrieves a list of posts (e.g. uses React Query), handles the selection of a specific post (e.g. manages the state of the selected ID in the Context API), and renders layout .
- PostsList – a presentation component that takes a list of posts and an onSelect function from props and displays a list of items (e.g., post titles as buttons). It doesn't care where the data came from, it just shows what it got.
- PostDetails – another presentational one that takes a post object (or its details) and displays e.g. the title and content.

This division makes it easier, for example, to test the presentation (we can provide artificial data to PostsList in tests and check the rendering without mocking fetchy) and reusing presentation components in other places (if, for example, we have another container with a different data source, we can still use PostsList to display).

Example: Fetching data from API and displaying list and details (React Query + Context API)

Finally, let's combine the concepts discussed in a practical example. We'll demonstrate a simple React application that retrieves a list of posts from a server and allows you to display the details

of a selected post . We'll use **React Query** for handling remote data and **Context API** for managing global state (select current post). Additionally, we'll use the container/presentation pattern for clarity.

Data Source: We will use the public JSONPlaceholder API , which provides test post data (each post has a userId , id , title , and body). Endpoint / posts returns a list of 100 posts, and / posts /{id} returns the details of a single post .

Implementation plan:

1. **Project Configuration:** We assume that the React project (e.g. created by Create React App or Vite) has the React library added Query (npm install @ tanstack / react-query). At the top level of the application (e.g. in src / main.jsx or index.jsx) we configure **QueryClient** and wrap the application in QueryClientProvider to React Query worked on the entire tree:

```
import { QueryClient , QueryClientProvider } from '@tanstack / react-query' ;
const queryClient = new QueryClient ();
const root = createRoot ( document . getElementById ( 'root' ));
root. render (
  < QueryClientProvider client = {queryClient} >
    < App />
  </ QueryClientProvider >
);
```

   Thanks to this, inside App and beyond we can use useQuery () .

2. **Context for global state:** We'll create a Context to store information about the currently selected post (its ID). This will help us share this selection with both the list (e.g., to highlight the selected item) and the details component. Example:

```
import { createContext , useState } from 'react' ;
export const SelectedPostContext = createContext ({
  selectedId : null ,
  setSelectedId : () => {}
});
export function SelectedPostProvider ( { children } ) {
  const [ selectedId , setSelectedId ] = useState ( null );
  return (
    < SelectedPostContext.Provider value = {{ selectedId , setSelectedId }}>
      { children }
    </ SelectedPostContext.Provider >
);
}
```

   This provider wrap around the part of the application that needs access to the selected ID. We can do this e.g. inside App :

   function App () {

```
  return (
    <SelectedPostProvider>
      < PostsContainer />
    </ SelectedPostProvider >
  );
  }
```

3. **Container component – PostsContainer :** This will be responsible for fetching data and integrating everything. We'll use React Query to get list of posts:

```
import { useQuery } from '@tanstack / react-query' ;
import axios from ' axios ' ;
import { useContext } from ' react ' ;
import { SelectedPostContext } from ' ./SelectedPostContext ' ;
import PostsList from ' ./PostsList ' ;
import Post Details from ' ./PostDetails ' ;

function PostsContainer () {
  const { selectedId , setSelectedId } = useContext ( SelectedPostContext );
  // Get the list of posts
  const { data : posts, isLoading , isError } = useQuery ([ 'posts' ], () =>
    axios. get ( 'https://jsonplaceholder.typicode.com/posts' ). then ( res => res. data )
);

  if ( isError ) return <div> Error charging posts ! </ div >;
  if ( isLoading ) return <div> Loading data ... </ div >;

  return (
    < div className = "posts-app " >
{/* List presentation component */}
     < PostsList posts = {posts} onSelect = {id => setSelectedId (id)} />
     {/* Details presentation component ( rendered when post selected) */}
{ selectedId && < PostDetails postId = { selectedId } />}
   </ div >
);
}
export default PostsContainer ;
```

Some clarifications: we're using axios.get here to fetch the data (for the reasons explained above, Axios will simplify the response parsing). useQuery with key [' posts '] will perform the fetch and return the array of posts as posts . We react to isLoading / isError states simply by displaying messages. If the data is loaded, we render layout with a list and optionally details. The onSelect function passed to PostsList sets the selected id in the context (which will result in rendering PostDetails).

4. **Presentation Component – PostsList :** Its purpose is to display a list of post titles and allow selection. Sample implementation:

```
function PostsList ( { posts , onSelect } ) {
  return (
```

```
   < ul className = "posts-list " >
{ posts.map (post => (
       < li
         key = { post.id }
         onClick = {() => onSelect ( post.id )}
style={{ cursor: 'pointer', margin: '5px 0' }}
       >
         < strong >{ post.title } </ strong >
       </ li >
))}
   </ ul >
);
}
export default PostsList ;
```

Each list item is clickable – when clicked, we call `onSelect` with `post.id` . The cursor style indicates that the item is interactive. You can also add a condition that highlights the currently selected post (if `selectedId` is also passed as a prop to know what's selected). For simplicity, I'm not doing that here, but you could, for example, compare `post.id === selectedId` and provide a different style.

5. **Presentation Component – `PostDetails` :** This component will display the details of the selected post . Since **we can already obtain the post data** from the list (list/ posts also returns the body), we could theoretically pass the entire post object to the details. However, details are often retrieved separately – let's say we also want to display a list of comments for the post (which weren't in the list). So, we'll create a separate query for the post details or additional data. We'll use the `postId` from props in the next `useQuery` :

```
function PostDetails ( { postId } ) {
  const { data : post, isLoading , isError } = useQuery (
    [ 'post' , postId ],
    () => axios. get ( `https://jsonplaceholder.typicode.com/posts/ ${postId} `). then ( res => res. data ),
{ enabled : !! postId } // execute only if postId is valid
);

  if ( isError ) return < div >Failed to load post details . </ div >;
  if ( isLoading || !post) return < div > Loading post... </ div >;

  return (
    < div className = "post-details " >
      < h2 >{ post.title } </ h2 >
      < p >{ post.body } </ p >
      < button onClick = {() => {/* possibly some action, e.g. close details */}}>
Close
      </ button >
    </ div >
);
}
export default PostDetails ;
```

This component also maintains its own states. We use the query key ['post', postId] – thanks to this, React The query will distinguish the cache for individual posts. If the detail has already been fetched, we'll retrieve it from the cache immediately. Otherwise, "Loading..." will be displayed. We'll also report the error.

6. **Integrating all the parts:** After these definitions, the entire application works like this: PostsContainer starts by getting a list of posts. When they arrive, it displays PostsList . The user clicks on the title, which triggers setSelectedId in the context. This causes re-rendering. PostsContainer (because it uses selectedId contextu) – now the condition { selectedId && < PostDetails ...>} starts to be true, so we render PostDetails , which calls its useQuery and retrieves the post content . The user sees the details. If he clicks on another post from the list, the selectedId changes, the PostDetails is updated to the new one (React The query will re-fetch the details for the new ID, but the old request will be automatically canceled if it was still in progress – React Query takes care of this). When selectedId is for example reset to null (e.g. user closes details), PostDetails does not render at all .

In the above example we saw **the interaction between Context API and React Query** : Context stores the state of the interface (selected element) and React Querying handles the state of data from the server. This approach is recommended—using tools appropriate for the state. As a result, the code is readable: the presentation components (PostsList , PostDetails) are simple and focused on rendering , while the container component (PostsContainer) orchestrates the logic (fetch + context) declaratively.

Extensions: This application can be easily extended, for example by adding details caching (React Query already does this – if we close the details and select the same post again, the data will appear immediately from the cache . You can also add a prefetch – on hover over the post title , call queryClient.prefetchQuery (['post', id], ...) to fetch the details in advance before the user clicks (this will create the impression of instant loading). React Query gives us such advanced capabilities almost instantly.

Summary

In this lecture, we covered various aspects of state management and communication in a React application :

- We distinguished between **local, global and remote state** and indicated which tools are most effective for maintaining each of them (from native hooks , through Context , to libraries like React Query).
- We presented **the Context API** as a way to achieve global state without prop drilling – with the Provider /Consumer mechanism – and we emphasized the importance of moderate use of context .

- We learned about **React Query (TanStack Query)** as a modern solution for fetching and caching data from the server, which simplifies handling HTTP queries, loading states, errors and data synchronization in the background.
- We compared **Axios vs Fetch** – paying attention to practical differences: JSON handling, interceptors , error handling, and ease of use.
- We discussed **loading and error states** – why handling them is crucial for UX and how to do it idiomatically in React (conditional rendering , user messages).
- We dealt with **synchronizing multiple queries** and refreshing data – i.e. how to execute requests in parallel or sequentially, when to use the cache , how to avoid double queries and keep data up to date (invalidation , refetch).
- We mentioned **response typing in TypeScript** to ensure type safety and better integration with tools (generics in axios / React Query) when working with the API.
- Finally, we applied all these elements in **a sample mini-app** – combining **Context** (UI state) and **React Query** (remote state) and separating components into container and presentation components for readability.

Literature:

1. https://react.dev/learn/managing-state (Accessed: 1/10/2025) - Official React documentation on different approaches to state management.
2. https://react.dev/learn/passing-data-deeply-with-context (Accessed: 1/10/2025) - Official React documentation for the Context API.
3. https://tanstack.com/query/latest/docs/react/overview (Accessed: 1/10/2025) - Official TanStack documentation Query (React Query).
4. https://axios-http.com/docs/intro (Accessed: 1/10/2025) - Official documentation of the Axios library .
5. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (Accessed: 1/10/2025) - MDN documentation on the built-in Fetch API.
6. https://zod.dev/ (Access date: 1/10/2025) - Documentation of the Zod library , used for data validation (mentioned in the lecture as an extension of TS typing).