POLITECHNIKA ŚWIĘTOKRZYSKA

Advanced frontend applications – lecture 4

Optimization, architecture and best practices

mgr inż. Mateusz Pawełkiewicz

1.10.2025

1. Introduction

In this lecture, we'll discuss what distinguishes a programmer who writes "working" code from an engineer who creates "scalable" systems. We're at a turning point for front-end technologies . The release of React 19 and the stabilization of new standards in tools like Vite , Vitest , and SonarQube have forced a redefinition of the concept of "best practices . "

Modern software engineering in the context of React is no longer just about knowing the syntax of JSX or hooks . It focuses on **an architecture** that can withstand the pressure of hundreds of new features; on **optimization** that is built into the compilation process (thanks to React Compiler) rather than enforced by hand-coded code; and on **quality** that is automatically verified by advanced CI/CD chains.

In this talk, we'll deconstruct the myth that scaling applications is solely a backend problem . We'll understand how poor architectural decisions on the frontend lead to technical debt that paralyzes development teams. We'll analyze how a modern approach to folder structure (Feature-Sliced Design), state management, and error handling creates a solid foundation for enterprise -class applications .

1.1 Definition of Frontend Scalability

Before we get into the technical details, we need to define what scalability means in the context of frontend . It's not about the number of queries per second (which is the domain of infrastructure), but rather three vectors:

1. **Code Scalability (Maintainability):** The ability of the codebase to accommodate new functionality without regression in existing modules. Does adding a new section in the user panel require modifying files in five different folders? If so, the architecture is unscalable .
2. **Team Scalability (Collaboration):** Multiple developers can work on the same project in parallel without constant conflicts in the version control system (git merge) conflicts) and without having to understand the entire system to change one button.
3. **Performance Scalability:** The ability of an application to maintain smooth performance and low load times (LCP - Largest Contentful Paint) despite the increasing amount of data downloaded and the complexity of the component tree.

2. System Architecture: From Chaos to Feature-Sliced Design

File organization has been a hotly debated topic for years. In the early stages of learning React (around 2018-2020), file type-based organization dominated. In 2025, this model is considered an anti-pattern in large systems.

2.1 Evolution of Design Structures

Historically, most React projects started with a simple "File Type First" structure:

src /

├ — components/ # All UI components (Button, Header, UserCard , ProductRow)

├ — hooks/ # All hooks (useAuth , useFetch , useScroll)

├ — context / # All contexts

└— utils / # All helper functions

This structure, while intuitive at first, leads to the phenomenon of "code hopping." To modify the functionality of the "Shopping Cart," the developer must open a components file , a hooks file , a utils file, and a styles file , which are scattered throughout the directory tree. In 2025, this is being abandoned in favor of **colocation** , or keeping things that change together, close together.

2.2 Feature-Based Architecture and Feature-Sliced Design (FSD)

The current industry standard is Feature - Based Architecture (FSD), or its more rigorous variant, Feature-Sliced Design (FSD). This approach treats an application as a set of independent business domains.

Bulletproof Structure React " (Domain)

In this model, the main code division is in the features / directory . Each folder within features / represents a specific business area of the application.

Recommended structure for 2025:

src /

├ — components/ # Components shared (UI Kit: Button, Modal, Input)

├ — hooks / # Global hooks (not domain-specific)

├ — lib / # Library configuration (Axios , React Query , i18n)

├ — features / # The heart of the application - business domains

| ├ — auth / # Everything related to authorization

| | ├── api / # Auth -specific API queries (login, register)

| | ├── components / # Components used ONLY in auth (LoginForm , RegisterButton)

| | ├── hooks / # Auth -specific hooks (useUser , useLogin)

| | ├── types / # TypeScript types for auth

| | └── index.ts # Public API of the module (Barrel file)

| ├── products/

| └── cart /

└── routes / # Routing definitions and lazy page loading

The main advantage of this approach is **modularity** . If we decide to discontinue the "Comments" functionality in the future, we simply delete the src / features / comments folder , ensuring that we don't leave dead code in the components or hooks folders .

2.2.2 Analysis Layers in Feature-Sliced Design

For very large teams (over 10-15 developers), full FSD is used, which introduces a strict hierarchy of layers. The principle is simple: a "higher" layer can only import from "lower" layers, never the other way around. This prevents cyclical dependencies (circular dependencies).

Layer hierarchy (from highest to lowest):

1. **App :** Initialization, global styles, providers (Redux , Theme , Router). This is where we put the application together.
2. **Pages :** Composition of views for specific routing routes . The page should be "thin" – mainly arranging widgets and features .
3. **Widgets :** Standalone UI blocks that connect data to a view (e.g., Header , ProductListWithFilters) . combine *Features* and *Entities* .
4. **Features :** Specific user interactions that deliver business value (e.g., AuthByEmail , AddToBasket , LikePost). This is where the "action" happens.
5. **Entities :** Business models and data visualization (e.g., User , Product , Order). Here we define what the product card looks like, but without the "Add to Cart" button (because it's a Feature).
6. **Shared :** Generic code , not related to business (UI Kit, helpers , API client).

The table below shows the differences in component responsibilities depending on their location in the structure:

Component Type	Location	Single Responsibility	Access to State	Examples
UI Component	src / components /	Clean presentation, no business logic. Receives props , renders HTML.	None (props only)	Button , Input , Card , Modal
Feature Comp.	src / features /*/	Implementing a specific business task. Combines UI with logic.	Local State, API	LoginForm , ProductFilters
Entity Comp.	src / features /*/	Displaying domain data.	Read-only props	UserAvatar , ProductPrice
Page Comp.	src / pages /	Layout for a given URL path.	Routing , URL params	DashboardPage , SettingsPage

2.3 Barrel Files and Dependency Management

A key element of the architecture in 2025 is the use of index.ts files (so-called Barrel Files) to encapsulate modules. The features / auth module should only expose what is necessary for the rest of the application (e.g., LoginForm , useUser). Internal helpers or sub-components should not be imported directly by other modules.

Example src /features/auth/ index.ts :

```
// Public API of the Auth module
export { LoginForm } from ' ./components/LoginForm ' ;
export { useAuth } from ' ./hooks/useAuth ' ;
export * from ' ./types ' ;
// We don't export e.g. internal password formatting
```

This way, by changing the internal structure of the auth folder , we don't break imports across the entire application, as long as the public contract (exports from index.ts) remains unchanged. This is an implementation of the **Open-Closed principle Principle** at the architectural level.

3. Performance Optimization: The React Era Compiler

In 2025, the way we think about performance in React has changed dramatically. For years (from React 16.8 to 18), developers had to manually manage object and function references to avoid unnecessary re -renders . This required the useMemo , useCallback , and React.memo , which often led to " memoization hell " and dependency array bugs.

3.1 React Compiler : Memoization Automation

React 19 introduced a stable version **of React Compiler** (formerly known as React Forget). It is a build tool that automatically analyzes component code and inserts appropriate memoization mechanisms at the compile level, rather than at runtime .

3.1.1 How React works Compiler ?

The compiler analyzes the data flow within a component. If it detects that the result of a function or JSX element depends on values that haven't changed, it automatically caches that result.

Before React 19 (Manual optimization):

```
const ExpensiveComponent = React.memo ( ( { data, onClick } ) => {
  const processedData = useMemo ( () => expensiveCalculation (date), [date]);

  const handleClick = useCallback ( () => {
    onClick (processedData.id);
},);

  return < Child data = { processedData } onClick = { handleClick } /> ;
});
```

In React 19 (With React Compiler):

```
// We don't need useMemo , useCallback or React.memo
const ExpensiveComponent = ( { data, onClick } ) => {
  const processedData = expensiveCalculation (data); // Compiler will meme it automatically

  const handleClick = () => {
    onClick (processedData.id);
}; // Compiler will stabilize reference this one function

  return < Child data = { processedData } onClick = { handleClick } /> ; // Compiler will memoize JSX call
};
```

The advantage is twofold: the code is cleaner (no cognitive overhead associated with `useMemo`) and safer (no errors resulting from omitting dependencies in the array).

3.1.2 When should I still use `useMemo` ?

Despite the existence of the compiler, `useMemo` hasn't completely disappeared. In 2025, we still use it in specific cases:

1. **Integration with external libraries:** If the library expects a stable reference and we are not sure whether the Compiler will handle the given pattern (although it does in 99% of cases).
2. **Explicit Memoization :** When we want to explicitly communicate to other programmers that a given piece of code is computationally very expensive and should be treated with special care.

3.2 Code Splitting and Lazy Loading

While React Compiler optimizes the *rendering process* (updating the view), **Code Splitting** (code splitting) optimizes the application *loading process* (Initial Load). In large SPA (Single Page Application) applications, downloading all JS code at startup is unacceptable.

Route-Based Modularization Strategy Splitting)

The most efficient method is to lazily load entire views based on routing . We do this using the lazy function (which gained better support for Server Components in React 19 , but works similarly on the client) and the Suspense component .

Example: Adding Lazy Loading to Routing

Let's say we have an application with a heavy admin panel. We don't want regular users to download the admin panel code .

```
import { Suspense, lazy } from 'react' ;
import { BrowserRouter , Routes, Route } from 'react-router- dom ' ;
import { LoadingSpinner } from ' ./components/ui/LoadingSpinner ' ;

// 1. Instead of static import :
// import AdminDashboard from './features/admin/Dashboard';

// 2. We use dynamic import from lazy :
const AdminDashboard = lazy( () => import ( './features/admin/Dashboard' ));
const UserProfile = lazy( () => import ( './features/profile/UserProfile ' ) );

export const AppRoutes = () => {
  return (
    < Browser Router >
      {/* 3. Suspense catches the moment of loading JS chunks */}
      < Suspense fallback = { < LoadingSpinner label = " Loading application ..." /> }>
        < Routes >
          < Route path = "/" element = { < HomePage /> } />
          < Route path = "/profile" element = { < UserProfile /> } />
          < Route path = "/admin" element = { < AdminDashboard /> } />
        </ Routes >
      </ Suspense >
    </ BrowserRouter >
);
```

```
};
```

In this model, webpack (or Vite) will separate .js files (so -called chunks) for the /profile and / admin paths . These will be downloaded by the browser only when the link is clicked.

3.2.2 Lazy Loading of Components (Component-Based) Splitting)

Lazy loading should be used not only for pages, but also for heavy interactive components that are not immediately visible (e.g. modals , rich text editors, advanced charts).

```
const HeavyChart = lazy( () => import ( './components/HeavyChart ' ) );

const AnalyticsWidget = () => {
  const = useState ( false );

  return (
    <div>
     < button onClick = {() => setShowChart (true)}> Show chart </ button >
{ showChart && (
       < Suspense fallback = { < div > Generating chart ... </ div > }>
         < HeavyChart />
       </ Suspense >
)}
    </ div >
);
};
```

This approach drastically improves the **TTI (Time to Interactive) indicator** .

4. Advanced Logic Abstractions: Custom Hooks

React's strength lies in its composition. Custom Hooks are a mechanism for extracting stateful logic from components, making code more readable and testable. In 2025, thanks to TypeScript , hooks will become fully secure, generic tools.

4.1 Example: Implementing useDebounce

Debouncing is a technique for delaying a function call until a specified amount of time has passed since the last call. It's crucial for implementing search engines (search-as-you-type) so as not to bombard the API with queries with every keystroke.

Many developers make the mistake of implementing debounce directly in the component, which leads to problems with timer clearing and memory leaks.

Implementation (TypeScript):

```
import { useState , useEffect } from 'react' ;

/**
* useDebounce - Hook that delays updating values.
*
* @param value The value to be debounced (e.g. text from input )
* @param delay Delay time in milliseconds
 * @returns Debounced value
*/
export function useDebounce < T >( value: T, delay: number ): T {
  // 1. Local state holding the delayed value
  const = useState<T >( value );

  useEffect ( () => {
    // 2. We set a timer that will update the state after the ' delay ' time
    const timer = setTimeout ( () => {
      setDebouncedValue (value);
    }, delay );

    // 3. Cleanup function : This is crucial!
    // If the value of ' value ' changes BEFORE the delay time elapses ,
    // React will call this cleanup function.
    // This ensures that the previous timer is removed and won't overwrite the state.
    return () => {
      clearTimeout ( timer );
};
}, [ value , delay ]); // The effect depends on the value and delay

  return debouncedValue ;
}
```

Usage in component :

```
const SearchComponent = () => {
  const = useState ( '' );
  // The API will be queried only 500ms after writing is finished
  const debouncedSearchTerm = useDebounce ( searchTerm , 500 );

  useEffect ( () => {
    if ( debouncedSearchTerm ) {
      // API call : searchApi ( debouncedSearchTerm )
}
},);

  return < input onChange = {(e) => setSearchTerm ( e.target.value )} /> ;
};
```

This abstraction allows the logic to be reused anywhere in the system.

4.2 Example: useFetch with AbortController

Data retrieval is the most common operation in web applications. In 2025, despite the popularity of React Query , it's worth understanding how to write a robust data fetching hook that handles Race Conditions **. A** race condition occurs when a user quickly changes filters—for example, clicks "Category A" and then "Category B." If the answer for "A" comes after the answer for "B," the user will see data for "A" while in category "B."

The solution is AbortController , a native browser API that allows you to cancel requests .

```
import { useState , useEffect , useRef } from 'react' ;

interface FetchState <T> {
  data : T | null ;
loading: boolean ;
error: Error | null ;
}

export function useFetch < T >( url : string , options?: RequestInit ) {
  const = useState < FetchState <T>>({
    data : null ,
    loading : true ,
    error : null ,
});

  // useRef stores the controller instance between renders
  const abortControllerRef = useRef < AbortController | null >( null );

  useEffect ( () => {
    // 1. Cancel previous query if in progress
    if ( abortControllerRef.current ) {
      abortControllerRef.current.abort ();
    }

    // 2. Create a new controller for the current query
    const controller = new AbortController ();
    abortControllerRef.current = controller;

    const fetchData = async () => {
      setState ( prev => ({... prev , loading : true , error : null }));

      try {
        const response = await fetch( url , {
...options,
          signal : controller.signal , // 3. Pass signal to fetch
});

        if (! response.ok ) {
```

```
        throw new Error ( `HTTP Error: ${ response.status } ` );
}

       const jsonData = ( await response.json ()) as T;
       setState ({ data : jsonData , loading : false , error : null });
     } catch ( err : unknown ) {
       // 4. Ignore errors resulting from intentional cancellation ( AbortError )
       if (err instanceof DOMException && err.name === ' AbortError ' ) {
         console .log ( 'Download canceled ( Debounce / Unmount )' );
         return ;
}

       setState ({
         data : null ,
         loading : false ,
         error : err instanceof Error ? err : new Error ( ' Unknown mistake ' )
       });
}
};

   fetchData ();

   // 5. Cleanup : Cancel query when unmounting a component
   return () => {
     controller.abort ();
};
}, [ url ]); // URL dependency

  return state;
}
```

This hook is resistant to memory leaks and asynchronous errors, making it " production-ready ".

5. Clean Code in React + TypeScript

Clean code in React is not only about formatting (Prettier does that), but above all about readability and predictability.

5.1 Changes in React 19: "Ref as a Prop "

cleanup changes in React 19 is the removal of the need to use forwardRef . For years, passing a reference to a child component required wrapping it in a special forwardRef function, which complicated TypeScript typing and made the code less readable.

Formerly (React <19):

```
const MyInput = forwardRef < HTMLInputElement , Props>( ( props, ref ) => (
  < input ref = {ref} { ...props } />
));
```

Now (React 19):

We pass the reference like any other prop . This is a return to simplicity.

```
interface MyInputProps extends React.ComponentProps <'input'> {
  label : string ;
}

// ref is now available directly in props
function MyInput ( { label, ref,...props }: MyInputProps ) {
  return (
    < label >
{label}
      < input ref = {ref} { ...props } />
    </ label >
);
}
```

This change drastically simplifies the creation of component and form libraries.

5.2 SOLID Principles in Components

1. **Single Responsibility Principle (SRP):** A component should do one thing. If a ProductCard retrieves data, processes it, validates a form, and displays a UI, it violates SRP. Logic should be separated into a hook. useProductData , and the form to ProductForm . The component should be view-only.
2. **Open / Closed Principle :** Components should be open to extension (e.g., via props) className , style , children), but closed to modification of their internal workings.
3. **Interface Segregation :** Don't force a component to accept the entire User object if it only needs avatarUrl . Instead interface Props { user: User } , use interface Props { avatarUrl : string } . This facilitates testing and reusability .

6. Tools and Code Quality: ESLint , Prettier , SonarQube

In a professional team, code quality cannot depend on the good will of the developer. It must be enforced through CI/CD.

6.1 ESLint + Prettier Configuration (Flat Config)

In 2025, the .eslintrc format is deprecated . The new standard is **Flat Config** (eslint.config.js), which solves plugin dependency issues .

Example of use

Installation:

npm install -D eslint globals @ eslint / js typescript- eslint eslint - plugin -react-hooks eslint - plugin -react-refresh eslint - config -prettier

Configuration (eslint.config.js):...source

and mport reactRefresh from ' eslint - plugin -react-refresh';

import tseslint from 'typescript- eslint ';

import prettierConfig from ' eslint - config -prettier';

export default tseslint.config (

{ ignores: ['dist', 'coverage', ' node_modules '] },

{

extends: [js.configs.recommended ,... tseslint.configs.recommended],

files: ['**/*.{ ts,tsx }'],

languageOptions : {

ecmaVersion : 2020,

globals : globals.browser ,

},

plugins : {

'react-hooks': reactHooks ,

'react-refresh': reactRefresh ,

},

rules: {

... reactHooks.configs.recommended.rules ,

'react-refresh/only-export-components': [

'war n',

{ allowConstantExport : true },

],

// Enforcing good practices

'@typescript- eslint /no-explicit-any': 'warn',

'no-console': ['warn', { allow: ['warn', 'error'] }],

},

},

Prettier configuration must be last to override other rules

prettierConfig

);

6.2 SonarQube and Static Analysis

SonarQube is a higher-level tool. While ESLint checks for syntax and local errors, SonarQube analyzes ** Code Smells **, cyclomatic complexity (how difficult the code is to test).

Integration with Vite and Vitest :
In order for SonarQube to see code coverage, we need to configure Vitest to generate LCOV reports.

1. ** File ` vitest.config.ts `:**
```typescript
export default defineConfig ({
test: {
coverage: {
reporter: ['text', ' lcov '], // Generates coverage/lcov.info file
      },
},
});
```

2. **File ` sonar-project.properties` :**
This file tells the scanner where to look for sources and reports.

```
properties
sonar.projectKey = my-react-app
sonar.sources = src
sonar.tests = src
sonar.test.inclusions =**/*. test.tsx ,**/*. spec.ts
sonar.exclusions = node_modules /**,dist/**
# Key line for JS/TS:
sonar.javascript.lcov.reportPaths =coverage/lcov.info
```

Running a scanner in the CI/CD pipeline allows you to block the merge of code that does not meet quality standards (e.g. test coverage below 80% or too high function complexity).

7. Global Error Handling and Resilience

An application that can't handle errors loses user trust. Error handling should be layered.

7.1 Error Boundaries

Error Boundaries are React components that "catch" JavaScript errors in the component tree below them, preventing the entire application from crashing.

Important Note for 2025: Error Boundaries still need to be class components (or use the ` react-error-boundary ` library, which is recommended). React 19 hasn't changed this fact explicitly for function components.

Pattern from `react-error-boundary`:

```
import { ErrorBoundary } from "react-error-boundary";

function ErrorFallback ({ error, resetErrorBoundary }) {
return (
<div role="alert" className ="error-card">
    <h3>Something went wrong</h3>
    <pre>{ error.message }</pre>
<button onClick ={ resetErrorBoundary }> Try it again </button>
</div>
);
}

// In code :
< ErrorBoundary fallbackRender ={ ErrorFallback } onReset ={() => setKey (k => k + 1)}>
< CriticalWidget />
</ ErrorBoundary >
```

7.2 Global Service HTTP errors

Instead of writing try / catch in each component, handle network errors (401, 403, 500) globally.

React Approach Query (TanStack Query v5):

This is currently the preferred approach over interceptors. Axios because it integrates with the React query lifecycle .

```
import { QueryClient , MutationCache , QueryCache } from '@tanstack / react-query' ;
import { toast } from 'react-hot-toast' ;

export const queryClient = new QueryClient ({
  queryCache : new QueryCache ({
    onError : ( error ) => {
      // Global Get Error Handling (GET)
      toast.error ( `Error fetching data: ${ error.message } ` );
    },
}),
  mutationCache : new MutationCache ({
    onError : ( error ) => {
      // Global modification error handling (POST, PUT, DELETE)
      // E.g. server-side form validation
      if ( error.status === 500 ) {
        toast.error ( ' Error server . Please try again later.' );
}
},
}),
});
```

Thanks to this, if the backend returns a 500 error, the user will automatically see an aesthetic message (Toast), and the developer does not have to write a single line of error handling code in the component.

Literature

1. **https://react.dev/blog/2024/12/05/react-19** (Accessed: 1/10/2025) – Official React 19 announcement, including details about React Compiler and changes in forwardRef .
2. **https://feature-sliced.design/docs/get-started/tutorial** (Access date: 1/10/2025) – A complete guide to the Feature-Sliced Design (FSD) methodology, discussing the division into layers (Entities , Features , Widgets).
3. **https://github.com/alan2207/bulletproof-react (Access date: 1/10/2025) –** Bulletproof reference architecture documentation and repository React ", which is the foundation of modern enterprise -class applications .
4. **https://tanstack.com/** (Access date: 1/10/2025) – TanStack Documentation Query (React Query) for global error handling and cache management.
5. **https://eslint.org/docs/latest/use/configure/configuration-files** (Accessed: 1/10/2025) – Documentation on the new "Flat Config" standard in ESLint , discussed in the code quality section.