POLITECHNIKA ŚWIĘTOKRZYSKA

Advanced frontend applications – lecture 5

Application integration and implementation

mgr inż. Mateusz Pawełkiewicz

1.10.2025

Introduction: From Local to Global

The lectures so far have focused on building an application that runs locally on the developer's machine. However, in a commercial setting, this is only half the battle. The application must integrate with external services (backend), be capable of secure user session management, and, most importantly, be deployed efficiently and reliably for a global audience.

This talk will walk us through the key steps of transforming a project from a local development environment (localhost) to a fully functional production web application. We'll discuss the integration architecture, security strategies, and automated deployment and monitoring processes.

Backend Integration : REST vs. GraphQL

Two approaches to API communication

Every dynamic front-end application must communicate with the backend to retrieve and send data. Historically, the dominant pattern is **REST (Representational State Transfer)** , a resource-based architecture (e.g., / users , /products) that uses HTTP methods (GET, POST, PUT, DELETE) to manipulate those resources.

GraphQL , a query language for APIs, has gained popularity in recent years. Instead of multiple endpoints defined by the server (as in REST), GraphQL typically provides a single endpoint . The client sends a query specifying precisely what data is needed, which eliminates the common REST problems of "over -fetching " or "under -fetching " data.

REST in Practice: The Axios Library

axios library is most commonly used to communicate with REST APIs in React applications . It is a promise-based HTTP client that simplifies sending HTTP requests. The advantage axios over the browser's built-in fetch includes automatic JSON data transformation, better error handling, and support for interceptors .

Centralize axios configuration

A key practice when using axios is to avoid calling axios.get (...) directly in components. This practice violates the DRY principle (Don't Repeat Yourself) and makes configuration management (e.g. adding authorization headers) more difficult.

axios "instance ." We define a single module (e.g., api.js) that exports a configured axios.create () instance. This allows us to set a common baseURL and, most importantly, centrally manage request and response *interceptors* .

Sample configuration of a centralized API client:

```
// src /api.js
import axios from ' axios ' ;

// 1. Creation instance That constant configuration
const API = axios.create ({
   baseURL : 'https://api.example.com/v1/' ,
   timeout : 10000 ,
   headers : {
      'Content-Type' : 'application/ json ' ,
}});

// 2. Configuring REQUEST interceptor //
This code will launch BEFORE sending each API.interceptors.request.use requests
(
   ( config ) => {
      // Example : dynamic download token and attaching it to the header
      const token = localStorage .getItem ( ' access_token ' );
      if (token) {
         config.headers.Authorization = `Bearer ${token}` ;
} return config ;
}, (error) => Promise . reject (error)
);

// 3. Configuring the RESPONSE
interceptor // This code will run AFTER the response is received, before it goes to.then ()
API.interceptors.response.use (
   ( response ) => response , // Return a successful response
   async ( error ) => {
      // Example: handling token expiration (e.g. status 401)
      if ( error.response ?.status === 401 ) {
         // Token refresh logic ...
         // e.g. redirect to login page
         console .warn ( ' The token has expired or is invalid.' );
} return Promise .reject ( error );
});

export default API;
```

With this approach, React components import an already configured instance (import API from './api ') and use it (API.get ('/ posts ')), and all authorization and error handling logic is managed centrally.

GraphQL in practice: Apollo Client

For GraphQL , while you can use axios or fetch , the industry standard is **Apollo Client** . The reason is that Apollo is much more than an HTTP client; it's a comprehensive state management library built specifically for GraphQL .

Apollo Client automates tasks that would have to be performed manually with REST and Axios :

- **Smart Caching:** Automatically caches query results in a normalized cache. If the same data is requested again, Apollo will immediately return it from the cache .
- **UI State Management:** Provides hooks (e.g., useQuery) that handle loading , error , and data states.
- **UI updates after mutations:** Automatically updates related queries after a data change (mutation) is performed.

Configuring and Using Apollo Client

Configuring Apollo Client involves creating a client instance and making it available to your entire React application using the < ApolloProvider > component.

Step 1: Installing dependencies
above sea level install @apoll about client graphql

Step 2: Configuration in main.jsx

```
import React from 'react' ;
import ReactDOM from 'react- dom /client' ;
import App from './App' ;
import { ApolloClient , InMemoryCache , ApolloProvider } from '@ apollo /client' ;

// 1. Initialization Apollo
client const client = new ApolloClient ({
  uri : ' https://flyby-router-demo.herokuapp.com/' , // Server URL GraphQL
```

```
  cache : new InMemoryCache (), // Startup in- memory
buffering });

// 2. Wrap application component Provider
const root = ReactDOM.createRoot ( document .getElementById ( 'root' ));
root.render (
  < React.StrictMode >
    < ApolloProvider client={client}>
      <App />
    </ ApolloProvider >
  </ React.StrictMode >
);
```

Step 3: Executing Queries and Mutations on the Component

Apollo provides dedicated hooks useQuery to retrieve data and useMutation to modify it.

```
import { gql , useQuery , useMutation } from '@ apollo /client' ;

// 1. Definition Query in the language GraphQL
const GET_LOCATIONS = gql `
query GetLocations {
locations { id name description } }` ;

// 2. Definition mutation
const ADD_TODO = gql `
mutation AddTodo ($type: String!) {
   addTodo (type: $type) {
id
type } }` ;

function DisplayLocations () {
  // 3. Use hooka useQuery
  const { loading, error, data } = useQuery (GET_LOCATIONS);

  // 4. Use hooka useMutation
  // Returns ' addTodo ' function to be called and condition mutations
  const = useMutation (ADD_TODO);

  if (loading) return <p> Loading ...</p> ;
  if (error) return <p> Error : { error.message }</p> ;

  const handleAddTodo = () => {
    addTodo ({ variables : { type : ' New task ' } });
};
```

```
 return (
   <div>
<button onClick ={ handleAddTodo } disabled={ mutationLoading }>
      Add Task
</button> { data.locations.map (({ id, name, description }) => (
<div key={id}> <h3>{name}</h3> <p>{description}</p> </div> ))} </div>
);}
```

JWT Token Authentication

Token Storage Dilemma

Regardless of whether we use REST or GraphQL , the most popular authentication method in modern applications is **JWT (JSON Web Token)** . After a successful login, the backend returns a token (a string) to the client, which must be included in each subsequent request.

token be safely stored on the client side (in the browser) ? The two most common places are localStorage and httpOnly. cookie .

localStorage : Convenience at the expense of security

localStorage is a browser API that allows you to store data (key-value) that is available even after the browser is closed.

- **Advantage:** Very easy to implement. JavaScript can easily read the token (localStorage.getItem (' token ')) and append it to the Authorization header (as we saw in the interceptor example axios).
- **Cross-Site XSS vulnerability Scripting)** . If an attacker manages to inject a malicious script into a website (e.g., via a vulnerability in the comment form), the script gains full access to localStorage . It can immediately read the token and send it to the attacker's server, allowing for complete hijacking of the user's session.

httpOnly cookie : XSS resistance, a new risk

An alternative is to store the token in a cookie set by the server with the httpOnly flag .

- **Advantage: The** httpOnly flag makes the cookie *completely inaccessible* to client-side JavaScript . Document.Cookie will not display it. This completely eliminates the risk of token theft via XSS attacks. The browser automatically includes this cookie in every request sent to the server domain.
- **Disadvantage: Vulnerability to CSRF (Cross-Site Request Forgery)** . Because the browser *automatically* sends cookies with every request (even if the request comes from a different, malicious site), the application becomes vulnerable to CSRF attacks. Modern browsers have significantly reduced this risk with the SameSite attribute (specifically SameSite=Strict), but have not eliminated it entirely.

Hybrid Approach: Best Practice 2025

None of the above methods are perfect. Therefore, modern, secure applications employ a hybrid approach that utilizes two types **of** tokens : **Access Tokens** and **Refresh Tokens . Token (Refresh Token)** .

This architecture works as follows:

1. **Access Token (short-lived , e.g. 15 minutes):** It is stored **in the application memory (e.g. in React state or global variable)** .
    - *XSS Security:* High. An XSS script cannot easily read it unless it is stored persistently (though advanced XSS attacks still pose a memory risk).
    - *CSRF Security:* Full. The token is not sent automatically; it must be manually included in the request by the client (e.g., by an interceptor) axios).

2. **Refresh Token (long-lived , e.g. 7 days):** It is stored in **httpOnly cookie** with the Secure and SameSite=Strict flags .
    - *XSS Security:* Full. JavaScript cannot access it.
    - *CSRF Security:* High. SameSite=Strict prevents cookies from being sent on cross-domain requests.

Flow of action:

- The user logs in. The server returns an Access Token (in the JSON response body) and a Refresh Token (in httpOnly cookie).
- The application stores the Access Token in memory and uses it for all API requests.
- The user refreshes the page (F5). The Access Token disappears from memory.
- The application (e.g., at startup) sends a request to a special endpoint / refresh_token . This request *does not contain an* Access Token (because it does not exist), but the browser *automatically appends* httpOnly to it cookie from Refresh Token .
- Server verifies Refresh Token and if it is valid, it sends back a new, fresh Access Token (for the next 15 minutes).
- The application saves the new Access Token in memory and continues working without requiring the user to log in again.

This approach combines httpOnly security cookie (XSS protection for long-term key) with in-memory token resistance to CSRF attacks.

Table: Comparison of JWT Storage Strategies

Method	**localStorage**	**httpOnly cookies**	**Hybrid (Memory + httpOnly)**
Major Risk	XSS (Critical)	CSRF (Medium)	Minimal (requires advanced XSS)
XSS protection	Lack	Perfect	Perfect (for Refresh Token)
CSRF protection	Perfect (requires SameSite)	Risk (mitigated by SameSite)	Perfect (for Access Token)
Durability (after F5)	Yes	Yes	Yes (via refresh mechanism)
Complexity	Low	Medium (requires backend coordination)	High (requires refresh logic)
Recommendation 2025	Not recommended	Acceptable (with SameSite=Strict)	**Recommended**

Configuring Environments in Vite Applications

Problem: Different configurations for different environments

An application almost always requires different settings depending on where it is running:

- **Development (Local):** Connects to local http://localhost:8000/api.
- **Staging (Test):** Connects to https://api.staging.example.com.
- **Production :** Connects to https://api.example.com and uses a different API key for the analytics service.

Hardcoding these values in code is unacceptable. Environment variables are the solution.

.env files in Vite

Vite has built-in support for environment variables using .env files in the project root:

- .env : Always loaded . Contains default values.
- .env.development : Only loaded in the development environment (npm run dev) .
- .env.production : Only loaded in the production environment (npm run build) .
- . env.local : Always loaded, but has higher priority than . env . Should be in . gitignore and contain local secrets.

VITE_ Prefix: Built-in Leak Protection

This is a key difference from other tools. To prevent accidental leakage of sensitive keys (e.g., DB_PASSWORD) to the client (browser) code, Vite **only** exposes variables that begin with the VITE_ prefix.

Example:

.env.production file :

```
DB_PASSWORD=super_secret_key_123
VITE_API_URL=https ://api.example.com
VITE_GOOGLE_ANALYTICS_KEY=GA_PROD_98765
```

In the application code:

```
// Access to variables is via a special object import.meta.env
console .log ( import .meta.env.VITE_API_URL ); // "https://api.example.com"
console .log ( import .meta.env.VITE_GOOGLE_ANALYTICS_KEY ); // "GA_PROD_98765"

// This variable WILL NOT be available in the client code!
console .log ( import .meta.env.DB_PASSWORD ); // undefined
```

This security feature forces the developer to make a conscious decision about which variables to make publicly available in the browser.

Build and Optimization

The npm run build command

Once the application is ready to be deployed, we run the command npm run build , which in the case of Vite calls vite build .

What's happening "under the hood": Rola Rollup

Vite is known for its lightning-fast development server, which doesn't require on-the-fly bundling . However, when building for production, this approach would be inefficient (too many small files to download).

That's why vite build uses a bundler underneath called **Rollup** . Rollup performs three key optimization tasks:

1. **Bundling :** Combines all JavaScript and CSS modules into fewer files (called " chunks ") to minimize the number of HTTP requests.
2. **Minification : Removes all unnecessary characters from your** code (whitespace, comments) and shortens variable names to drastically reduce file sizes **.**
3. **Tree- shaking: This is an advanced** dead-code elimination technique. elimination). Rollup analyzes the import paths throughout the application and if it detects that a function (or entire module) has been imported but *never used* , it will be *removed* from the final production file.

The result is a dist folder containing highly optimized, static assets (HTML, CSS, JS), ready to be deployed to any server.

Build verification : vite preview

After running npm run build , the development (dev) server is no longer running. So how can I test if the production build is working correctly?

Vite offers a vite recommendation preview (run by npm run preview). This command starts a simple server that serves files from the built dist folder .

Warning: vite Preview is for local previewing and testing of the production build *only* . It **should absolutely not** be used as a production server. It is neither efficient, secure, nor scalable. It lacks key production features such as Gzip / Brotli compression or appropriate security headers.

Deployment : Application Deployment

With the dist folder , we have several options for deploying our application.

Comparison of implementation platforms

The choice of platform depends on the complexity of the project, budget and required control.

- **Vercel / Netlify :** These are modern " GitOps " (or " Jamstack ") platforms. They work by connecting to a Git repository (e.g. GitHub).
  - *How it works:* You push code to a repository (e.g. to the main branch), the platform automatically detects the change, runs npm run build and deploys the result to the global CDN (Content Delivery Network).
  - *Advantages:* Incredible simplicity (zero server configuration), free plans for open-source projects , automatic CI/CD, previews for each Pull Requesta , serverless feature support .
  - *Specializations:* Vercel was created by the creators of Next.js and is optimal for it. Netlify has a richer ecosystem of built-in services, such as Netlify Forms or Identity .
- **GitHub Pages :** Free static hosting service from GitHub .
  - *Advantages:* Completely free, integrated with the repository.
  - *Cons:* Supports *only* static pages (no serverless features , no backend). Ideal for documentation, portfolio , or simple websites.
- **Own VPS (e.g. DigitalOcean , AWS) + Nginx :**
  - *How it works:* You rent a virtual server (VPS), install an operating system (e.g. Ubuntu), a web server (e.g. Nginx) and configure everything *yourself :* dist file transfer (e.g. via scp), Nginx configuration for serving files, SSL certificates, logs, etc.
  - *Advantages:* Full control over the environment, ability to host any backend (Node , Python , Go) on the same machine.
  - *Cons:* Full responsibility. Requires advanced knowledge of systems administration, security, and networking. This is the most complex and time-consuming option.

Table: Comparison of implementation options

Characteristic	**Vercel / Netlify**	**GitHub Pages**	**Own VPS (Nginx)**
Ease of use	Very easy (Git-Ops)	Easy	Very difficult (full setup)

Backend support	Serverless Features	Lack	Full (any backend)
CI/CD	Built-in, automatic	Requires GitHub Actions	Requires manual configuration (e.g. GitHub Actions + SSH)
Global CDN	Yes (built-in)	Yes (built-in)	No (requires manual configuration e.g. Cloudflare)
Cost (starting)	Free plan	Free	Paid (from ~$5/month)
Perfect for	Jamstack applications , Next.js , commercial projects	Portfolio , documentation, static pages	full-stack applications , custom requirements

Practical Example: Deployment on Vercel from GitHub

Vite application on Vercel is extremely simple:

1. Push the Vite app code to the GitHub repository .
2. Log in to Vercel (using your GitHub account).
3. Click " Add New... Project".
4. Select the GitHub repository you want to deploy.
5. Vercel will automatically detect that it is a Vite project and set the appropriate build command (vite build) and the output directory (dist).
6. Click " Deploy ".
7. After a few minutes, the application is deployed and available globally at * .vercel.app . From that point on, any git push to the main branch will automatically trigger a new production deployment.

CI/CD: Introduction to GitHub Actions

Vercel and Netlify hide the complexity of CI/CD from us. However, if we use GitHub Pages or your own VPS, you'll need to configure this process yourself. The most popular tool for this is **GitHub. Actions** .

CI/CD is an abbreviation for:

- **CI (Continuous Integration (Continuous Integration):** The practice of automatically running tests and building applications *every time* someone pushes code to a repository. This prevents the main code branch from being broken.
- **CD (Continuous Deployment - Continuous Deployment:** Automatically deploying an application to production *after* it successfully passes the CI (test and build) stage.

Practical Example: Validation Workflow (CI)

We create a file in our repository: .github / workflows / ci.yml . This file defines a " workflow ," which is a set of steps for GitHub to execute .

The following validation workflow will run on every push and pull_request to check if the code is valid (passes tests and builds).

```
#. github / workflows / ci.yml
name : CI Workflow - Test and Build

# Run this workflow on every push and pull request
on: [ push , pull_request ]

jobs :
 test_and_build :
  runs-on : ubuntu-latest # Use a virtual machine with Ubuntu

  steps :
   # Step 1: Downloading the code from the repository
   - name : Checkout code
     uses : actions /checkout@v4 #

   # Step 2: Configuring the Node.js environment
   - name : Setup Node.js 20.x
     uses : actions /setup-node@v4 #
     with :
       node-version : '20.x'
       cache : ' asl ' # *Key optimization*: caches node_modules

   # Step 3: Installing Dependencies
   - name : Install dependencies
     # ' npm ci' is faster and more deterministic than ' npm install ' in CI
     run: above sea level you #

   # Step 4: Running the tests
   - name : Run tests
     run: above sea level test # - If this fails, the workflow will stop here
```

```
    # Step 5: Verify the production build
    - name : Run build
      run: above sea level run build # - Checks if the application builds without errors
```

This workflow *It doesn't deploy anything* . Its sole purpose is to validate and protect the main branch . If tests or builds fail, GitHub Actions means Pull Request as "broken" and will prevent it from being merged .

A separate file (e.g. deploy.yml) would be configured to *only run* after a successful CI on the main branch (e.g. on: push : branches : [main]) and contain the steps for the actual deployment (e.g. copying dist files to the VPS via SSH or using an action to deploy to GitHub Pages).

Error Monitoring and Observability

The app is deployed. How do we know if it's working correctly for users? The famous "it works for me" problem is real – bugs often depend on the browser, network, extensions, or specific user actions. Relying on user reports is inefficient.

Bug Tracking vs. Session Replays

There are two main approaches to monitoring front-end applications

1. **Sentry (Error Tracking):** Answers the question: " *What* went wrong?" Sentry is a tool designed for developers. Its purpose is to catch every unhandled exception in your code (e.g., TypeError : cannot read property ' name ' of undefined), grouping errors and providing the developer with a full *stack trace* (stack trace), breadcrumbs (what the user clicked before the error) and information about the environment (browser, OS).
2. **LogRocket (Session Replays):** Answers the question: " *What did* the user see when something went wrong?" LogRocket acts as a "security camera" (or "black box") for the application. It records the user session—DOM changes, mouse movements, console logs, and network requests—allowing the developer to watch a video of the exact problem. This is invaluable for debugging hard-to-reproduce UX bugs and also identifies signals of user frustration, such as " rage " clicks " (repeated, rapid clicking) or " dead clicks " (clicking on inactive elements).

The two markets overlap – Sentry added basic session replays, and LogRocket offers error tracking. However, their fundamental purpose is different. Sentry is a technical diagnostic tool (stack trace), while LogRocket is a visual context tool (video).

Table: Sentry Comparison vs. LogRocket 2025

Characteristic	Sentry	LogRocket
Main Purpose	Bug and performance tracking	Session Replays and UX Analytics
Key Functionality	Error grouping, stack trace , alerts	Session video playback, heat maps
Perfect for...	Debugging technical issues , backend monitoring	Debugging visual/UX issues , analyzing user journeys
Pricing Model (mainly)	Payment for the number of events (errors)	Payment for the number of sessions (recordings)
Backend support	Wide (Node , Python , Java, etc.)	frontend focused)

Recommendation: Always start by implementing Sentry . It's a fundamental tool for maintaining application health. LogRocket is added later when deeper insights into user behavior are needed or when errors reported by Sentry are difficult to reproduce.

Practical Example: Integrating Sentry with React

Adding Sentry to React is simple and consists of two key steps:

Step 1: Installation and Initialization
above sea level install @ sentry / react
In main.jsx (or index.js), *before* calling ReactDOM.createRoot :

```
import React from 'react' ;
import ReactDOM from 'react- dom /client' ;
import App from './App' ;
import * as Sentry from "@sentry/react" ;

// 1. Sentry
Initialization Sentry.init ({
 dsn : "YOUR_DSN_KEY_FROM_SENTRY_PANEL" ,
 integrations :,
 // Setting sampling
 tracesSampleRate : 1.0 , // 100% of transactions to be monitored performance
 replaysSessionSampleRate : 0.1 , // 10% of sessions will be recorded
```

```
});

const root = ReactDOM.createRoot ( document .getElementById ( 'root' ));
root.render (
 < React.StrictMode >
  <App />
 </ React.StrictMode >
);
```

Step 2: Configuring the Error Boundary (Crucial !)
Sentry's initialization catches global errors, but it won't catch rendering errors inside React components (because React catches them itself). For Sentry to see them, we need to wrap our application in a Sentry -provided ErrorBoundary component .
In App.jsx (or wherever your main router is):

```
import * as Sentry from "@sentry/react" ;
import MyRoutes from './ MyRoutes ' ; // Example component with routing

function App() {
 return (
  // 2. Wrap the entire application (or key parts of it)
  < Sentry.ErrorBoundary
   fallback ={< p> An unexpected error occurred. Our team has been notified.</p>}
> < MyRoutes />
</ Sentry.ErrorBoundary >
);}

export default App ;
```

Now, if any child component of MyRoutes throws an error while rendering , Sentry.ErrorBoundary will catch it, show the user a fallback interface and automatically send a full report with the component stack trace to Sentry .

Introduction to Progressive Web Apps (PWA)

Definition and benefits

Progressive Web App (PWA) is not a specific technology, but a set of standards and practices that allow a web application to mimic the behavior of a native application (mobile or desktop).

Key benefits include:

1. **Installability:** Users can "add to home screen." The app then launches in its own window, without the browser address bar, giving it a native app feel.
2. **Offline - first support :** Using a technology called **Service Worker** , an application can cache its resources (HTML, CSS, JS, images, and even API data). This allows the application to run and view content even without an internet connection .

Practical Example: Configuring vite-plugin-pwa

Manually creating a Service Worker and application manifest is complex. The Vite ecosystem offers a plugin called vite-plugin-pwa that automates 90% of this process.

Step 1: Installation
above sea level install vite-plugin-pwa -D
Step 2: Configuring vite.config.js
the plugin to the Vite configuration file , defining the application manifest and update strategy.

```
import { defineConfig } from ' vite '
import react from '@ vitejs / plugin-react '
import { VitePWA } from ' vite-plugin-pwa '

export default defineConfig ({
  plugins :

    // Web App Manifest Definition
    // This file tells the browser what the application should look like after "installation"
manifest: { //
      name : 'My PWA Application' ,
      short_name : ' MyPWA ' ,
      description : 'Description of my progressive web app.' ,
      theme_color : '# ffffff ' , // Application toolbar color
      icons : [ // Icons used on the home screen
{ src : 'pwa-192x192.png' , // The file must be located in the /public folder
          sizes : '192x192' ,
          type : ' image / png '
}, { src : 'pwa-512x512.png' , // The file must be in the /public folder
          sizes : '512x512' ,
          type : ' image / png '
} ] },
```

```
    // Optional: Service Worker configuration ( Workbox )
    // By default, Workbox caches all static resources
    workbox : {
      globPatterns : [ '**/*.{ js,css,html,ico,png,svg }' ] // Cache these files
} }) ]})
```

Step 3: Adding icons
You need to create the required icons (e.g. pwa-192x192.png) and place them in the /public folder.
After executing npm run build , the vite-plugin-pwa plugin automatically:

1. Generates a manifest.webmanifest file from the manifest object.
2. It will generate a Service Worker file (sw.js) using the Google Workbox library that will cache the resources defined in workbox.globPatterns .
3. It will inject the appropriate <link rel="manifest "> tags and the Service Worker registration script into the index.html file.

Summary and Audit: Analysis of the Lighthouse Report

Audit of the implemented application

Once an app has been deployed to Vercel (or another platform), we need to close the loop and measure the quality of our work. The primary tool for this is **Lighthouse** , which is integrated with Chrome's DevTools .

To conduct an audit:

1. Open the deployed app in Chrome (preferably in Incognito mode so extensions don't interfere with results).
2. Open DevTools (F12).
3. Go to the " Lighthouse " tab.
4. Select categories (e.g. Performance, Accessibility , SEO) and mode (Mobile).
5. Click " Analyze " page load ".

Lighthouse Results

Lighthouse evaluates a website across five key categories, giving it a score from 0 to 100 (where 90+ is excellent).

1. **Performance:** The most important and complex category. It measures how quickly a page is *viewed* by a user. It focuses on three **Core Web Vitals** (Key Web Vitals), which are also Google's ranking factors:
    - **LCP (Largest Contentful Paint):** *Loading performance.* How quickly does the largest element (e.g., main image, text block) appear on the screen?
    - **TBT (Total Blocking Time):** *Interactivity .* How long is the main browser thread "blocked" by JavaScript , preventing it from responding to a user click?
    - **CLS (Cumulative Layout Shift):** *Stability visual .* Do page elements "jump" while loading (e.g. an ad loads and moves the text the user wanted to click on)?
2. **Accessibility :** Checks whether the website is usable for people with disabilities (e.g. whether images have alt attributes, whether color contrast is sufficient).
3. **Best Practices :** Technical audit (e.g. whether the site uses HTTPS, whether there are any errors in the console).
4. **SEO (Search Engine Optimization): Basic check whether the page has SEO-critical elements** (e.g. <title> tag , <meta description >).
5. **PWA (Progressive Web App):** (Will appear if a Service Worker is detected .) Checks if the manifest is valid, if the app works offline , and if it is installable.

Lighthouse score (e.g., "Performance: 60") isn't a judgment, but a starting point for analysis. The real value of this tool lies below the scores, in the " Opportunities " and "Diagnostics" sections. Lighthouse is an advanced diagnostic tool. It tells us *exactly what's slowing down our site (e.g., "Optimize images," "Remove* render- blocking resources ") and provides a list of specific tasks to perform to improve the user experience.

Literature

1. https://www.apollog r aphql.com/docs/react/get-started (Accessed: 1/10/2025) – Official Apollo Client configuration guide , explaining in detail the mechanisms of caching (InMemoryCache) and managing GraphQL queries .
2. https://auth0.com/blog/ r efresh-tokens-what-are-they-and-when-to-use-them/ (Access date: 1/10/2025) – Specialized analysis of token refresh mechanisms (Refresh Tokens), which is the foundation for the hybrid authentication model discussed in the lecture.
3. https://docs.github.co m /en/actions/use-cases-and-examples/building-and-testing/building-and-testing-nodejs (Access date: 1/10/2025) – GitHub Technical Documentation Actions , describing CI/CD processes, test automation, and building production artifacts for Node.js environments .
4. https://docs.sentry.io/ p latforms/javascript/guides/react/ (Accessed: 1/10/2025) – Sentry integration guide for React applications , covering advanced error diagnostics, Error configuration Boundary and performance monitoring.
5. https://web.d ev /articles/vitals (Access date: 1/10/2025) – A compendium of knowledge about Core Web Vitals (LCP, FID, CLS), essential for interpreting Lighthouse audit results and optimizing user experience.