POLITECHNIKA ŚWIĘTOKRZYSKA

Advanced frontend applications – lecture 1

Introduction

mgr inż. Mateusz Pawełkiewicz

1.10.2025

Lecture Objective: To prepare a modern front-end developer's work environment and understand the architecture of applications based on React and Vite. We'll discuss the architecture of modern web applications (various rendering and page generation methods), the roles of popular front-end frameworks, the benefits of using Vite, the structure of a React + Vite project, key configuration files, a typical developer workflow, the basics of JSX/TSX with TypeScript, code organization with imports and path aliases, and methods for debugging React applications.

Architecture of modern web applications (SPA, CSR, SSR, SSG)

Modern web applications can be built with a variety of architectures and rendering models. Understanding concepts like **SPA** , **CSR** , **SSR** , and **SSG** is crucial to informed application design:

- **Single Page Application (SPA)** – a single-subpage application. This is a web application that loads as a single HTML document, and subsequent user interactions do not cause the page to reload. All application logic (UI, JS code, server communication) is loaded once in the browser, and subsequent subpages are rendered dynamically on the client side. Initially, the HTML contains almost no content – **all content is loaded via JavaScript** and embedded in a single page "container." Popular libraries and frameworks for building SPAs include React , Angular , and Vue . The advantages of SPAs include very smooth, application-like interaction (after loading, there are no full page reloads during navigation) and full control over the client-side architecture. However, the disadvantages include potentially long initial load times (all application code must be downloaded before content appears) and SEO issues – the page initially has no content loaded, so search engine crawlers "see" an empty document. SPAs are mainly suitable for applications rich in interactions, where the dynamic interface and frequent data updates on the client side are important, and traditional subpages with static content are less important.

- **Client-Side Rendering (CSR)** – client-side rendering. This approach is closely related to SPAs. In the CSR model, **the browser renders page content using JavaScript** . The server initially provides minimal HTML (usually just a div container and links to JS/CSS files), and all display logic runs in the browser. This means **the user may see a blank page until JavaScript loads and renders the content** . The disadvantages include a longer time to content appearance (especially on slower connections or lower-powered devices) and a negative impact on SEO (search engine bots don't execute JavaScript by default, so without additional techniques, they may not see CSR content). The advantage is high interactivity and the ability to dynamically update the UI in the browser without reloading. **Note:** In practice, the term CSR is often used interchangeably with SPA, as most SPAs use client-side rendering.

- **Server-Side Rendering (SSR)** – rendering after page In this approach, the full HTML for the page is generated on the server **with each request** , and only then, the finished, "filled" HTML content is delivered to the browser. This allows the user to immediately receive ready-made content (text, images) without having to wait for JavaScript to

execute. This improves the first display time and significantly simplifies search engine indexing. After the HTML page is delivered, the JS code is included, which, through **hydration** , takes over client-side control. This involves attaching events and enabling further interaction, as in SPAs. In SSR, each subpage is generated on the fly (or partially cached), which means higher server load and more database queries, but the content is always up-to-date and personalized to the user's request. SSR can be slightly slower **to full interactivity** than pure SPAs (because it must process the JS twice – once on the server, once in the browser), but this is often a worthwhile compromise. SSR is recommended for applications that require **good SEO or content that depends on the user's context (personalization)** , e.g. news portals, e-commerce with dynamic recommendations, etc. In the React ecosystem, a popular SSR solution is Next.js (which can render pages on the server or statically, depending on the needs).

- **Static Site Generation (SSG)** – static page generation. Pages are **fully pre-rendered during the application build phase** (build time) and then served as ready-made HTML files with static content. Unlike SSR, generation doesn't occur on every request, but rather in advance—for example, when a new version of the site is deployed. Such pages don't require server processing power to generate content on the fly, so they can be **hosted on a simple server** and handle very high traffic with minimal latency. The advantages of SSG include **lightning-fast response times, high performance, and excellent SEO** (we get ready-made HTML). The disadvantage is the lack of dynamic content— **content changes require rebuilding and redeploying the page** , which makes it difficult to present frequently updated data. SSG is ideal for blogs, documentation, and marketing sites—where the content is relatively static or updated infrequently and doesn't require user personalization. Examples of SSG tools include Gatsby (for React) or Next.js in static mode. It is worth adding that modern solutions allow for mixing approaches – e.g. Next.js allows you to combine SSG with SSR at the level of individual pages (so-called ISR – Incremental Static Regeneration, i.e. refreshing static pages from time to time).

Architecture Summary: The choice between SPA/CSR, SSR, and SSG depends on the project's needs. SPA/CSR provide maximum interactivity at the expense of SEO and first render time. SSR provides up-to-date, dynamic content and improved SEO, but at the cost of increased server load. SSG guarantees excellent speed and SEO for static content, but does not support personalization or frequent changes without additional mechanisms. Increasingly, these approaches are being combined in a hybrid fashion to **achieve the advantages of each** —for example, a website can be built as an SPA but use SSR for the initial rendering of key subpages, or be statically generated and enriched with dynamic fragments by the client.

The role of frontend frameworks – React, Vue, Angular (a short comparison)

Front-end frameworks and libraries play a crucial role in building modern applications, especially SPAs. The most popular are **React** , **Angular** , and **Vue** . While they address similar

problems (component-based interface architecture, efficient DOM rendering, UI state management), they differ in philosophy and scope. A brief comparison of these technologies:

- **React** is a JavaScript library developed by Facebook (Meta), primarily used for building views (the V in MVC). React provides **a declarative component model** based on **JSX** and manages a virtual DOM for efficient view updating. It is **very flexible** , but **focuses solely on the view layer** . This means that React often uses additional libraries for routing (React Router), global state management (Redux, Context API, etc.), and query handling (e.g., the fetch or Axios libraries). React emphasizes the simplicity of its core and **a strong extension ecosystem** . From a learning perspective, it has a moderate entry threshold (you need to understand JSX and a few concepts like **hooks** , the component lifecycle, and state/props), but thanks to its large community, resources and libraries are easy to find. React is currently **the most popular front-end approach** , and due to this popularity, there are many developers and community support available.

- **Angular** – **a full-fledged front-end framework** developed by Google (it's a newer version of Angular, completely different from the historical AngularJS). Angular is **a comprehensive toolkit** : it has a built-in module system, routing, its own way of managing state, forms, server communication, and more. Angular **requires the use of TypeScript** (it's integrated with it by default), which improves code quality and makes large projects easier to maintain. Overall, Angular is heavier and has **a higher initial complexity** – it requires learning many concepts (components, services, dependency injection, pipes, the Zone mechanism for detecting changes, etc.). However, in return, it offers **a structured approach** – perfect for large enterprise applications where code consistency and full functionality out of the box (fewer external dependencies) are important. You could say that Angular is "batteries included": you get everything in one package, but at the cost of greater complexity. The main difference is that Angular is **an MVC/MVVM framework** , where we have HTML templating with additions (e.g. *ngIf,* ngFor tags), and logic in components in TS - unlike the pure, functional style of React based only on JavaScript/JSX.

- **Vue** – Described as a "progressive framework" created by Evan You (who worked on Angular, hence some inspiration). Vue strives to combine the strengths of React and Angular, being **lighter and simpler** than Angular, but more "complete" than pure React. In Vue, components can be written in .vue files using **HTML templates with add-ons** (similar to Angular templating) and a built-in reactive data mechanism. Vue is valued for its **gentle learning curve** – a simple component can be written in familiar-looking HTML with a touch of JS. For more complex needs, Vue also has an ecosystem: Vuex (for state management, although Vue 3 favors the built-in Composition API), Vue Router, etc. Vue's performance is very good (comparable to React), and its core size is small. Unlike Angular and React, Vue isn't built by a corporate giant, but it does have a strong community. It can be thought of as a "middle ground" solution: **more structured than React , less complex than Angular** .

In simple terms: *Angular is a full framework* , *React is a UI library* , *and Vue is something in between – a lightweight progressive framework* . All three tools allow you to create modern, fast user interfaces and are based on components and the principle of unidirectional data flow (although Angular also supports bidirectional data binding). The choice often depends on the project's requirements and the team's experience. In practice, **React currently dominates the market** , while Angular is sometimes chosen for enterprise projects (especially when TypeScript and a full out-of-the-box framework are preferred), while Vue is popular in the open-source community and smaller projects where rapid productivity is valued.

Why Vite? – Modern Bundler, HMR, TypeScript Support

When creating front-end applications, a tool for building and maintaining the project (a so-called *bundler* or *build tool*) is essential. For years, Webpack was the standard , but since 2020, **Vite has gained significant popularity** . Vite (the French name means "fast") is a modern bundler created by Evan You (author of Vue.js) with **developer speed in mind** . Compared to older solutions, Vite works differently: it uses **browser-based ES modules (ESMs)** and **rebuilds code on demand** . Here are the key advantages of Vite:

- **Rapid dev server startup** – Vite doesn't instantiate the entire application bundle when the development server starts. Instead, it launches a local server that serves files directly as ES modules. When a browser requests a module, Vite dynamically transpiles and delivers it. This ensures **virtually instantaneous startup time** , even in large projects, whereas Webpack, for example, must preprocess the entire project before bringing up the server. The difference is particularly noticeable in cold startup: Webpack can take many seconds to start a large application, while Vite can take a fraction of a second.

- **HMR (Hot Module Replacement)** . Vite has **built-in** ultrafast HMR. When working on a project, after saving a file, changes are **instantly refreshed in the browser without reloading the entire page** , and importantly, **they preserve the application's state** (e.g., without resetting the content of form fields or component state). Vite achieves this by tracking module dependencies and precisely loading only the changed code fragments. Updates typically occur in <100ms , making development smoother – the effect of changes is immediately visible. HMR also existed in Webpack, but Vite's HMR is inherently faster and less finicky because it relies on native ES modules.

- **TypeScript Support (and More)** – Vite supports .ts and .tsx files by default, with no additional configuration required. Under the hood, it uses the ultra-fast esbuild compiler (**written** in Go) to transpile TS to JavaScript, making it ~20-30x faster than a traditional tsc run . Importantly, Vite transpiles TypeScript **without type checking** – it assumes that your editor (IDE) or a separate background process will handle this. This "transpile only" mode is intentional to avoid performance slowdowns – Vite focuses on delivering code to the browser as quickly as possible, and static type-checking can be performed in parallel (e.g., by running tsc --noEmit --watch or the IDE's built-in tools). In addition to TS, Vite also out-of-the-box supports modern JavaScript (ESNext), JSX/TSX (via appropriate plugins), importing CSS files, images, and more.

- **Modern ecosystem and easy configuration** – Vite uses Rollup by default to build the production version, allowing for **optimal bundle generation** . Vite configuration is **simple and minimal** – most sensible settings are default. A typical vite.config.js/ts configuration file can be a few lines long (often, just plugging in a framework plugin and that's it). There's no need for lengthy configuration files like Webpack – Vite works with **smart defaults** for popular scenarios (React, Vue). Of course, it can be extended through **plugins** (Vite is compatible with many Rollup plugins) and custom options, but with a typical React/TS project, practically nothing needs to be changed. This advantage is particularly appreciated, as it allows you to focus on writing the application instead of tedious tool configuration.

- **Fast builds and pre-bundling** – Although Vite delivers its greatest benefits in development mode, it's worth mentioning that **production builds** are also efficient. Based on Rollup, it automatically **splits code into separate chunks** (e.g., vendor libraries) and applies optimizations like tree-shaking, minification, compression, etc. The build process is simple – with a single vite command. build creates static files in /dist . Furthermore, when the dev server starts, Vite **pre-bundles dependencies** (using esbuild)—it combines and optimizes node_modules bundles immediately, so the browser doesn't have to load dozens of separate modules. This makes **the dev server's cold start and the first application load time** in dev mode faster than with classic bundlers .

Overall, Vite provides **a faster and smoother** developer workflow compared to older tools. As one review put it, *"Vite provides a faster development server, instant HMR, and simpler configuration compared to Webpack's heavier, more complex setup. "* This is why it has become the default choice for many new frontend projects (especially React, Vue, Svelte).

Project structure in Vite + React

When you create a new React project using Vite, you're presented with a specific file and directory structure. Understanding this structure will help you navigate the code efficiently. The typical structure of a project generated with the npm create vite@latest my-app -- --template react-ts command looks like this:

```
my-app/ # Main project folder
├ ── node_modules/ # Installed npm dependencies
├ ── public/ # Static files served "as-is"
│ └── vite.svg # (e.g. Vite logo image)
├ ── src/ # Application sources (TypeScript/TSX code, styles, resources)
│ ├ ── assets/ # (e.g. images, icons used in code)
│ │ └── react.svg
│ ├ ── App.tsx # Main React application component
│ ├ ── App.css # CSS styles for the App component
│ ├ ── index.css # Global application CSS styles
│ ├ ── main.tsx # Application startup file (entry point)
│ └── vite-env.d.ts # Type definitions for Vite (auxiliary)
├ ── .gitignore # Git configuration file (ignored files)
```

```
├ ── index.html # Main application HTML file
├ ── package.json # NPM manifest - project meta data and scripts
├ ── tsconfig.json # Main TypeScript configuration
├ ── tsconfig.app.json # TS configuration specific to the application code
├ ── tsconfig.node.json # TS configuration for tools (e.g., Vite config)
├ ── vite.config.ts # Vite configuration file
└── ... (e.g. linter configurations, readme, etc.)
```

(The above layout corresponds to the default Vite template with React+TS . There may be minor differences depending on the Vite version and libraries used.)

A few things are worth noting:
- The **public/ directory** is used to store static files that **are not processed by bundler** . Anything we place there will be available under the same relative path as the directory (e.g., public/vite.svg is available as /vite.svg). This directory typically contains favicons, static images, PWA *manifest.json* files , etc.
- The **src/ directory** contains **the entire application source code** . This is where our React components (.tsx files), logic files (e.g., utilities, hooks), .css / .scss styles , and possibly subdirectories organizing the code (e.g., components/ , hooks/ , pages/ – depending on the structure we adopt). The generated project already has a sample App.tsx component and a main.tsx file that assembles it.
- **index.html** is located in the project's main folder (not in src). This is **the HTML template** for the page. It contains basic meta tags and a #root div where React will embed the application. Vite treats this file as an entry point for the development server – when running in dev mode, Vite automatically injects the appropriate <script> into it, pointing to our code (e.g., <script type="module" src="/src/main.tsx"></script> appears automatically). Important: we don't need to manually add script tags in HTML; Vite manages this for us based on imports in the code.
The **main.tsx file** in src/ is the starting point of the React application. This is where the React component tree is initialized and connected to the browser's DOM. In a Vite template, main.tsx usually looks something like this:

```
import { createRoot } from 'react- dom /client';
import App from './App';
import './index.css';

createRoot ( document.getElementById ('root')!).render(
<App />
);
```

The above code retrieves the element with the id root from index.html and renders our main <App /> component inside it. This allows the entire React application to run within this div#root .
(Note: the new *createRoot API from React 18 is used here ; older Reacts used ReactDOM.render .)*
- **App.tsx file** – contains the main application component (usually the root of the component tree, typically including router configuration, page layout, etc.). The default template includes a simple counter example using useState (the counter's state) and a button to increment it. This

file also imports App.css with local style definitions for the component.
- **tsconfig*.json files** – (we'll discuss them in detail in the next chapter) – are responsible for configuring the TypeScript compiler. The generated project has several of them: the main tsconfig.json file and two specialized ones: tsconfig.app.json for the application code and tsconfig.node.json for tools running in Node (e.g., the Vite configuration file). By splitting the TS configuration into browser and Node parts, you can set different target JS versions or libraries for each part. Generally, however, you don't need to change much initially – the default configuration is modern and tailored to Vite.
- **vite.config.ts file** – this is the Vite tool configuration. It's very concise for a basic React application. By default, it contains something like this:

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

export default defineConfig ({
  plugins : [react()]
});
```

As you can see, the React plugin is hooked in here, providing **JSX/TSX** and **Fast Refresh (HMR) support for React** , as well as a few improvements (e.g., automatic import of react/jsx-runtime). Otherwise, Vite uses its default settings – it will automatically find index.html as the entry point, start the server on port 5173, support ES Modules , TypeScript , etc. In a typical project, you won't need to change anything unless you want to, for example, set aliases for imports, additional plugins, API proxies, or change the port – in which case you'll edit this file.

The overall structure of a Vite + React project is very **clean and minimalistic** . Everything we write goes to src/ , static files go to public/ . Configurations are kept at the top (the main folder) and usually don't require major changes at startup, allowing us to jump right into coding functionality.

Configuration files: **vite.config.ts** and **tsconfig.json**

Let's take a closer look at two key configuration files in our project:

vite.config.ts – Vite configuration

As mentioned, the default vite.config.ts file is minimalistic. Using defineConfig , we export an object with Vite options. Key elements: - **Plugins:** In the plugins section We're plugging in the official @vitejs/plugin-react plugin (for React projects). This plugin automates **JSX/TSX support** (that is, it integrates Babel with the appropriate JSX transformation), enables **fast refresh** (HMR) for React components, and sets a new JSX runtime (the so-called **automatic runtime** for React 17+), so we no longer need to import React at the top of each JSX file. In other words, the plugin ensures that JSX and React syntax work in Vite out-of-the-box. If we were building a project in another framework, there would be other plugins here (e.g., @vitejs/plugin-vue for Vue). - **Resolve/Alias:** If we want to set **path aliases** (more on that in a moment), we can do so in this

configuration, e.g., via the resolve.alias field . By default, Vite doesn't have aliases set (apart from '/@fs/' for filesystem access, which is rarely used). - **Server options:** In vite.config, you can configure the behavior of the development server (server section). For example, you can change the port: server: { port: 3000 } if 5173 conflicts. You can also set proxy, HMR options, open (browser auto-opening), etc. - **Build options:** Options for the build process (build section) – e.g., changing the target JS version, customizing chunk-split, enabling source map generation for production, etc. - **Env, CSS, and more:** Vite.config also allows you to configure handling of environment variables, loading CSS files (e.g., preprocessors), integration with tests, SSR (if you're using Vite for SSR), etc. However, in a basic project, we don't need to touch this.

A huge advantage of Vite is that **for standard applications, the configuration is already there . As the documentation states, you usually don't need to modify** vite.config.ts at all unless you have a specific need – you can focus on the application code.

tsconfig.json – TypeScript compiler configuration

TypeScript requires a tsconfig.json configuration file , which specifies, among other things, the language version, target JS standard, type behavior, supported environment libraries (DOM, ESNext), and many other options. In the Vite + React project (react-ts template), we'll notice that we have three files: tsconfig.json , tsconfig.app.json , and tsconfig.node.json . This structure divides the settings into:
- *base* (tsconfig.json),
- *frontend* (tsconfig.app.json),
- *tools (Node)* (tsconfig.node.json).

The main tsconfig.json acts as a container, pointing to references to the other two configs. This allows TypeScript to build the frontend and utility components separately, improving compilation times and streamlining dependencies.

tsconfig.app.json file contains the appropriate settings for the React code: - target : sets the target JavaScript version (e.g., ES2020) – this determines which TS constructs to transpile and which to leave behind. Vite and esbuild may override some things during bundling anyway, but we're generally aiming for modern browsers. - lib : specifies the available libraries in the environment – probably **DOM** , ES2020, etc., as we're writing a browser app. - jsx : set to "react-jsx" – this is important because it means using the new JSX transform. This saves **us from having to import React in every file** ; JSX functions are automatically injected by the compiler (this is new since React 17). - module : usually "ESNext" – as we're using ES modules. - **Bundler mode:** You can see the "moduleResolution" option : "bundler" – this is a new TS option that makes TypeScript understand imports like bundler (e.g., it allows importing .ts / .tsx extensions without errors, supports aliases , and other bundler features). - isolatedModules: true and noEmit: true : these are requirements and optimizations for working with Vite. noEmit prevents TypeScript from generating separate .js files during compilation – this task is handled by the bundler (Vite). Thanks to isolatedModules , TS will warn us if we have used functions incompatible with transpiling individual modules (e.g., certain features like const enum – Vite/esbuild won't handle them). - Various **strict options** : e.g., strict: true , noUnusedLocals , noUncheckedSideEffectImports ,

etc. – these are TypeScript good practices, enabling more rigorous code checking. The default template has them enabled, which encourages writing more secure code.

tsconfig.node.json, on the other hand, applies to files that are run by Node (e.g., vite.config.ts , possibly tests, scripts). Here, lib can be set to a newer one (ES2023) and include the Node API , and target to ES2022, etc. It's crucial that moduleResolution is also set: bundler , noEmit , etc., to maintain consistency with Vite's approach. In other words, the Vite configuration file and other tools are also compiled with TS, but with separate parameters tailored to the Node environment (e.g., **we don't need the DOM in the library** , but we can use newer ES features).

to set path aliases for TypeScript , for example , we need to edit tsconfig.app.json (and optionally tsconfig.node.json) and add a "paths" section there – more on that in the next section. Initially, however, these files don't require any intervention from us, as they've been designed to **work well with TypeScript** and modern JavaScript.

Typical workflow – creating and running a project (Vite + npm)

Working with a Vite project is very convenient. We'll now show you a typical **workflow** : from project initiation, through launching the development server, to eventually building a production application.

1. **Creating a new project:** The easiest way is to use the official wizard. In the terminal, navigate to the desired folder and run the command:

   ```
   npm create vite@latest my-app -- --template react -ts
   ```

   The above command will create a my-app directory with a ready-made React application framework written in TypeScript (the --template react-ts flag selects a template). The Vite wizard can interactively ask for the project name and type – we provided them in the above command. Once the scaffolding is complete, the messages will instruct you to navigate to the folder and start the project.

2. **Installing dependencies:** Go to the newly created folder and install npm packages:

   ```
   cd my-app
   npm install
   ```

3. **Starting the development server:** After successful installation, we can run the application in developer mode:

   ```
   npm run dev
   ```

   By default, Vite will open a local server at http://localhost:5173/ (or another address if 5173 is already taken). You'll see a launch message in the console, along with the local and network addresses. Now you can open a browser and see the app running—it

should display a simple page with the Vite/React logo and a counter (this is the content of the default App.tsx).

4. **Working on the code:** During development, we edit files in src/ – upon saving, Vite will automatically refresh the appropriate modules in the browser thanks to HMR. There's no need to manually reload the page. The workflow is iterative: code -> save -> immediately see the effect in the browser window. If an error occurs in the code, the browser console (or Vite terminal) will display an appropriate error message, facilitating quick fixes.

5. **Building the production version (optional at the end):** When the application is ready to be deployed, we execute the command:

   ```
   npm run build
   ```

   This will trigger the build process (Vite will roll up the entire project to static files). The result is a dist/ directory containing minified HTML, JS, CSS, and images, ready to be uploaded to the server. Vite will take care of optimizations—e.g., **minifying code, tree-shaking unused modules, chunking, and** so on. You can also run a local server to test the build:

   ```
   npm run preview
   ```

   which will start a server serving files from dist (useful for checking that everything works after building).

To summarize, working with Vite typically boils down to: **npm run dev while writing code, and npm run build before deployment** . Thanks to fast HMR and minimal configuration, the workflow is very smooth – you can focus on implementing functionality instead of configuring tools.

JSX and TSX Basics – Functional Component with Types

JSX (JavaScript XML) is a syntax that allows you to write HTML-like code directly in JavaScript. In React, we use JSX to define the user interface within our components. **TSX** is a JSX extension for TypeScript – .tsx files allow you to use JSX with type checking. Let's explore the basics through a simple component.

In React, the dominant approach is to write **functional components** . Here's an example of such a component in TypeScript:

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

interface HelloProps {
name: string;
}

const Hello: React.FC<HelloProps> = ({ name }) => {
```

```
const [count, setCount] = useState<number>(0);

return (
<div>
<h1>Hello, {name}!</h1>
<p>The {count} button was clicked.</p>
<button onClick={() => setCount(count + 1)}>
Click me
</button>
</div>
);
};

export default Hello;
```

The code above highlights several important elements of **JSX/TSX syntax** and type usage: - **JSX elements:** In the component's return block, we return a JSX structure that resembles HTML: we have , <h1> , <p> , <button> tags . In JSX, we can use **any HTML tags** (the browser will end up with plain HTML anyway). We can also create our own components and use them as tags (e.g., <MyComponent />). - **Embedding variables/expressions:** Within JSX, we use curly braces { ... } to embed dynamic JavaScript values. For example, {name} will be replaced with the value of the name variable , {count} with the value of the count state . We can execute simple expressions, call functions, etc. there. - **Typed props: Our** Hello component accepts a prop name (the user's name) defined in the HelloProps interface . With React.FC<HelloProps> , we indicate that Hello is a functional component that expects props compatible with HelloProps . As a result, if we try to use <Hello> without passing a name or with the wrong type (e.g., a number instead of a string), TypeScript will throw a compilation error. Typing props allows us to catch many errors while coding. - **UseState Hook with Type: We used** the useState<number> Hook to create the count state as a number. UseState returns the tuple [value, setter] . Thanks to <number>, the compiler knows that count is a numeric type, and setCount expects a number. If we tried, for example, setCount("text") , TS will signal an error. React **Hooks (e.g.,** useState , useEffect , useContext) have generic types, which makes work much easier – the state type is automatically recognized based on the initial value, or we can explicitly specify it, as above. - **Inline Events and Functions:** In JSX, DOM events can be assigned declaratively. For example, in <button onClick={() => setCount(count + 1)}> we passed an arrow function to the onClick attribute . React will make sure that this function is called on click. We increment the count state by 1. Note: we don't use addEventListener – React manages events itself. Additionally, TypeScript knows the event type (onClick -> MouseEvent etc. if we need a parameter). - **React Import (note):** In new projects (React 17+) *we don't need to import* React in every JSX file, as long as we're using the new JSX runtime (which is achieved thanks to the tsconfig "jsx": "react-jsx" setting and the Vite plugin). In our example, we're only importing React for the React.FC type itself , but you could skip that and use the FC type from the React library differently. Older projects always required importing React from 'react' at the top – this isn't necessary these days, but you can import it for readability or type-awareness.

This example shows the basic structure of a component: definition (here via const Hello = ...), any **state/effect hooks inside** , and returning a UI in JSX. Components can also be defined as regular

functions (e.g., function Hello(props: HelloProps) { ... }), the effect is the same. The important thing is that **JSX must return a single root DOM element** – that's why we often wrap multiple elements inside a or use the special <>...</> syntax (React fragment) to avoid adding an unnecessary wrapper.

TSX ensures that our component and JSX are type-correct. For example, if we try to insert a component with an invalid prop, TS will warn us. JSX attributes are also typed – for example, onClick for <button> expects a function with the signature (event: MouseEvent<HTMLButtonElement>) => void , so assigning anything else would be an error. This makes working with React+TSX more secure – we detect many errors before the application runs.

To summarize: **JSX** allows you to write readable interfaces by combining the power of HTML and JavaScript, while **TypeScript (TSX)** adds type checking, which is invaluable in larger applications. In your daily work, it's worth familiarizing yourself with the typing of components, props, states, and event functions – this will increase code reliability.

Module import and export, path aliases (e.g. @/components , @/hooks)

JavaScript has supported modularity since ES6 – we can divide code into modules and use import / export to organize the project. In a React application, we typically have dozens or hundreds of modules (each component is a module, plus supporting modules). Managing them becomes crucial for code readability.

Export/import basics:
- We can **export** functions, variables, classes from one file and then **import** them in another. For example, if we write export function formatDate(d: Date) { ... } in the utils.ts file , then in another file we can import { formatDate } from './utils' to use it.
- There are two types of exports: **named** (export const X = ...; export function Y() {...}) and **default** (export default Component). Named imports require the exact name within curly braces ({ X, Y }), while default imports are specified without curly braces and can be given any local name. For example: export default function App(){...} in App.tsx , and import: import App from '@/App' (assuming an alias or relative path).
In React, we usually export components by default (e.g., each component file exports one component), while utility functions, types, etc., are exported by name. This division allows us to easily import multiple things from a single module when needed. Web bundlers (including Vite) and TypeScript support ES Modules syntax, so we can use import / export without any problems. It's worth avoiding the archaic require() method (it's CommonJS, not natively compatible with browsers).

Path aliases:
In a larger project, it's very helpful to configure aliases for import paths. Instead of writing complex relative paths like import MyComp from '../../../../components/MyComp' , we can define a shortcut, e.g., @ for the src directory . This will make the import cleaner: import MyComp from '@/components/MyComp' . Thanks to aliases: - Refactoring and moving files is easier (we don't have to correct multiple levels of ../). - The code is more readable – it's immediately clear that

something is being imported from our root directory (and not, for example, from node_modules).

How do I set an @ alias in Vite + TS? It's a two-step process: 1. **Configuring Vite:** In the vite.config.ts file , add the resolve.alias option . We need to import the Node path module and use it to set an absolute alias. Example :

```
import { defineConfig } from ' vite ';
import react from '@ vitejs / plugin -react';
import * as path from 'path';

export default defineConfig ({
  plugins : [react()],
resolve: {
alias: {
'@': path.resolve (__ dirname , './ src '),
'@components': path.resolve (__ dirname , './ src /components'),
'@hooks': path.resolve (__ dirname , './ src /hooks')
}
}
});
```

Above we mapped @ to the src folder and for example also sub-aliases @ components , @ hooks . From now on, Vite knows when bundling that, for example, importing from '@/utils/api' should look for the src/utils/api.ts file .

1. **TypeScript configuration:** Vite itself will bundle aliases correctly, but for **TypeScript and our editor** to understand these paths (otherwise we would get "Cannot find module '@/...'" errors), we need to add appropriate entries in tsconfig.app.json (and similarly in tsconfig.node.json if aliases are also used, e.g., in tests or configs). In tsconfig, we use the paths field with the "baseUrl": "." (or "baseUrl": "src") setting. Configuration example:

   ```
   {
   " compilerOptions " :
   {
   " baseUrl "
   : " .
   ```

   ```
   "./tsconfig.node.json " }]
   }
   ```

This snippet means: when you have an import in your code that starts with @/ , it should be treated as a relative path specified in the array (i.e., src/*). This will ensure that both **the TS compiler and** the editor's Intellisense will recognize aliases.

After these steps, we can confidently write, for example, import LoginForm from '@/components/LoginForm' everywhere in the project, instead of worrying about how many ../ we need . This is especially practical when frequently importing things from deeply nested directory structures.

@ alias (often used to refer to src in web projects), you can define other aliases—e.g., components , hooks , api , styles , etc.—at your discretion to group imports. It's important to use this consistently and remember to update both vite.config and tsconfig.

Note: Some Vite libraries offer plugins that automate aliasing using tsconfig, but manual configuration, as described above, is simple and works flawlessly. Don't overdo it with aliases either – too many can be confusing. Typically, 1-3 aliases in a project are sufficient (usually @ for src , and possibly separate ones for frequently used module subnets).

Browser Debugging and React DevTools Extensions

Even the best developers sometimes write buggy code, so effective debugging skills are crucial. For front-end applications, we have powerful tools available in the browser itself, as well as additional extensions, especially useful for React.

1. Built-in browser development tools (DevTools):
Every modern browser (Chrome, Firefox, Edge) has a Developer Tools panel (activated, for example, by pressing F12). The following are particularly useful for debugging React: - **JavaScript Console:** This is where all logs (console.log), warnings, and runtime errors are stored. If our code throws an exception, we'll see a red error message and a stack trace (call path). The console also allows us to interactively execute JS commands in the context of the current page, which can be helpful for quick tests (e.g., inspecting an object, calling a global function). - **Elements Panel (or Inspector):** This shows the current **DOM structure and** page styles. While React's DOM tree is the result of component rendering, in practice we often need to check, for example, whether an element has the correct CSS class or style, or whether it even exists in the DOM. This panel allows you to select an element on the page (using a selector or clicking in the interface) and preview its CSS properties, box model, and modify them on the fly. This helps debug issues with the appearance (CSS) or HTML structure generated by the application. - **Sources Panel:** This is where you can view JS/TS sources (thanks to Vite's sourcemaps, you can see the TypeScript/TSX code almost as written) and, most importantly, set **breakpoints** . A breakpoint allows you to stop code execution at a given point and analyze the application's state (variables, calls) step by step. This is often indispensable for debugging logical errors. For example, you can open a component file, find the line of interest (e.g., in the handleSubmit function), and add a breakpoint. When the code reaches it, execution pauses, giving you control (step-by-step execution, value preview). The Sources panel also offers tools for debugging asynchronous code (e.g., events, timers) and various improvements, such as blacklisting libraries (to avoid accessing them). - **Network panel:** Useful when the application communicates with the server (e.g., fetch

API). Here, you can see what HTTP requests are being made, their results, parameters, headers, and responses. If, for example, data isn't loading, the Network panel allows you to check whether the request was successful, whether there was an error (404/500), and whether the response is correct. - Other panels, such as **Application** (storage, e.g., LocalStorage, cookies) or **Performance** (speed profiling), can also be used, but less frequently for basic debugging.

Thanks to Vite's browser integration, **sourcemaps** are generated by default, so in the Sources panel we will see the original TSX code instead of the transpiled JS – this makes it easier to track errors precisely in our code.

2. React Developer Tools extension:
Standard DevTools doesn't "know" anything about our React components—they only see the pure DOM and JavaScript. That's why the React team created a dedicated tool: **React Developer Tools** . This extension is available for Chrome, Firefox, and Edge (installable from browser stores). After adding it, **two new tabs will appear in DevTools** : - **Components:** Lets you browse the hierarchy *of actual React components* , regardless of the DOM. Here, you can see a tree like the one we create from components, with component names (e.g., <App> , <Header> , <MyButton> , etc.). Clicking on a component displays the details: *props* passed to it, internal *state* (if it uses useState or state in a class component), and, if applicable, context (Context) or hooks. This makes it much easier to understand the current state of the application. For example, you can find a component responsible for displaying a product list and check what properties it has received and what its current state is (this helps find the source of an invalid value, for example). What's more, React DevTools allows you **to edit props and state in a component** live , which will instantly update the view. This is great for testing component behavior without changing the code (e.g., change prop count: 5 to 10 and see if the count updates). - **Profiler:** The second tab is for profiling rendering performance. You can record interactions and see which components are rendering, how long it takes, etc. In the context of this lecture, let's just mention that it exists – it's an advanced tool for diagnosing potential performance bottlenecks.

React Developer Tools is absolutely **essential when working with React** – it allows you to see "deep inside" your application. It makes debugging much faster because you don't have to guess what React is thinking – you can actually see it. It's worth noting that React DevTools automatically detects React applications. Once installed, as soon as you launch your React website (in dev or prod mode – it doesn't matter), the extension will recognize and activate. According to the official materials, **React DevTools allows you to inspect components, edit their props/state, and identify performance issues** . Installation is as simple as installing the extension and refreshing the page; for Chrome/Firefox, it's a single click in the store.

Tip: Besides React DevTools, there are other useful extensions for debugging: e.g. if we use Redux for global state – *Redux DevTools* , if we have routing libraries – some have their own inspectors, etc. But for starters, standard browser tools plus React DevTools cover most needs.

In summary, front-end debugging is all about **observation** (what's happening? What errors are in the console?), **inspection** (viewing the DOM, components, state), and **interaction** (breakpoints, live modification). Mastering DevTools and React DevTools will significantly increase the efficiency of your application's troubleshooting.

Practical example: creating a project, a component with hooks, adding a UI library

Now, we'll walk through a concrete example, combining many of the topics discussed. We'll show you how to create a new React project with Vite, how to implement a simple component using the useState and useEffect hooks , and then how to integrate UI libraries into such a project—using **Tailwind CSS** (a CSS framework) and **Ant Design** (a React component library) as examples.

Creating a new Vite + React (TypeScript) project

1. **Project Initialization:** Let's use the create-vite command again. In the console, execute:

   ```
   npm create vite@latest my- vite -app -- --template react- ts
   ```

   This will create the my-vite-app folder with the finished project. If the command prompts you interactively, select React and TypeScript . Once the process is complete, navigate to the directory:

   ```
   cd my- vite -app
   npm install
   ```

   (installing dependencies). Then run: npm run dev to check if everything works. Vite should start the server at http://localhost:5173. Open this page – you should see the default content (the "Vite + React" text, the Vite and React logos, and a counter button). This means our environment is ready.

2. **Structure Overview:** In your code editor (e.g., VSCode), let's look at the file structure. You should see the layout discussed earlier—configuration files, a src folder with main.tsx and App.tsx , etc. App.tsx contains the default counter code, as follows:

   ```
   function App() {
   const [count, setCount ] = useState (0);
   return (
   <>
   <h1> Vite + React</h1>
   <div className ="card">
   <button onClick ={() => setCount ((count) => count + 1)}>
   count is {count}
   </button>
   <p> Edit <code> src /App.tsx</code> and save to test HMR</p>
   </div>
   </>
   );
   }
   ```

This code is a simple component with a counter (count state incremented on click). Here's an example of using useState and handling the onClick event, as discussed. Let's try modifying this code or adding a new component.

A simple component using **useState** and **useEffect**

We'll now demonstrate creating a custom component that, for example, displays the current time and updates every second. We'll use the **useEffect hook for this** .

1. **Adding the Clock component:** In the src/ folder, create the Clock.tsx file . In it, we'll write the function component:

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

function Clock() {
const [time, setTime ] = useState (new Date());

  useEffect (() => {
const timer = setInterval (() => {
    setTime (newDate());
}, 1000);
// cleanup function:
return () => {
    clearInterval (timer);
};
}, []); // empty dependency - effect starts myself once by mount

return <div> Current time : { time.toLocaleTimeString ()}</div>;
}

export default Clock;
```

What's going on here:

2. We initialize the time state with the current date (new Date()).
3. We use useEffect with an empty dependency array [] , so the effect will only fire **once** the component is mounted. Inside the effect, we set an interval (every 1000ms, or every second) and update the time state with the new date. This will rerender the component every second, refreshing the displayed time. We also return a cleanup function – clearInterval(timer) – which will be called when the component is unmounted (this prevents interval leaks).
4. In the renderer, we return a simple displaying text with the current time, formatted to a readable time using toLocaleTimeString() .
5. This component demonstrates the **useEffect method** for performing a side effect (setting an interval) and clearing it. This is a common pattern for things like subscriptions, timers, event listeners, and so on.

6. **Integrating the component into the app:** Now, to see our clock, let's place it in the main view. Open App.tsx and import Clock at the top:

   ```
   import Clock from './Clock';
   ```

   Then in JSX (e.g., under existing elements), let's add <Clock /> . For example:

   ```
   <div className="card">
   ... {/* existing counter code */}
   </div>
   <Clock />
   <p className="read-the-docs">Click on the Vite and React logos to learn more</p>
   ```

   (We inserted <Clock /> before the line with <p className="read-the-docs"> .) After saving the file, thanks to HMR, the page in the browser will automatically update. We should now see an additional element with the current time, changing every second.

Verification: If the time is refreshing, everything is working. We can open the devtools console and in the Components tab (React DevTools), make sure that our <Clock> component is mounted and its time state is changing. Alternatively, add console.log to setInterval to log the time – we'll see it in the console every second, confirming that the effect is working.

1. **HMR Demonstration:** We can also test Hot Module Replacement. Let's open the Clock.tsx code and, for example, change the text "Current time" to "Time now." Note: In a split second, the page should refresh only this fragment (without losing application state; for example, the count counter in the App will remain the same). HMR saves us from constant manual refreshes.

This way, we added our own component and used key hooks. **useState** provided the state (time), **and useEffect** provided the side effect (interval). This pattern is very common: useEffect integrates with a browser API or external resources, and upon state change, rerendering occurs. In React, these mechanisms allow for much more complex functionality, but the concept remains similar.

Adding a UI library – Tailwind CSS or Ant Design

Front-end development often uses libraries that speed up UI creation – whether CSS libraries like **Tailwind** or ready-made component collections like **Ant Design** , **Material-UI** , **Bootstrap** , etc. We will briefly show you how to integrate two different types of solutions:

Tailwind CSS *– utility-first CSS framework*

Tailwind is a very popular CSS framework that provides hundreds of ready-made CSS classes (so-called utility classes) that handle individual styling aspects (e.g., bg-red-500 for a red background, text-center for centered text, p-4 for padding, etc.). Instead of writing your own CSS classes,

Tailwind styles them directly using these ready-made classes in the className attributes of elements.

To add Tailwind to our project (React + Vite), we follow these steps: 1. **Installing dependencies:** In a Vite project, installing Tailwind is simple. We execute the following in the terminal:

```
npm install -D tailwindcss postcss autoprefixer
```

(The -D flag means we're adding to devDependencies, as these tools are only needed during build time.) This will install Tailwind and its requirements: PostCSS (the CSS processing engine) and Autoprefixer (adds prefixes for CSS compatibility).

1. **Initializing the configuration:** After installation, run the command:

   ```
   npx tailwindcss init -p
   ```

   This will generate two configuration files: **tailwind.config.js** and **postcss.config.js** . The first is used to customize Tailwind (e.g., add your own colors, themes), the second tells PostCSS which plugins to use (by default, it will add an entry for tailwindcss and autoprefixer).

2. **Configuring paths (content):** Let's open tailwind.config.js . We need to tell Tailwind which files to search for CSS classes used in the project (called purge or content paths). This way, Tailwind knows which styles are actually being used and can remove unused ones (called tree-shaking CSS) during build to reduce the size of the resulting CSS. In the file, we add:

   ```
   export default {
   content: [
   "./index.html",
   "./ src /**/*.{ js,ts,jsx,tsx }"
   ],
   theme: {
   extend: {},
   },
     plugins : [],
   }
   ```

   that is, we add paths to all files where Tailwind classes may appear – in our case, all .html and .jsx / .tsx files in src . We save the file.

3. **Adding Tailwind CSS directives:** Tailwind works through special directives in the CSS file, which are expanded to the full framework content during build. Typically, you'll use a single main CSS file (e.g., index.css). Our project already imports src/index.css in main.tsx .

Let's open index.css and add three lines
to the top of the file :

@tailwind base;
@tailwind components;
@tailwind utilities;

This will cause Tailwind to insert the following into the build locations: reset/normalize styles (base), predefined components (optional, e.g., ready-made classes like .container), and a full range of utilities . If index.css had any of our definitions, you can leave them above or below – the order of these directives matters (base -> components -> utilities, according to the documentation).

4. **Importing CSS into the project:** In our case, the index.css file is already imported in main.tsx (import './index.css';), so we don't need to do anything else. When we run npm run dev , Vite will automatically apply PostCSS + Tailwind. We can test this: let's try using a Tailwind class in our code. For example, in App.tsx, let's change the <h1>Vite + React</h1> heading by adding the Tailwind classes:

<h1 className ="text-3xl font-bold underline text-blue-600"> Vite + React</h1>

Let's save – the page will refresh. If Tailwind is working, our header should now have larger, bold, underlined blue text. (Classes: text-3xl – extra large font size, font-bold – bold, underline – underline, text-blue-600 – blue from the color palette).

We can also open the dev tools and see in the Elements -> Computed Styles tab that this h1 has CSS rules assigned from Tailwind. This will confirm the integration.

1. **Done!** Now we can use Tailwind's entire arsenal of utility classes in our components. For example, <button className="bg-green-500 text-white py-2 px-4 rounded-lg hover:bg-green-600">Click</button> will create a beautiful green button with rounded edges and a hover effect. We don't need to write a single custom CSS class (though of course you can; Tailwind doesn't prohibit it).

Tailwind significantly speeds up styling, especially UI prototyping. However, it requires some getting used to (plenty of classes in markup, instead of the traditional HTML+CSS split). In our project, integration went quickly because Vite and PostCSS work well with Tailwind. **It's worth noting** that sometimes the latest version of Tailwind can have integration issues – the aforementioned article suggested installing version 3.4.1 if there were problems, but we assume these issues are temporary. In general, the Tailwind documentation thoroughly describes the integration process with Vite, and it's similar to the one above.

Ant Design *– React Component Library*

Ant Design (AntD for short) is a complete UI framework that provides ready-made, styled React components: from buttons, through forms, tables, modals, and entire view sets. Using such a library saves time – instead of writing your own component and styles, you use a pre-built one.

Adding Ant Design to your React + Vite project is also simple: 1. **Installing the library:** We do:

```
npm install antd
```

(without -D , this goes to dependencies, as it will be used in the application code). The antd package contains both React components and the necessary styles (in AntD 5, styling is implemented via CSS-in-JS).

1. **Using the AntD component:** Once installed, we can immediately import components and use them in our JSX. For example, let's try using a simple **Button component** from Ant Design. Let's edit App.tsx :
2. At the beginning of the file let's add: import { Button } from 'antd';
3. Inside return, perhaps instead of our current <button> we will use <Button> :

   ```
   <Button type="primary"> Button AntD </Button>
   ```

   AntD exports its components with ready-made styles. The type="primary" attribute will highlight the button with the primary accent styles. After saving, the styled button should appear in our application (by default, AntD produces a blue "Primary Button"). If so, it means the integration is working. It's worth noting that **we didn't manually import any CSS in the code** – Ant Design version 5+ uses the CSS-in-JS mechanism and **automatically includes styles** using components. In older versions (AntD 4 and earlier), you had to import a global CSS file (e.g., antd/dist/antd.css), but starting with v5, including a ready-made CSS file has been discontinued. The only thing we suggest adding is the so-called reset: antd/dist/reset.css , if you need to reset the browser's default styles to the AntD style. We can do this by adding an import of 'antd/dist/reset.css'; in main.tsx, right next to the index.css import . However, this is not strictly required – the components will display without it, but resetting can unify the appearance (e.g. fonts, margins) with the AntD assumptions.

4. **Using Components:** Ant Design has extensive documentation. For example, to use the **DatePicker component** , simply:

   ```
   import { DatePicker } from ' antd ';
   ...
   < DatePicker />
   ```

   And we get an elegant date picker. All the logic and styling are built in. The only thing you might need to do is **adjust the styles** to suit your branding. AntD allows for theming (e.g., changing primary colors) via CSS variables or less, but that's a topic for a separate lecture. In our context, it's important that integration is fast.

5. **Checking in the browser:** After adding, for example, an AntD button, it's worth taking a look at DevTools Elements – you'll see that the button's structure is actually a rather

complex DOM with classes, and in the Styles panel you can see the AntD styles (injected as <style> in the head). React DevTools will show <Button> as a composed element, although internally it's a complex mechanism.

Literature

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