# 5 React

## CS 425
## Web Applications Development

Alex Petrovici

## Content

## Introduction /

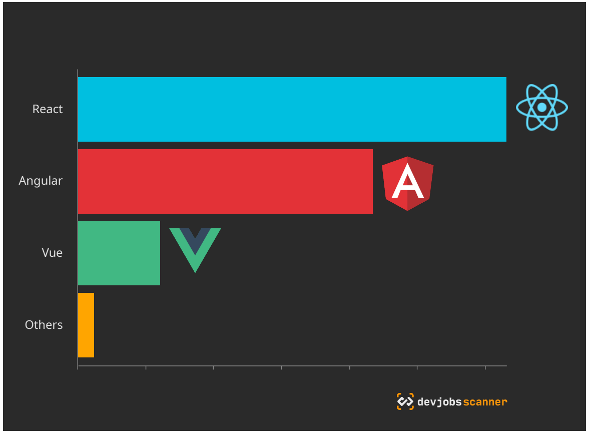
### What is React ?

- React (Meta) is a library made for building *web and native user interfaces*. It uses Hypertext Markup Language (HTML) (via JavaScript XML (JSX)) and manipulates the Document Object Model (DOM) through JavaScript in a browser.
- React Native (Meta) translates React components to mobile apps (Android, iOS).
- React Native Windows (Microsoft) translates React components to Windows software.
- React Native macOS (Microsoft) same but to macOS.

Some React examples:

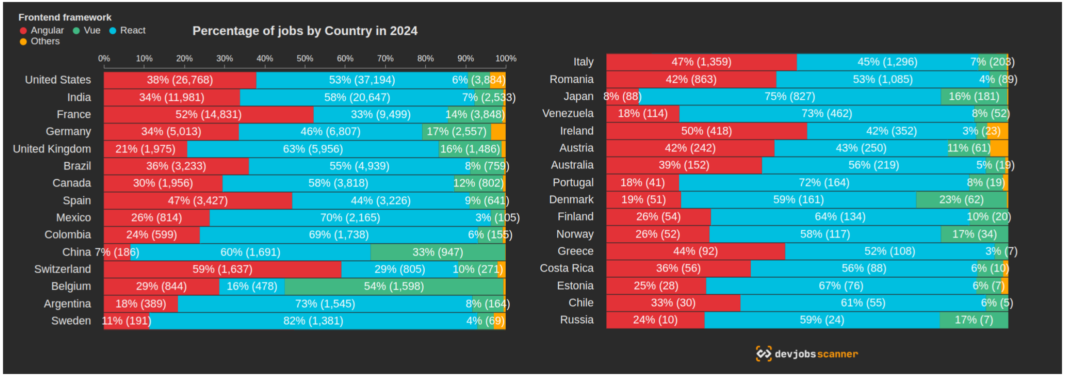
- Facebook
- MS Office, Outlook, Teams, Xbox, Skype
- Discord, Netflix
- Other React Native applications

## Introduction /



Source: devjobsscanner

## Introduction /



Source: devjobsscanner

## First React App / Backend

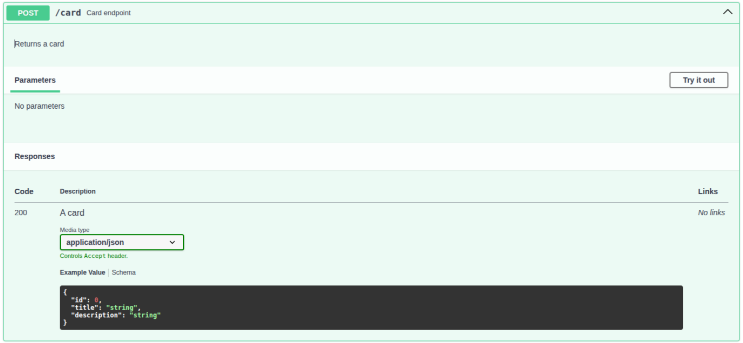
- The frontend (React) is responsible of the endpoints that returns HTML.
- The backend (FastAPI) is responsible of the endpoints that return or transforms data and has access to the database.

**backend/app.py**

```python
from fastapi import APIRouter, FastAPI
from pydantic import BaseModel

class Card(BaseModel):
    id: int
    title: str
    description: str
...
@app.post("/card")
async def get_card(request: Request) -> Card:
    data = await request.json()
    _id = data.get("id")
    return Card(id=_id, title=f"Card {_id}", description=f"Card {_id} description")
```

## First React App / Backend

This is the endpoint that will return the card data:


POST /card Card endpoint

Returns a card

Parameters

Try it out

No parameters

Responses

| Code | Description | Links |
|---|---|---|
| 200 | A card | No links |

Media type
application/json
Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "description": "string"
}
```

This will be rendered by the react app in the page 28.

## First React App / React Initialization: Install Node.js

We will use **Vite** to create the React application, and **Vite** requires **Node.js** version 20.19+ or 22.19+. Let's begin by ensuring we have the correct version of **Node.js** installed.
NOTE: Install from source to ensure that we have the latest version: nodejs.org
Get **Node.js v25.0.0 (Current)** for **Linux** using **nvm** with **npm**:

```
# Download and install nvm:
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash

# in lieu of restarting the shell
\. "$HOME/.nvm/nvm.sh"

# Download and install Node.js:
nvm install 25

# Verify the Node.js version:
node -v # Should print "v25.0.0".

# Verify npm version:
npm -v # Should print "11.6.2".
```

## First React App / React Initialization

Initialize the React app from the project root.
More documentation about starting a Vite project: https://vite.dev/guide/

```
❱ npm create vite@latest

> npx
> "create-vite"

│
◇  Project name:
│  frontend
│
◇  Select a framework:
│  React
│
```

## First React App / React Initialization

Choose SWC (Speedy Web Compiler) for the compiler since is much faster than the default (Babel) because SWC uses Rust.

```
◇  Select a variant:
│  TypeScript + SWC
│
```

## First React App / React Initialization

```
◇  Use rolldown-vite (Experimental)?:
│  No
│
◇  Install with npm and start now?
│  No
│
◇  Scaffolding project in /home/alex/projects/siu/cs425/examples/react/frontend...
│
└  Done. Now run:

cd frontend
npm install
npm run dev
```

## First React App / React Initialization

Now let's finnish the installation and start the development server.

```
❯ cd frontend
❯ npm install

added 150 packages, and audited 151 packages in 11s

44 packages are looking for funding
run `npm fund` for details

found 0 vulnerabilities
```

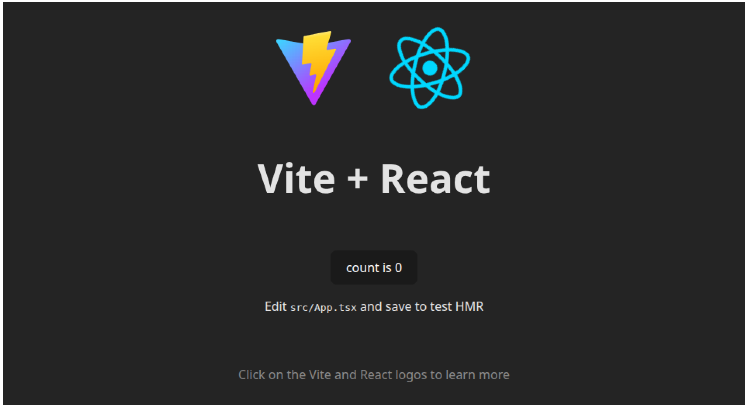
## First React App / React Initialization

Then run the front server.

```
❯ npm run dev

> frontend@0.0.0 dev
> vite


  VITE v7.1.12  ready in 143 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

## First React App / React Initialization

You should see the following output in your browser:



Edit `src/App.tsx` and save to test HMR

Click on the Vite and React logos to learn more

Let’s analyze what we just did:

## First React App / React Initialization

Let's see the content of /frontend

```
└── 🗀 frontend
    └── 🗀 public                    -- Static assets that will be served directly
    └── 🗀 src                       -- Source code
        └── 🗀 assets                  -- Static assets
        ├── App.css                    -- CSS file
        ├── App.tsx                    -- 2. The React component that will be rendered.
        ├── index.css                  -- CSS file
        ├── main.tsx                   -- 1. Then the react app enters here.
    ├── index.html                 -- 0. Browser loads index.html, then loads main.tsx.
    ├── package.json               -- Dependencies (like pyproject.toml but for npm)
    ├── package-lock.json          -- Package lock file (like poetry.lock)
```

## First React App / React Initialization

1. The client asks for **index.html**, which is loaded by the client.

```
...
<div id="root"></div>
<script type="module" src="/src/main.tsx"></script>
...
```

```
We initialized an empty div with the id "root" in the index.html file.
```

## First React App / React Initialization

2. Now the client asks for **main.tsx**.

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

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

```
JavaScript will find the element #root and will replace its content with
what will be rendered (client-side) with the output from the component App.
```

## First React App / React Initialization

3. The server returns a transpiled (see transpilation) version of **main.tsx**, which is executed by the client. During development, vite does on-the-fly transformation and serves the transformed JavaScript in real-time. In production it would return compiled (bundled, minified, static) JavaScript.

## First React App / React Initialization

**frontend/src/App.tsx**

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";
...
```

```
We have several imports:
    - useState is a React hook that allows us to manage the state of the component.
    - reactLogo and viteLogo are paths to images from the assets dir.
    - App.css from the same dir.
```

## First React App / React Initialization

**frontend/src/App.tsx**

```
...

function App() {
    const [count, setCount] = useState(0);

    return (
        ...
    );
}
```

```
We have a functional component.
It stores inside the value count and will be changed with the function setCount.
It has a return statement that returns a JSX element.
This is what React renders in the browser.
```

## First React App / React Initialization

**frontend/src/App.tsx**

```
...

return (
    <>
        ...
    </>
);
```

```
React components must return exactly one root element.
<>...</> is a Fragment and groups multiple elements without adding an extra node
to the HTML output.
```

JSX it’s not real HTML. The compiler translates it into pure JavaScript:

```
return React.createElement("h1", null, "Vite + React");
```

## First React App / React Initialization

**frontend/src/App.tsx**

```
...

return (
    <>
    <div>
        <a href="https://vite.dev" target="_blank">
            <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
            <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
    </div>
    ...
    </>
);
```

Two a href tags that uses the previously imported images. Style managed by App.css.

## First React App / React Initialization

**frontend/src/App.tsx**

```
...

return (
    <>
    ...
    <h1>Vite + React</h1>
    ...
    </>
);
```

```
Just a h1 element styled by examples/react/frontend/src/index.css
```

## First React App / React Initialization

**frontend/src/App.tsx**

```
...
    <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>
...
```

```
Here we have the main behaviour of the app.
onClick is a event handler that will be called when the button is clicked.
Then the html will be re-rendered with the new count value.
```

Hot Module Replacement (HMR) allows modules to update at runtime only the changed parts, avoiding a full refresh.

## First React App / React Initialization

### onclick

In vanilla HTML, writing inline handlers **onclick** like this is a bad practice:

```
<button onclick="alert('hi')">Click</button>
```

Mixes behavior directly into HTML, pollutes the global scope and makes debugging harder.

### onClick

React's **onClick** is a declarative prop. React doesn't inject a raw attribute into HTML. It attaches a JavaScript function to the virtual element in memory, using React's internal event system.
This is what React renders:

```
<button>count is 0</button>
```

## First React App / React Initialization

### frontend/src/App.tsx

```
...
function App() {
    ...
  }
export default App;
```

```
export default makes the App component the default export of this module.

This means when another file imports from this module without using curly braces,
they will receive the App component.

For example:
    import App from './App.tsx'              -- imports the default export (App component)
    import { App } from './App.tsx'          -- would look for a named export called App
```

## First Custom Component /

### Custom Component

Let’s implement a custom component that will display the card previously defined in the page 7.
We want to make

- a Card component that displays a single card,
- a CardHandler that retrieves data and displays it in a wrapper.

We may either start from the unitary behaviour of the Card component, or by starting from how we want the CardHandler to work. This time we’ll start from how we want the whole wrapper to work.

## First Custom Component / main.tsx

We start by updating the main entry point to import the future custom component:
**frontend/src/main.tsx**

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

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
    <CardHandler />
  </StrictMode>,
)
```

## First Custom Component / CardHandler.tsx

**frontend/src/CardHandler.tsx**

```
import "./CardHandler.css";
import { useState } from "react";
import Card, { type CardData, fetchCardById } from "./Card";

function CardHandler() {
    const [cardId, setCardId] = useState<number>(0);
    const [cards, setCards] = useState<CardData[]>([]);

    const getCard = () => { ... };
    return ( ... );
}
export default CardHandler;
```

```
The CardHandler is responsible for handling the user behaviour,
so it will allow the user to select the desired cardId,
then will fetch the content of the Card
and then it will append the content of the card wrapper in the UI.
```

## First Custom Component / CardHandler.tsx

Let’s start with the goal:
**frontend/src/CardHandler.tsx**

```
return (
  ...
    <div className="card-container">
      {cards.map((card) => (
        <Card key={card.id} cardData={card} />
      ))}
    </div>
  </div>
);
```

```
The elements will be appended to the card-container div.
```

## First Custom Component / CardHandler.tsx

**frontend/src/CardHandler.tsx**

```
return (
  <div className="card-handler">
    <div className="controls">
      <input type="number" placeholder="ID" value={cardId}
        onChange={(e) => { setCardId(Number(e.target.value)); }}
      />
      <button onClick={getCard}>Get Card</button>
    </div>
    ...
  </div>
);
```

```
The user selects the cardId with the number input, then clicks the button to get the card.
This will trigger the getCard function.
```

## First Custom Component / CardHandler.tsx

Let's define the function in CardHandler that is responsible for fetching the card data:
**`frontend/src/CardHandler.tsx`**

```
import Card, { type CardData, fetchCardById } from "./Card";
...
function CardHandler() {
    ...
const getCard = () => {
    fetchCardById(cardId).then((data) => {
      setCards((prevCards) => [...prevCards, data]);
    });
  };
}
```

```
getCard gets the data by calling fetchCardById,
then calls setCards which will update the cards state.

If you noticed, fetchCardById is defined in the Card component,
because the Card component is responsible for fetching instances of its own type.
```

## First Custom Component / Card.tsx

Let’s implement the Card component:

**frontend/src/Card.tsx**

```
import "./Card.css";

export interface CardData {
  id: number;
  title: string;
  description: string;
}

export const fetchCardById = async (cardId: number): Promise<CardData> => {};

function Card({ cardData }: { cardData: CardData }) {}

export default Card;
```

```
The interface CardData defines the structure we expect to receive from the backend.
The function Card is the component that will be rendered in the UI.
The const fetchCardById is the function that will get the data from the backend.
```

## First Custom Component / Card.tsx

Let’s implement what the user will see when the Card component is rendered:
**frontend/src/Card.tsx**

```
function Card({ cardData }: { cardData: CardData }) {
    return (
      <div className="card">
        <h3>Card {cardData.id}</h3>
        <h4>Title: {cardData.title}</h4>
        <p>Description: {cardData.description}</p>
      </div>
    );
  }
```

```
Just a div with some attributes.
```

## First Custom Component / Card.tsx

Let's implement the function that will get the data from the backend:

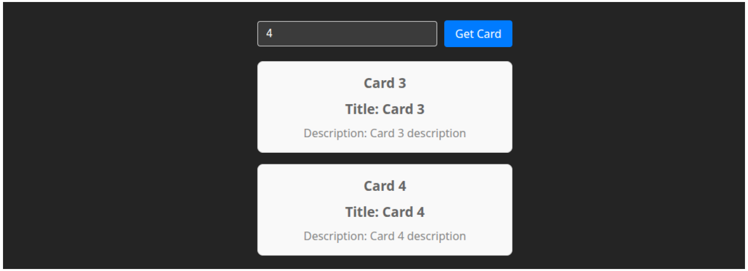
**`frontend/src/Card.tsx`**

```
export const fetchCardById = async (cardId: number): Promise<CardData> => {
    const response = await fetch(`http://localhost:8000/card`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ id: cardId }),
    });
    const responseData = await response.json();
    return responseData;
  };
```

```
The function fetchCardById is asynchronous because
it may take some time to get the data from the backend.
It returns a Promise that resolves to the CardData object.
This means that the function expects to return a valid CardData object at some point.
```

## First Custom Component / Card.tsx

You should see the following output in your browser:



Card 3
Title: Card 3
Description: Card 3 description

Card 4
Title: Card 4
Description: Card 4 description

You may update the cardId input and then click on the button to get the card from the backend.

## Using multiple endpoints / Dependencies

Install dependencies:

```
cd examples/react/frontend
npm install react-router-dom
```

## Using multiple endpoints / Dependencies

**main.tsx** will remain our entry point and will allow to render the different components.
We will use **<Routes>** to define the different endpoints of our React application. Each **<Route>** is accesable from an endpoint and will return a **Component**.
React-router-dom is a routing library for React. It manipulates the browser's history API and keeps your UI in sync with the current URL. It gives you **<BrowserRouter>**, **<Routes>**, **<Route>**, **<Link>**, **<Navigate>**, and a state machine that decides which component to render for a given path.
Npm package `https://www.npmjs.com/package/react-router-dom`

## Using multiple endpoints / Dependencies

**frontend/src/main.tsx**

```
import "./index.css";
import { createRoot } from "react-dom/client";
import { StrictMode } from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Navigation from "./components/Navigation.tsx";
import App from "./App.tsx";
import CardHandler from "./CardHandler.tsx";

createRoot(document.getElementById("root")!).render(
<StrictMode>
    <BrowserRouter>
    <Navigation />
    <Routes>
        <Route path="/" element={<App />} />
        <Route path="/cards" element={<CardHandler />} />
    </Routes>
    </BrowserRouter>
</StrictMode>
);
```

## Using multiple endpoints / Dependencies

We'll use the same pp.tsx and **CardHandler.tsx** components and we'll focus on routing them from a **Navigation** component.

**frontend/src/components/Navigation.tsx**

```
import { Link } from "react-router-dom";
import "./Navigation.css";
function Navigation() {
return (
    <nav className="navigation">
    <Link to="/" className="nav-link">
        Home
    </Link>
    <Link to="/cards" className="nav-link">
        Cards
    </Link>
    </nav>
);}
export default Navigation;
```

## Glossary / Terms I

| | |
|---|---|
| transpilation | The process of converting code from one programming language to another, usually with a similar abstraction level. It differs from compiled, which converts code into machine instructions. A transpiler (short for “source-to-source compiler”) preserves high-level structure—functions, variables, and logic and adapts syntax or semantics. Examples: TypeScript to JavaScript, JSX to JavaScript, Scala to Java. 18 |

## Glossary / Abbreviations I

| | |
|---|---|
| DOM | Document Object Model 3 |
| HMR | Hot Module Replacement. A feature that allows for the replacement of modules in a running application, without the need to reload the whole page. 24 |
| HTML | Hypertext Markup Language 3, 6 |
| JSX | JavaScript XML 3 |