Class Activity 11: Fetching resources from backend

(SUMBIT ALL YOUR CODE ON MOODLE)

In this activity, we will learn to connect the frontend to the backend building a full ReactJS App.

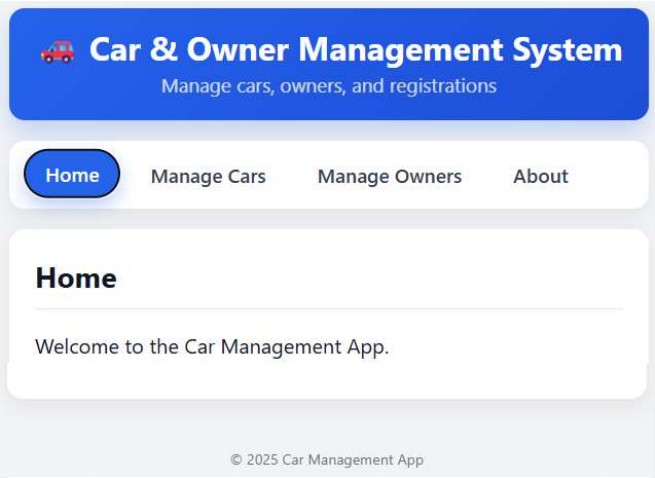
The api.tsx – functions to fetch data from the backend

First, let us implement the fetch methods using axios in a separate file called api.tsx under services folder. These functions do not change much from project to project. In fact you can copy the same file to another project and use it as is, you just need to change the end points. Also, you need to add other functions for other endpoints that you want to implement.

```
import axios from "axios";

const BASE = "http://localhost:8080";

export async function apiGet(path: string) {
  try {
    const response = await axios.get(`${BASE}${path}`);
    return response.data; // JavaScript object from API
  } catch (error) {
    console.error("GET request failed:", error);
    throw error; // re-throw so React can handle it
  }
}

export async function apiPost(path: string, data?: any) {
  try {
    const response = await axios.post(`${BASE}${path}`, data);
    return response.data;
  } catch (error) {
    console.error("POST request failed:", error);
    throw error;
  }
}

export async function apiPut(path: string, data?: any) {
  try {
    const response = await axios.put(`${BASE}${path}`, data);
    return response.data;
  } catch (error) {
    console.error("PUT request failed:", error);
    throw error;
  }
}

export async function apiDelete(path: string) {
  try {
    const response = await axios.delete(`${BASE}${path}`);
    return response.data;
  } catch (error) {
    console.error("DELETE request failed:", error);
    throw error;
  }
}
```

Part 1: design or the frontend website

The plan is to build a simple frontend website using REACT-typescript like this:



These are the components we will have:

- App
- Header
- Menu
- MainBody:
    - HomePage
    - ManageCarsPage
    - ManageOwnersPage
    - AboutPage
- Footer

Component Tree Structure

The rendered tree when the app is running (conceptually) should be as follows:

```
<App>
  <div className="container">
    <Header />
    <Menu currentPage={currentPage} onChangePage={setCurrentPage} />
    <MainBody currentPage={currentPage}>
       ├─ if currentPage === "home":
       │     <HomePage />
       │
       ├─ if currentPage === "cars":
       │     <ManageCarsPage />
       │
       ├─ if currentPage === "owners":
       │     <ManageOwnersPage />
       │
       └─ if currentPage === "about":
             <AboutPage />
    </MainBody>
    <Footer />
  </div>
</App>
```

Conceptually, this is what should be happening:

- App always renders: Header, Menu, MainBody, Footer
- MainBody only swaps what's inside <main> based on currentPage state variables. The state variable will be set in the menu component.
- ManageCarsPage is just content inside MainBody

Implementing the general structure

The App should always render Header/Menu/Footer

```
import { useState } from 'react'
import Header from './components/Header'
import Menu from './components/Menu'
import MainBody from './components/MainBody'
import Footer from './components/Footer'
import { Page } from './services/types'
import './App.css'
export function App() {
  const [currentPage, setCurrentPage] = useState<Page>("home");

  return (
    <div className="container">
      <Header />
      <Menu currentPage={currentPage} onChangePage={setCurrentPage} />
      <MainBody currentPage={currentPage} />
      <Footer />
    </div>
  );
}
```

The MainBody.tsx – selects which component to show

MainBody should not replace the entire layout; it should only change its inner content. Ensure MainBody keeps its wrapper <main className="main-body">

```
import '../App.css'
import type { Page } from "../services/types";

interface MainBodyProps {
  currentPage: Page;
}

export function MainBody({ currentPage }: MainBodyProps) {
  return (
    <main className="main-body">
      {currentPage === "home" && <HomePage />}
```

```
      {currentPage === "cars" && <ManageCarsPage />}
      {currentPage === "owners" && <ManageOwnersPage />}
      {currentPage === "about" && <AboutPage />}
    </main>
  );
}

function HomePage() {
  return (
    <section>
      <h2>Home</h2>
      <p>Welcome to the Car Management App.</p>
    </section>
  );
}

function ManageCarsPage() {
  return (
    <section>
      <h2>Manage Cars</h2>
      <p>Here you will be able to view, add, edit, and delete cars.</p>
      {/* Replace with your forms/tables later */}
    </section>
  );
}

function ManageOwnersPage() {
  return (
    <section>
      <h2>Manage Owners</h2>
      <p>Here you will be able to manage car owners.</p>
      {/* Replace with your forms/tables later */}
    </section>
  );
}

function AboutPage() {
  return (
    <section>
      <h2>About</h2>
      <p>This app is used to manage cars and their owners.</p>
    </section>
  );
}
```

The various pages must only render page content only and they should return a section component.

The Menu.tsx – sets the currentPage state variable

The menu is implemented as buttons. When you click on a button action is taken to change the style and the currentPage value as follows:

```
import type { Page } from "../services/types";
import '../App.css'
interface MenuProps {
  currentPage: Page;
  onChangePage: (page: Page) => void;
}

export function Menu({ currentPage, onChangePage }: MenuProps) {
  return (
    <nav className="menu">
      <ul>
        <li>
          <button
            className={currentPage === "home" ? "active" : ""}
            onClick={() => onChangePage("home")}
          >
            Home
          </button>
        </li>
        <li>
          <button
            className={currentPage === "cars" ? "active" : ""}
            onClick={() => onChangePage("cars")}
          >
            Manage Cars
          </button>
        </li>
        <li>
          <button
            className={currentPage === "owners" ? "active" : ""}
            onClick={() => onChangePage("owners")}
          >
            Manage Owners
          </button>
        </li>
        <li>
          <button
            className={currentPage === "about" ? "active" : ""}
            onClick={() => onChangePage("about")}
          >
            About
          </button>
        </li>
      </ul>
    </nav>
  );
}
```

The type Page is defined in a separate file under services, called types.tsx:

```
export type Page = "home" | "cars" | "owners" | "about";
```

Test the app

You can test the app now and check that the menu works. The style is bad but it works. Write a style sheet to make it look good and correct. In particular the menu should become horizontal.

Part 2: Implementing ManageCarsPage.tsx page

ManageCarsPage must only render page content. It should be like:

```
export function ManageCarsPage() {
  // state, hooks, handlers...
  ……
  ……..
  return (
    <section>
      <h2>Manage Cars</h2>
      {/* table + form */}
    </section>
  );
}
```

It should **not**:

- Manage currentPage
- Render <Header/>, <Menu/>, or <Footer/>
- Call ReactDOM.createRoot(...)
- Return <html>, <body>, or ...

Make sure ManageCarsPage is not using <html>, <body>, or . If you currently navigate to ManageCarsPage by rendering it directly (e.g. in main.tsx you changed <App /> to <ManageCarsPage />), then the header/menu/footer will disappear.

Here is an incomplete code that defines the hooks and helper functions:

```
import { useEffect, useState } from "react";
import { apiGet, apiPost, apiPut, apiDelete } from "../services/api";
import type { CarRequestModel, CarResponseModel, OwnerResponseModel } from "../models/Models";


type Mode = "create" | "edit";

const emptyCar: CarRequestModel = {
  make: "",
  model: "",
  color: "",
  year: new Date().getFullYear(),
  vin: "",
  registrationNumber: "",
  price: 0,
  ownerId: 0,
};

export function ManageCarsPage() {
  const [cars, setCars] = useState<CarResponseModel[]>([]);
  const [owners, setOwners] = useState<OwnerResponseModel[]>([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const [formMode, setFormMode] = useState<Mode>("create");
  const [editingId, setEditingId] = useState<number | null>(null);
  const [formData, setFormData] = useState<CarRequestModel>(emptyCar);
```

```
useEffect(() => {
  loadCars();
  loadOwners();
}, []);

async function loadCars() {
  try {
    setLoading(true);
    setError(null);
    const data = await apiGet("/cars");
    setCars(data as CarResponseModel[]);
  } catch (err) {
    console.error(err);
    setError("Failed to load cars.");
  } finally {
    setLoading(false);
  }
}

async function loadOwners() {
  try {
    const data = await apiGet("/owners");
    setOwners(data as OwnerResponseModel[]);
  } catch (err) {
    console.error(err);
    setError("Failed to load owners.");
  }
}

// Type as 'any' to avoid JSX runtime conflicts
function handleChange(e: any) {
  const { name, value } = e.target;

  setFormData((prev) => ({
    ...prev,
    [name]:
      name === "year" || name === "ownerId"
        ? Number(value)
        : name === "price"
        ? Number(value)
        : value,
  }));
}

function handleSubmit(e: any) {
  e.preventDefault();
  saveCar().catch((err) => {
    console.error(err);
    setError("Failed to save car.");
  });
}

async function saveCar() {
  setError(null);

  if (formMode === "create") {
    const created = await apiPost("/cars", formData);
    setCars((prev) => [...prev, created as CarResponseModel]);
  } else if (formMode === "edit" && editingId != null) {
    const updated = await apiPut(`/cars/${editingId}`, formData);
    setCars((prev) =>
      prev.map((c) => (c.id === editingId ? (updated as CarResponseModel) : c))
    );
  }

  resetForm();
}

function resetForm() {
```

```
    setFormData(emptyCar);
    setFormMode("create");
    setEditingId(null);
  }

  function handleEdit(car: CarResponseModel) {
    setFormMode("edit");
    setEditingId(car.id);

    setFormData({
      make: car.brand,
      model: car.model,
      color: car.color,
      year: car.modelYear,
      vin: car.vin,
      registrationNumber: car.registrationNumber,
      price: car.price,
      ownerId: car.owner?.id ?? 0,
    });
  }

  async function handleDelete(id: number) {
    if (!window.confirm("Are you sure you want to delete this car?")) return;

    try {
      setError(null);
      await apiDelete(`/cars/${id}`);
      setCars((prev) => prev.filter((c) => c.id !== id));
    } catch (err) {
      console.error(err);
      setError("Failed to delete car.");
    }
  }

  return (
    <section>
      <h2>Manage Cars</h2>

      {error && <p style={{ color: "red" }}>{error}</p>}
      {loading && <p>Loading cars...</p>}

      {/* Cars list */}
      <table className="cars-table">
       ..............
              ..............
      </table>

      {/* Form */}
      <h3>{formMode === "create" ? "Add New Car" : "Edit Car"}</h3>

      <form onSubmit={handleSubmit} className="car-form">
        ..............
              ..............
      </form>
    </section>
  );
}
```

Notice how we use the async keyword for all the functions that need to call the fetch api, because they are all asynchronous calls. The `CarRequestModel`, `CarResponseModel`, `CarResponseModel`, `OwnerResponseModel` are defined in the models.tsx file to mirror the models in the backend. For example, here is one of them:

```
export interface CarRequestModel {
  make: string;
  model: string;
  color: string;
  year: number;
  vin: string;
  registrationNumber: string;
  price: number;
  ownerId: number;
}
```

Implementing ManageOwnersPage.tsx page

Finish implementing **ManageOwnersPage.tsc** by following what you have learned above.

Here is the final project structure:

src/
├── components/
│ ├── App.tsx
│ ├── Header.tsx
│ ├── Menu.tsx
│ ├── MainBody.tsx
│ ├── Footer.tsx
├── pages/
│ ├── ~~HomePage.tsx~~ (right now it is defined inline)
│ ├── ManageCarsPage.tsx
│ ├── ManageOwnersPage.tsx
│ ├── ~~AboutPage tsx~~ (right now it is defined inline)
├── services/
│ ├── api.ts
│ ├── types.ts
├── models/
│ ├── models.ts

Testing the Frontend with Spring Boot Backend

Overview

When you run your React frontend and Spring Boot backend on your local machine, they run on **different ports**:

- **React Frontend**: typically http://localhost:3000 or http://localhost:5173 (Vite)
- **Spring Boot Backend**: typically http://localhost:8080

When your frontend tries to make HTTP requests to the backend (e.g., fetching cars or owners), the browser blocks these requests by default due to **CORS (Cross-Origin Resource Sharing)** security policy.

What is CORS?

CORS is a security feature implemented by web browsers to prevent malicious websites from making unauthorized requests to your backend.

The Problem:

- Your React app runs on http://localhost:5173
- Your Spring Boot API runs on http://localhost:8080
- These are considered **different origins** (different ports = different origins)
- Browser blocks the request and shows an error like:
- Access to fetch at 'http://localhost:8080/api/cars' from origin
- 'http://localhost:5173' has been blocked by CORS policy

The Solution:

Configure your Spring Boot backend to **allow** requests from your React frontend.

Step-by-Step: Configuring CORS in Spring Boot

1. Create a Configuration Class

In your Spring Boot project, create a new Java class called *WebConfig.java* in your configuration package in the same folder as the main app:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig {
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
```

```
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost:5173", "http://localhost:3000")
// your React dev URL(s)
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
                        .allowedHeaders("*")
                        .allowCredentials(true);
            }
        };
    }
}
```

2. Understanding the Configuration

Let's break down what each part does:

Configuration	Explanation
@Configuration	Tells Spring this is a configuration class
@Bean	Creates a Spring-managed bean
.addMapping("/**")	Applies CORS settings to **all endpoints** in your API
.allowedOrigins(...)	Specifies which frontend URLs are allowed to make requests
.allowedMethods(...)	Specifies which HTTP methods are allowed (GET, POST, PUT, DELETE, OPTIONS)
.allowedHeaders("*")	Allows all headers in the request
.allowCredentials(true)	Allows cookies and authentication headers to be sent

3. Adjust the Allowed Origins

Make sure the allowedOrigins matches your React development server:

- **Create React App** uses http://localhost:3000
- **Vite** uses http://localhost:5173
- You can include both to be safe!

Testing Steps

1. Start Your Spring Boot Backend

In your Spring Boot project directory

./mvnw spring-boot:run

Or if using Gradle

./gradlew bootRun

Verify it's running by visiting: http://localhost:8080

2. Start Your React Frontend

In your React project directory

npm run dev

Or

npm start

Your React app should open at http://localhost:3000 or http://localhost:5173

3. Test the Connection

Open your browser's **Developer Tools** (F12) and check the **Console** and **Network** tabs:

Success:

- No CORS errors in the console
- Network tab shows successful API calls (status 200)
- Data loads in your frontend

Failure:

- CORS error in console
- Network tab shows failed requests
- **Solution**: Double-check your WebConfig.java and restart the backend

---

Common Issues and Solutions

Issue 1: CORS Error Still Appears

Solution:

- Make sure you **restart** your Spring Boot application after adding the CORS configuration
- Verify the port numbers match in allowedOrigins

Issue 2: 404 Not Found

Solution:

- Check your API endpoint URLs in the React code

- Verify your Spring Boot controllers have the correct @RequestMapping paths

Issue 3: Connection Refused

Solution:

- Make sure your Spring Boot backend is actually running
- Check if it's running on port 8080 (or adjust your fetch URLs accordingly)