Fullstack JavaScript with Spring Boot

3rd semester @ Erhvervsakademi København

GitHub

Outline

- Cross-Origin Resource Sharing (CORS)
- JavaScript Array methods for manipulating data
- Sorting in a HTML table

Cross-Origin Resource Sharing (CORS)

> Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers that restricts web pages from making requests to a different domain than the one that served the web page.

CORS is a mechanism that allows web servers to specify who can access their resources and how they can be accessed.

What would happen if CORS was not implemented?

> If CORS was not implemented, any web page could make requests to any server, which could lead to security vulnerabilities such as Cross-Site Request Forgery (CSRF) and data theft.

Example:

Suppose a user is logged into their bank account on `bank.com` and visits a malicious website `malicious.com`. If CORS was not implemented, the malicious website could make requests to `bank.com` on behalf of the user, potentially stealing sensitive information or performing unauthorized actions.

CORS

The *Scheme*, *Host*, and *Port* of the URL must be the same as the one that served the web page for the request to be allowed.

Example:

- `http://www.example.com` can make requests to `http://www.example.com/api`
  - **Scheme**: `http`
  - **Host:** `www.example.com`
  - **Port:** `80` (default for http)

Our problem

> When we try to make a request from our frontend (running on `http://localhost:5500`) to our backend (running on `http://localhost:8080`), the browser will block the request because they are on different origins.

Solution

We can solve this problem in multiple ways:

1. Configure CORS in our Spring Boot backend to allow requests from our frontend.
2. Put the frontend inside the Spring Boot application (`src/main/resources/static`) and serve it from the same origin.

Configuring CORS in Spring Boot

There are several ways to configure CORS in Spring Boot:

1. Using `@CrossOrigin` annotation on controller classes or on specific handler methods
2. Using `WebMvcConfigurer` to configure CORS globally

Example: CORS configuration on `@CrossOrigin` annotation

```
@RestController
@CrossOrigin(origins = "http://localhost:5500")
public class MyController {
// ...
}
```

> This will allow requests from `http://localhost:5500` to access the resources of this controller.

Example: CORS configuration using `WebMvcConfigurer`

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:5500")
                .allowedMethods("GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS")
                .allowedHeaders("*");
    }
}
```

> This will allow requests from `http://localhost:5500` to access all resources of the application.

CORS

> **When a web page makes a request to a different domain, the browser sends an HTTP request with an Origin header that indicates the domain of the web page.**

Example:

```
GET /api/data HTTP/1.1
Host: api.example.com
Origin: http://www.example.com
```

> The `Origin` header is used by the server to determine whether to allow the request or not. The browser will add this automatically when making cross-origin requests.

CORS Preflight Request

> For certain types of requests (e.g., `PUT`, `DELETE`, `PATCH`), the browser sends a preflight request to the server to check if the actual request is allowed.

```
OPTIONS /api/data HTTP/1.1
Host: api.example.com
Origin: http://www.example.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Content-Type
```

> The server must respond to the preflight request with appropriate CORS headers to allow the actual request to proceed, like this:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, PATCH, OPTIONS
Access-Control-Allow-Headers: Content-Type
```

JavaScript Array methods for manipulating data

> When we receive data from the backend, we often need to manipulate it before displaying it on the frontend. JavaScript provides several array methods that can help us with this.

- `map()` : Creates a new array by applying a function to each element of the original array.
- `filter()` : Creates a new array with all elements that pass a test implemented by a function.
- `reduce()` : Executes a reducer function on each element of the array, resulting in a single output value.
- `sort()` : Sorts the elements of an array and mutates the original array.
- `forEach()` : Executes a provided function once for each array element.

Example: Using `map()` to transform data

Using an anonymous arrow function:

```
const number = [1, 2, 3, 4, 5];
const squared = number.map(x => x * x); // [1, 4, 9, 16, 25]
```

Alternatively:

```
const number = [1, 2, 3, 4, 5];
function square(x) {
  return x * x;
}
const squared = number.map(square); // [1, 4, 9, 16, 25]
```

Example: Using `map()` to transform list of objects

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];

const userNames = users.map(user => user.name); // ['Alice', 'Bob', 'Charlie']
```

> Note that `map()` does not modify the original array, it returns a new array with the transformed values.

Example: Using `filter()` to filter data

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(x => x % 2 === 0); // [2, 4]
```

> The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Example: Using `filter()` to filter list of objects

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];

const usersAbove30 = users.filter(user => user.age > 30);
// [{ name: 'Charlie', age: 35 }]
```

> The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Example: Using `reduce()` to aggregate data

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0); // 15
```

> The `reduce()` method executes a reducer function on each element of the array, resulting in a single output value. The second argument (0 in this case) is the initial value of the accumulator.

Example: Using `reduce()` to aggregate list of objects

```
const order = {
  items: [
    { name: 'Item 1', price: 10 },
    { name: 'Item 2', price: 20 },
    { name: 'Item 3', price: 30 }
  ]
};
const totalPrice = order.items.reduce((total, item) => total + item.price, 0); // 60
```

> The `reduce()` method executes a reducer function on each element of the array, resulting in a single output value. The second argument (0 in this case) is the initial value of the total price.

Example: Using `sort()` to sort data

> The `sort()` method sorts the elements of an array in place and returns the sorted array. By default, it sorts elements as strings in ascending order.

Sorting numbers in ascending order:

```
const numbers = [5, 2, 9, 1, 5, 6];
numbers.sort((a, b) => a - b); // [1, 2, 5, 5, 6, 9]
```

Sorting numbers in descending order:

```
const numbers = [5, 2, 9, 1, 5, 6];
numbers.sort((a, b) => b - a); // [9, 6, 5, 5, 2, 1]
```

Example: Using `sort()` to sort list of objects

Sorting by name in ascending order:

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];
users.sort((a, b) => a.name.localeCompare(b.name));
```

Sorting by name in descending order:

```
users.sort((a, b) => b.name.localeCompare(a.name));
```

> The `localeCompare()` method compares two strings in the current locale and returns a number indicating whether the reference string comes before, after, or is the same as the given string.

Example: Using `forEach()` to iterate over data

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(x => console.log(x));
```

> The `forEach()` method executes a provided function once for each array element. It does not return a new array, it simply executes the function for each element.

Sorting in a HTML table

> We can use the `sort()` method to sort data in a HTML table when the user clicks on a column header.

For this we will use a general function that can sort by any property of the objects in the array:

```
function sortBy(property, isAscending = true) {
    return (a,b) => {
        const valueA = a[property];
        const valueB = b[property];
        if (typeof valueA === 'string' && typeof valueB === 'string') {
            return isAscending ? valueA.localeCompare(valueB) : valueB.localeCompare(valueA);
        }
        if (typeof valueA === 'number' && typeof valueB === 'number') {
            return isAscending ? valueA - valueB : valueB - valueA;
        }
        return 0;
    };
}
```

Using the `sortBy` function to sort by name in ascending order:

Note that `sortBy(property, isAscending)` returns a **comparator function** that can be used in the `sort()` method.

```
users.sort(sortBy('name', true)); // Sort by name in ascending order
```

Given that `users` is an array of objects with a `name` property, this will sort the users by their names in ascending order.

Modularizing with ES Modules

> ES Modules allow us to split our code into separate files and import/export functionality as needed.

When using modules, we need to use the `type="module"` attribute in our script tag in `index.html`:

```
<script type="module" src="app.js"></script>
```

> This tells the browser that `app.js` is an ES module, allowing us to use `import` and `export` statements in our JavaScript files.

Example: Exporting and importing functions in ES Modules

```
export async function fetchData() {
    // Fetch data from API
}
```

```
import { fetchData } from './api.js';

async function initApp() {
    const data = await fetchData();
    // Initialize app with data
}
```

Two ways to export functionality in ES Modules:

Using named exports:

```
export function fetchData() {
    // Fetch data from API
}
```

Using default export:

```
export default function fetchData() {
    // Fetch data from API
}
```

When using default export, you can import it with any name you want (no curly braces needed):

```
import fetchData from './api.js';
```