04

React & SpringBoot 연동

01. CROS(Cross-Origin Resource Sharing)

- 스크립트가 현재 로드된 출처와 다른 출처(origin)로 리소스를 요청할 때 웹 브라우저가 발생시키는 보안 메커니즘
- 동일 출처 정책(Same-Origin Policy)
  - 현재 로드된 페이지와 같은 출처에만 요청을 보낼 수 있는 정책
  - 출처 : (도메인, 프로토콜, 포트)

- 서버와 클라이언트 간의 요청을 제어하고, 다른 출처에서의 리소스 접근을 허용하거나 차단
- CORS를 통해 서버가 허용된 출처에 한해 데이터를 접근할 수 있도록 설정

01. CROS(Cross-Origin Resource Sharing)

CROS 절차

1. 스크립트가 url을 통해 요청전달을 시도

   : 같은 출처인지 확인

2. 다른 출처로 보낸다면 요청 전달을 일시 중지
3. 요청을 전달하려는 출처로 프리플라이트 요청(Preflight Request)을 전달

   : 다른 출처에 대한 요청을 서버가 허용하는지 미리 확인

4. 서버가 요청을 허용하면 중지되었던 요청을 전달. 허용하지 않으면 브라우저는 CORS 오류를 발생시키고 실제 요청을 차단

※ 프리플라이트 요청(Preflight Request)

: 브라우저가 서버로 OPTIONS 메서드를 사용하여 보내는 요청

01. CROS(Cross-Origin Resource Sharing)

■ 전역단위로 설정

```
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 implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**") // 모든 경로에 대해 CORS 허용
                .allowedOrigins("http://localhost:3000") // 허용할 출처
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                                    // 허용할 HTTP 메서드
                .allowedHeaders("*") // 모든 헤더 허용
                .allowCredentials(true); // 쿠키 및 인증 정보 허용(필요 시)
    }
}
```

02. AJAX(Asynchronous JavaScript and XML)

- JavaScript를 사용하여 웹 페이지가 새로 고침 없이 서버와 데이터를 비동기적으로 주고받을 수 있게 해주는 기술
  - 비동기 동신(fetch())
  - JSON(JavaScript Object Notation) 데이터 사용

- XMLHttpRequest()를 통해 비동기 통신을 사용했으나 최근에는 Fetch API가 더 많이 사용됨

- Fetch()
  - Promise를 기반으로 하여 비동기 처리

02. AJAX(Asynchronous JavaScript and XML)

Promise

- ECMAScript6에서 추가된 기능
- JavaScript의 비동기 작업을 처리하기 위해 사용되는 객체
- 비동기 작업이 완료될 때까지 대기하거나 작업 결과에 따라 다음 처리를 지정
- Promise의 4가지 상태
  - Pending (대기)
    - 비동기 작업이 아직 완료되지 않은 상태로 약속(Promise)이 이루어지거나 거부되기 전의 초기 상태
  - Fulfilled (이행)
    - 비동기 작업이 성공적으로 완료된 상태로 then 메서드를 통해 성공 결과를 처리
  - Rejected (거부)
    - 비동기 작업이 실패한 상태로 catch 메서드를 통해 에러를 처리
  - Settled (완료)
    - Promiss의 작업의 성공/실패가 모두 완료된 상태

02. AJAX(Asynchronous JavaScript and XML)

Promise

- Promise 객체는 Promise 생성자 함수로 생성
- resolve와 reject 두 가지 콜백 함수를 인수로 받음
- resolve : then의 함수에 매개변수에게 내용 전달
- reject : catch의 함수에 매개변수에게 내용 전달

```
const promise = new Promise((resolve, reject) => {
    const success = true;

    if (success) {
        resolve("작업이  성공했습니다!"); // 성공  시  호출
        } else {
        reject("작업이  실패했습니다."); // 실패  시  호출
        }
});
```

02. AJAX(Asynchronous JavaScript and XML)

Promise

```
const asyncTask = new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000); // 1초 후에 1을 반환
});

asyncTask
    .then(result => {
        console.log(result); // 1
        return result + 1;
    })
    .then(result => {
        console.log(result); // 2
        return result + 1;
    })
    .then(result => {
        console.log(result); // 3
    })
    .catch(error => {
        console.error(error); // 에러 발생 시 출력
        });
```

02. AJAX(Asynchronous JavaScript and XML)

Promise.all()

- 여러 개의 Promise가 모두 이행될 때까지 기다렸다가, 모든 결과를 배열로 반환
- 하나라도 실패하면 catch로 이동

```
Promise.all([promise1, promise2, promise3])
    .then(results => {
        console.log(results); // 모든 promise 결과가 배열로 반환됨
        })
    .catch(error => {
        console.error(error); // 하나라도 실패하면 에러 출력
        });
```

02. AJAX(Asynchronous JavaScript and XML)

Promise.race()

- 여러 개의 Promise 중 가장 먼저 완료된 Promise의 결과를 반환
  하나라도 실패하면 catch로 이동
- 이행되거나 거부된 첫 번째 Promise의 결과가 전달

```
Promise.race([promise1, promise2, promise3])
    .then(result => {
        console.log(result); // 가장 빨리 완료된 promise의 결과
        })
    .catch(error => {
        console.error(error); // 가장 빨리 실패한 경우 에러 출력
        });
```

02. AJAX(Asynchronous JavaScript and XML)

Promise.allSettled()

- 모든 Promise의 성공 여부에 관계없이 모든 Promise가 처리될 때까지 대기
- 각각의 Promise가 성공/실패여부와 그 결과를 배열로 반환.

```
Promise.allSettled([promise1, promise2, promise3])
    .then(results => {
        console.log(results); // 각 Promise의 상태와 결과가 객체 배열로 반환됨
        });
```

Promise.any()

- 여러 Promise 중 하나라도 이행되면 그 결과를 반환.
- 모든 Promise가 실패할 경우에만 에러가 발생.

```
Promise.any([promise1, promise2, promise3])
    .then(result => {
        console.log(result); // 가장 먼저 이행된 결과
        })
    .catch(error => {
        console.error(error); // 모든 promise가 실패한 경우
        });
```

02. AJAX(Asynchronous JavaScript and XML)

fetch()

- fetch()는 Promise를 반환하여 요청이 성공했는지 실패했는지에 따라 .then()과 .catch()로 처리

```
fetch(url, options)
    .then(response => {
        if (!response.ok) {
            throw new Error("네트워크  응답이  성공하지  않았습니다.");
        }
        return response.json();
// response.body에 있는 ReadableStream 데이터를 JSON 형태로  변환
        })
    .then(data => {
        console.log(data); // 서버에서  받은  데이터
        })
    .catch(error => {
        console.error("Fetch 에러:", error);
    });
```

02. AJAX(Asynchronous JavaScript and XML)

fetch()

GET 요청

```
fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("에러  발생:", error));
```

POST 요청

```
fetch("https://api.example.com/data", {
    method: "POST",
    headers: {
        "Content-Type": "application/json", // “text/plain”
    },
    body: JSON.stringify({ name: "John", age: 30 }), // 보낼  데이터
})
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("에러  발생:", error));
```

02. AJAX(Asynchronous JavaScript and XML)

fetch()

PUT 요청

```
fetch("https://api.example.com/data/1", {
    method: "PUT",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({ name: "Jane", age: 25 }), // 업데이트할 데이터
})
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("에러 발생:", error));
```

02. AJAX(Asynchronous JavaScript and XML)

fetch()

DELETE 요청

```
fetch("https://api.example.com/data/1", {
    method: "DELETE",
})
    .then(response => {
        if (response.ok) {
            console.log("데이터  삭제  성공");
        } else {
            throw new Error("삭제  실패");
        }
    })
    .catch(error => console.error("에러  발생:", error));
```

02. AJAX(Asynchronous JavaScript and XML)

await 사용

- 비동기 작업이 완료될 때까지 함수를 일시 중지하는 역할
- 비동기 함수를 동기함수처럼 동작하게 함
- 비동기 함수 내부에서만 사용가능
- 비동기 함수를 순차적으로 처리할 수 있음

```
try {
  const response = await fetch("http://localhost:8080/product-list");
  if (!response.ok) {
    throw new Error("네트워크  응답이  실패했습니다.");
  }
  const result = await response.json();
  console.log(result);
} catch (error) {
  console.log(error.message);
}
```

02. AJAX(Asynchronous JavaScript and XML)

```
useEffect(() => {
    const fetchData = async () => {
        try {
          const response = await fetch("http://localhost:8080/product-list");
          if (!response.ok) {
            throw new Error("네트워크  응답이  실패했습니다.");
          }
          const result = await response.json();
          result.map(t=>dispatch(productAdd(t)));
          } catch (error) {
          console.log(error.message);
        }
      };
      fetchData(); // 비동기  함수  호출
    },
[]);
```

- Async를 함수 앞에 붙이면 그 함수는 비동기 함수가 되고 promiss를 리턴
- useEffect에게 전달되는 콜백함수는 비동기함수이면 안됨
  - 비동기 함수를 정의하고 호출하여 사용