07. 비동기 통신

1. AJAX (Asynchronous JavaScript and XML)

AJAX는 JavaScript를 사용하여 서버로부터 비동기적으로 데이터를 가져오는 기술입니다.

XML뿐만 아니라 JSON, 텍스트 등 다양한 데이터를 가져올 수 있습니다.

AJAX 예제 - `XMLHttpRequest` 사용하기

```
const xhr = new XMLHttpRequest(); // XMLHttpRequest 객체 생성
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts', true); // GET 요청 설정
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) { // 요청이 완료되었을 때
    if (xhr.status === 200) { // 응답이 성공적일 때
      console.log(JSON.parse(xhr.responseText)); // JSON 데이터를 파싱해서 출력
    } else {
      console.error('Request failed');
    }
  }
};
xhr.send(); // 요청 전송
```

장점

- 오래된 브라우저에서도 동작 (호환성 높음)

단점

- 코드가 길고 가독성이 떨어짐
- 비동기 처리 코드가 복잡해질 수 있음

---

2. Fetch API

`fetch()`는 현대적인 자바스크립트에서 사용되는 API로, `Promise` 기반으로 데이터를 가져옵니다.

Fetch 사용법

```
fetch('https://jsonplaceholder.typicode.com/posts') // GET 요청
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not ok'); // 오류 처리
    }
    return response.json(); // JSON 형식으로 변환
  })
  .then((data) => console.log(data)) // 성공적으로 데이터를 가져왔을 때
  .catch((error) => console.error('Fetch error:', error)); // 오류가 발생했을 때
```

POST 요청 예제

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json', // JSON 형식임을 명시
  },
  body: JSON.stringify({ title: 'foo', body: 'bar', userId: 1 }),
})
  .then((response) => response.json())
  .then((data) => console.log('POST Response:', data))
  .catch((error) => console.error('Error:', error));
```

Fetch 장점과 단점

장점

- 코드가 깔끔하고 이해하기 쉽다.
- `Promise`를 기반으로 하므로 비동기 처리를 효과적으로 관리 가능하다.

단점

- `fetch`는 네트워크 오류 외의 HTTP 오류(404, 500 등)를 자동으로 처리하지 않는다.

---

3. Promise

`Promise`는 비동기 작업을 처리하기 위한 객체입니다.

Promise 기본 사용법

```
const getData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true; // 성공 여부를 가정
    if (success) {
      resolve('Data loaded successfully');
    } else {
      reject('Error loading data');
    }
  }, 2000);
});

getData
  .then((message) => console.log(message)) // 성공 시 실행
  .catch((error) => console.error(error)); // 실패 시 실행
```

---

4. Axios

Axios는 `Promise` 기반의 HTTP 클라이언트로, Fetch보다 더 편리한 기능을 제공합니다.

Axios 설치

Node.js 환경:

```
npm install axios
```

브라우저에서 사용할 경우 CDN 링크 추가:

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Axios 사용 예제 - GET 요청

```
axios
  .get('https://jsonplaceholder.typicode.com/posts')
  .then((response) => {
    console.log(response.data); // 응답 데이터
  })
  .catch((error) => {
    console.error('Error fetching data:', error);
  });
```

Axios 사용 예제 - POST 요청

```
axios
  .post('https://jsonplaceholder.typicode.com/posts', {
    title: 'foo',
    body: 'bar',
    userId: 1,
  })
  .then((response) => {
    console.log('POST Response:', response.data);
  })
  .catch((error) => {
    console.error('Error:', error);
  });
```

5. 병렬처리

`Promise.all`을 사용하면 여러 개의 Promise를 병렬로 실행할 수 있습니다.

```
// 병렬로 API 요청하기
const fetchUser = fetch('https://jsonplaceholder.typicode.com/users/1');
const fetchPosts = fetch('https://jsonplaceholder.typicode.com/posts');

Promise.all([fetchUser, fetchPosts])
  .then((responses) => Promise.all(responses.map((res) => res.json()))) // 응답 JSON 변환
  .then(([user, posts]) => {
    console.log('User:', user);
    console.log('Posts:', posts);
  })
  .catch((error) => console.error('Error:', error));
```

6. `async/await` 란?

`async/await`는 자바스크립트에서 **비동기 작업**을 처리하는 최신 문법입니다.

- `async`: 함수를 비동기로 선언합니다.
- `await`: Promise의 이행(성공) 또는 거절(실패)을 기다립니다.

`async/await`는 **Promise** 기반으로 동작하며, 콜백이나 `then` 체인 대신, 동기 코드처럼 작성할 수 있도록 도와줍니다.

1) `async` 함수

`async` 키워드로 선언된 함수는 항상 **Promise**를 반환합니다.

- 반환 값이 Promise가 아니라면, 반환 값을 자동으로 Promise로 감쌉니다.

```
sync function example() {
  return 'Hello, async!'; // Promise.resolve('Hello, async!')
```

```
}

example().then((result) => console.log(result)); // 출력: He
llo, async!
```

2) `await` 키워드

`await`는 **Promise가 해결될 때까지** 기다립니다.

- `await`은 반드시 `async` 함수 내부에서만 사용할 수 있습니다.

```
async function example() {
  const result = await new Promise((resolve) => setTimeout
(() => resolve('Done!'), 1000));
  console.log(result); // 1초 후 출력: Done!
}

example();
```

---

7. `async/await`와 기존 비동기 처리 방식 비교

1) 콜백 방식

```
setTimeout(() => {
  console.log('콜백 실행');
  setTimeout(() => {
    console.log('중첩된 콜백 실행');
  }, 1000);
}, 1000);
```

문제점: 콜백이 중첩되면서 가독성이 떨어지는 **"콜백 지옥"** 문제가 발생.

---

2) Promise 방식

```
new Promise((resolve) => {
  setTimeout(() => resolve('첫 번째 Promise 완료'), 1000);
})
  .then((result) => {
    console.log(result);
    return new Promise((resolve) => setTimeout(() => resolv
e('두 번째 Promise 완료'), 1000));
  })
  .then((result) => console.log(result));
```

장점: `then` 체인을 사용해 콜백 지옥을 완화.

문제점: 여전히 체인이 길어지면 복잡해질 수 있음.

---

3) `async/await` 방식

```
async function process() {
  const step1 = await new Promise((resolve) =>
    setTimeout(() => resolve('첫 번째 단계 완료'), 1000)
  );
  console.log(step1);

  const step2 = await new Promise((resolve) =>
    setTimeout(() => resolve('두 번째 단계 완료'), 1000)
  );
  console.log(step2);
}

process();
```

장점: 코드가 동기식처럼 읽혀 가독성이 좋아짐.