웹 시스템과 웹 개발의 이해

웹 시스템의 개요

웹 시스템의 정의

웹 시스템이란 인터넷을 통해 정보와 서비스를 제공하는 시스템으로, 클라이언트와 서버가 상호작용하여 작동합니다. 웹 애플리케이션은 웹 시스템의 한 형태로, 웹 브라우저를 통해 접근할 수 있는 응용 소프트웨어입니다.

예로는 다음과 같습니다:

- **온라인 쇼핑몰**: Amazon, eBay
- **블로그**: WordPress, Blogger
- **포럼**: Reddit, Stack Overflow
- **웹 애플리케이션**: Gmail, Facebook, Trello

웹 시스템의 구성요소

웹 시스템은 크게 클라이언트, 서버, 데이터베이스 세 부분으로 나뉩니다.

- **클라이언트**: 사용자가 웹 브라우저를 통해 접속하는 부분입니다. 클라이언트는 웹 페이지를 요청하고 서버로부터 받은 데이터를 표시합니다.
- **서버**: 클라이언트의 요청을 처리하고 필요한 데이터를 제공하는 역할을 합니다. 서버는 정적 파일을 제공하는 웹 서버와 동적인 비즈니스 로직을 처리하는 애플리케이션 서버로 나뉩니다.
- **데이터베이스**: 데이터를 저장하고 관리합니다. 서버는 데이터베이스에 저장된 데이터를 클라이언트의 요청에 따라 조회하거나 갱신합니다.

웹 시스템의 작동 원리

웹 시스템은 HTTP/HTTPS 프로토콜을 사용하여 클라이언트와 서버가 통신합니다.

- **HTTP**: Hypertext Transfer Protocol의 약자로, 웹에서 데이터를 주고받기 위한 프로토콜입니다. HTTP는 보안이 없으며, 데이터를 암호화하지 않고 전송합니다.
- **HTTPS**: HTTP Secure의 약자로, HTTP에 SSL/TLS를 추가하여 데이터를 암호화하여 전송합니다.

요청과 응답의 흐름:

1. 클라이언트가 웹 브라우저를 통해 특정 URL에 접근합니다.
2. 브라우저는 해당 URL의 서버에 HTTP 요청을 보냅니다.
3. 서버는 요청을 처리하여 필요한 데이터를 데이터베이스에서 조회합니다.

4. 서버는 조회한 데이터를 바탕으로 HTTP 응답을 생성합니다.
5. 클라이언트는 서버로부터 받은 응답 데이터를 웹 페이지로 렌더링합니다.

클라이언트 측 기술

웹 브라우저의 역할

웹 브라우저는 클라이언트 측에서 웹 페이지를 요청하고, 서버로부터 받은 HTML, CSS, JavaScript 파일을 해석하여 사용자가 볼 수 있는 웹 페이지로 렌더링하는 역할을 합니다. 주요 웹 브라우저로는 Chrome, Firefox, Safari, Edge가 있습니다.

HTML, CSS, JavaScript 개요

- **HTML (HyperText Markup Language)**: 웹 페이지의 구조를 정의합니다. HTML은 문서의 제목, 본문, 이미지, 링크 등을 정의하는 데 사용됩니다.
- **CSS (Cascading Style Sheets)**: 웹 페이지의 스타일을 정의합니다. CSS는 글꼴, 색상, 레이아웃 등을 설정하여 웹 페이지의 시각적 디자인을 담당합니다.
- **JavaScript**: 웹 페이지의 동작과 인터랙션을 정의합니다. JavaScript는 사용자 입력 처리, 데이터 검증, 동적 콘텐츠 업데이트 등을 수행합니다.

예제:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Basic Web Page</title>
    <style>
        body { font-family: Arial, sans-serif; }
        h1 { color: blue; }
    </style>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is a basic web page.</p>
    <script>
        document.querySelector('h1').onclick = () => alert('Hello,
World!');
    </script>
</body>
</html>
```

서버 측 기술

서버의 역할

서버는 웹 서버와 애플리케이션 서버로 나뉩니다.

- **웹 서버**: 클라이언트의 요청을 처리하고 정적 파일(HTML, CSS, JavaScript)을 제공합니다.
- **애플리케이션 서버**: 동적인 비즈니스 로직을 처리하고, 데이터베이스와 상호작용하여 데이터를 조회하거나 갱신합니다.

서버 언어 개요

서버 측에서는 다양한 프로그래밍 언어가 사용됩니다. 주요 언어로는 PHP, Node.js, Python이 있습니다.

PHP:

```php
<?php
echo "Hello, World!";
?>
```

Node.js:

```javascript
const http = require('http');
http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello, World!\n');
}).listen(3000);
```

Python (Flask):

```python
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run()
```

데이터베이스 개요

데이터베이스는 데이터를 저장하고 관리합니다. 주요 데이터베이스 시스템으로는 관계형 데이터베이스와 NoSQL 데이터베이스가 있습니다.

- **관계형 데이터베이스**: MySQL, PostgreSQL. 테이블 간의 관계를 정의하여 데이터를 구조화합니다.
- **NoSQL 데이터베이스**: MongoDB, Cassandra. 유연한 데이터 모델을 제공하여 비정형 데이터를 처리합니다.

클라이언트와 서버의 통신

HTTP(HyperText Transfer Protocol)는 웹에서 클라이언트와 서버 간의 데이터를 주고받기 위해 사용되는 프로토콜입니다. 주로 HTML 문서를 주고받는 데 사용되지만, 이미지, 비디오, JSON 데이터 등 다양한 형태의 데이터를 전송할 수 있습니다. HTTP는 요청-응답(request-response) 모델을 기반으로 작동합니다.

HTTP의 기본 작동 방식

1. **클라이언트 요청**:
   - **웹 브라우저**: 사용자가 브라우저에 URL을 입력하거나 링크를 클릭하면, 브라우저는 HTTP 요청을 생성하여 서버에 보냅니다.
   - **요청 구성 요소**:
     - **요청 라인**: 메서드 (GET, POST 등), 요청 URL, HTTP 버전으로 구성됩니다.
     - **헤더**: 요청에 대한 추가 정보를 포함합니다. 예: `User-Agent`, `Accept`, `Authorization`.
     - **바디**: (선택 사항) POST와 같은 요청에서 서버로 전송할 데이터를 포함합니다.

   예시:

   ```
   GET /index.html HTTP/1.1
   Host: www.example.com
   User-Agent: Mozilla/5.0
   ```

2. **서버 응답**:
   - 서버는 요청을 처리하고, 클라이언트에 HTTP 응답을 보냅니다.
   - **응답 구성 요소**:
     - **상태 라인**: HTTP 버전, 상태 코드(예: 200 OK, 404 Not Found), 상태 메시지로 구성됩니다.
     - **헤더**: 응답에 대한 추가 정보를 포함합니다. 예: `Content-Type`, `Content-Length`.
     - **바디**: 요청한 리소스(예: HTML 문서)의 실제 데이터를 포함합니다.

   예시:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 138

<html>
<head>
<title>Example</title>
</head>
<body>
<h1>Hello, world!</h1>
</body>
</html>
```

HTTP의 특징

1. **비연결성 (Stateless)**:
   - HTTP는 기본적으로 비연결성 프로토콜입니다. 각 요청-응답 쌍은 독립적이며, 서버는 이전 요청에 대한 정보를 유지하지 않습니다.
   - 이를 보완하기 위해 쿠키, 세션, 토큰 등을 사용하여 상태를 유지합니다.
2. **무상태성 (Connectionless)**:
   - 클라이언트와 서버는 요청과 응답이 완료되면 연결을 끊습니다.
   - HTTP/1.1부터는 `Keep-Alive` 헤더를 통해 연결을 유지할 수 있습니다.
3. **HTTP 메서드**:
   - **GET**: 데이터를 요청합니다. 서버에서 자원을 가져옵니다.
   - **POST**: 데이터를 서버로 보냅니다. 자원을 생성하거나 데이터 처리를 요청합니다.
   - **PUT**: 자원을 업데이트합니다.
   - **DELETE**: 자원을 삭제합니다.
   - **HEAD**: GET과 유사하지만 응답 바디를 포함하지 않습니다.
   - **OPTIONS**: 서버에서 지원하는 메서드를 반환합니다.
4. **HTTPS**:
   - HTTP에 보안 계층(SSL/TLS)을 추가하여 데이터를 암호화합니다.
   - 이를 통해 데이터의 기밀성, 무결성, 인증을 보장합니다.

HTTP의 작동 예시

1. 사용자가 브라우저에 `http://www.example.com/index.html`을 입력합니다.
2. 브라우저는 서버 `www.example.com`에 `GET /index.html HTTP/1.1` 요청을 보냅니다.
3. 서버는 요청을 처리하고, `200 OK` 상태 코드와 함께 HTML 문서를 응답합니다.
4. 브라우저는 받은 HTML 문서를 렌더링하여 사용자에게 표시합니다.

프로그래밍 세계관

프로그래밍 언어들

C 언어

C 언어는 프로그래밍 언어의 조상 중 하나로, 시스템 프로그래밍과 저수준 메모리 조작에 강력합니다.

- **흥미로운 사실**: C 언어는 유닉스 운영체제를 개발하기 위해 만들어졌습니다.
- **간단한 예제**:

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    printf("Sum: %d\n", add(2, 3));
    return 0;
}
```

C++

C++는 C 언어의 확장으로, 객체지향 프로그래밍(OOP)을 지원합니다.

- **흥미로운 사실**: C++는 게임 개발에서 많이 사용됩니다.
- **간단한 예제**:

```cpp
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::cout << "Sum: " << add(2, 3) << std::endl;
    return 0;
}
```

C#

C#은 마이크로소프트가 개발한 언어로, .NET 프레임워크에서 주로 사용됩니다.

- **흥미로운 사실**: C#은 Unity 게임 엔진의 주요 스크립팅 언어입니다.
- **간단한 예제**:

```
using System;

class Program {
    static int Add(int a, int b) {
        return a + b;
    }

    static void Main() {
        Console.WriteLine("Sum: " + Add(2, 3));
    }
}
```

Javascript

JavaScript는 웹 개발의 필수 언어로, 브라우저에서 동작하는 인터랙티브 웹 페이지를 만들 수 있습니다.

- **흥미로운 사실**: JavaScript는 10일 만에 만들어졌습니다.
- **간단한 예제**:

```
function add(a, b) {
    return a + b;
}

console.log("Sum:", add(2, 3));
```

Python

Python은 읽기 쉽고 쓰기 쉬운 문법으로 인해 매우 인기 있는 언어입니다.

- **흥미로운 사실**: Python의 이름은 코미디 그룹 몬티 파이튼(Monty Python)에서 유래되었습니다.
- **간단한 예제**:

```
def add(a, b):
    return a + b

print("Sum:", add(2, 3))
```

Java

Java는 플랫폼 독립적인 언어로, "Write Once, Run Anywhere"의 철학을 가지고 있습니다.

- **흥미로운 사실**: 전 세계 ATM 기기의 약 3/4가 Java로 작성되었습니다.
- **간단한 예제**:

```java
public class HelloWorld {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println("Sum: " + add(2, 3));
    }
}
```

Kotlin

Kotlin은 Java의 대안으로, 간결하고 안전한 문법을 제공합니다.

- **흥미로운 사실**: Kotlin은 구글의 공식 Android 개발 언어입니다.
- **간단한 예제**:

```kotlin
fun add(a: Int, b: Int): Int {
    return a + b
}

fun main() {
    println("Sum: ${add(2, 3)}")
}
```

PHP

PHP는 서버 측 스크립팅 언어로, 동적인 웹 페이지를 생성하는 데 사용됩니다.

- **흥미로운 사실**: 전 세계 웹사이트의 약 79%가 PHP를 사용합니다.
- **간단한 예제**:

```php
<?php
function add($a, $b) {
    return $a + $b;
}

echo "Sum: " . add(2, 3);
?>
```

Ruby

Ruby는 단순성과 생산성을 강조한 언어로, Ruby on Rails 프레임워크로 유명합니다.

- **흥미로운 사실**: Twitter는 처음에 Ruby on Rails로 작성되었습니다.
- **간단한 예제**:

```ruby
def add(a, b)
    a + b
end

puts "Sum: #{add(2, 3)}"
```

Swift

Swift는 애플이 개발한 언어로, iOS와 macOS 애플리케이션 개발에 사용됩니다.

- **흥미로운 사실**: Swift는 빠르고 안전한 언어로 설계되었습니다.
- **간단한 예제**:

```swift
func add(_ a: Int, _ b: Int) -> Int {
    return a + b
}

print("Sum: \(add(2, 3))")
```

Go(GoLang)

Go는 구글이 개발한 언어로, 간결하고 효율적인 시스템 프로그래밍을 목표로 합니다.

- **흥미로운 사실**: Go는 병렬 처리를 쉽게 하기 위해 설계되었습니다.
- **간단한 예제**:

```go
package main

import "fmt"

func add(a int, b int) int {
    return a + b
}

func main() {
```

```
    fmt.Println("Sum:", add(2, 3))
}
```

프로그래밍 별 학습곡선

https://www.clien.net/service/board/park/14302921

프로그래밍 패러다임

절차적 프로그래밍(Procedural Programming)

절차적 프로그래밍은 순차적으로 명령을 실행하는 방식입니다.

- **간단한 예제**:

```
def add(a, b):
    return a + b

print("Sum:", add(2, 3))
```

객체지향 프로그래밍(Object-Oriented Programming)

객체지향 프로그래밍은 객체와 클래스를 사용하여 데이터를 구조화합니다.

- **간단한 예제 (Python)**:

```
class Calculator:
    def add(self, a, b):
        return a + b

calc = Calculator()
print("Sum:", calc.add(2, 3))
```

함수형 프로그래밍(Functional Programming)

함수형 프로그래밍은 함수의 조합으로 프로그램을 구성하는 방식입니다.

- **간단한 예제 (JavaScript)**:

```
const add = (a, b) => a + b;
console.log("Sum:", add(2, 3));
```

선언형 프로그래밍(Declarative Programming)

선언형 프로그래밍은 원하는 결과를 무엇인지 설명하는 방식입니다.

- **간단한 예제 (SQL)**:

```sql
SELECT name FROM users WHERE age > 30;
```

이벤트 기반 프로그래밍(Event-Driven Programming)

이벤트 기반 프로그래밍은 이벤트의 발생과 처리를 중심으로 프로그램을 구성합니다.

- **간단한 예제 (JavaScript)**:

```javascript
document.getElementById("myButton").addEventListener("click", () => {
    console.log("Button clicked!");
});
```

반응형 프로그래밍(Reactive Programming)

반응형 프로그래밍은 데이터 흐름과 변화에 반응하여 동작하는 방식입니다.

- **간단한 예제 (RxJS)**:

```javascript
const { fromEvent } = rxjs;
const { map } = rxjs.operators;

fromEvent(document, 'click')
  .pipe(map(() => 'Button clicked!'))
  .subscribe(console.log);
```

비동기 프로그래밍(Asynchronous Programming)

비동기 프로그래밍은 작업이 완료될 때까지 기다리지 않고 다음 작업을 실행하는 방식입니다.

- **간단한 예제 (JavaScript)**:

```javascript
function addAsync(a, b, callback) {
    setTimeout(() => {
        callback(a + b);
    }, 1000);
}
```

```
addAsync(2, 3, (result) => {
    console.log("Sum:", result);
});
```

프로그래밍 언어와 개발 플랫폼 선택

프로그래밍 언어와 개발 플랫폼을 선택하는 것은 마치 우리가 일상에서 사용하는 도구나 즐기는 취미를 선택하는 것과 비슷합니다. 중요한 것은 각 도구가 어떤 용도에 적합한지 이해하고, 자신의 필요와 취향에 맞게 선택하는 것입니다. 큰 차이는 없으니, 자신에게 맞는 것을 사용하면 됩니다. == 큰 차이는 없으니

예시 1: 요리 도구 선택

프로그래밍 언어를 선택하는 것은 요리할 때 어떤 도구를 사용할지 선택하는 것과 같습니다.

- **Python**: 다목적으로 사용할 수 있는 주방 도구 세트. 초보자도 쉽게 배울 수 있고, 다양한 요리를 할 수 있습니다.
- **Java**: 튼튼한 주방 도구 세트. 복잡한 요리도 할 수 있지만, 조금 더 숙련된 사용자가 필요할 수 있습니다.
- **JavaScript**: 빠르고 쉽게 요리를 할 수 있는 도구. 웹 요리(웹 개발)에 특히 강합니다.
- **C/C++**: 전통적인 요리 도구 세트. 낮은 수준의 세밀한 요리가 가능하지만, 배우는 데 시간이 좀 더 걸립니다.

이처럼, 어떤 언어를 사용하든 궁극적으로는 요리를 완성할 수 있습니다. 중요한 것은 자신이 편하게 사용할 수 있는 도구를 선택하는 것입니다.

예시 2: 운동 선택

개발 플랫폼을 선택하는 것은 운동을 선택하는 것과 비슷합니다.

- **Node.js**: 역동적이고 빠른 운동. 웹 애플리케이션 개발에 적합하며, 빠르게 시작할 수 있습니다.
- **Spring**: 체계적이고 구조화된 운동. 대규모 엔터프라이즈 애플리케이션 개발에 적합합니다.
- **Django**: 균형 잡힌 운동. Python을 사용하며, 빠른 개발이 가능하지만 강력한 기능도 제공합니다.
- **Rails**: 즐거운 운동. Ruby를 사용하며, 생산성을 높여주는 다양한 도구가 포함되어 있습니다.

운동을 선택할 때 가장 중요한 것은 자신이 즐기고 지속할 수 있는 것을 찾는 것입니다. 어떤 운동이든 건강에 좋고, 꾸준히 하면 큰 효과를 볼 수 있습니다.

예시 3: 자동차 선택

개발 도구나 프레임워크를 선택하는 것은 자동차를 선택하는 것과 비슷합니다.

- **VS Code**: 다재다능한 세단. 다양한 플러그인을 통해 기능을 확장할 수 있습니다.

- **IntelliJ IDEA**: 고급형 세단. 강력한 기능과 편의성을 제공하지만, 약간의 학습 곡선이 있습니다.
- **Eclipse**: 견고한 SUV. 전통적이지만 강력한 기능을 제공합니다.
- **Sublime Text**: 경쾌한 스포츠카. 빠르고 가볍지만, 플러그인 설치가 필요할 수 있습니다.

자동차를 선택할 때는 자신의 운전 스타일과 용도에 맞는 것을 선택하면 됩니다. 모든 자동차가 목적지에 도달할 수 있지만, 편의성과 스타일에 따라 선택이 달라질 수 있습니다.

Git과 GitHub의 역할 요약

Git

Git은 분산 버전 관리 시스템으로, 소스 코드의 변경 이력을 추적하고 협업을 효율적으로 관리하는 도구입니다. Git의 주요 역할은 다음과 같습니다:

1. **버전 관리**:
   - 코드 변경 이력을 저장하여 이전 버전으로의 복구가 가능.
   - 각 변경 사항을 커밋(commit)으로 기록하여 언제든지 변경 내역을 추적할 수 있음.
2. **분산 저장소**:
   - 중앙 서버 없이 로컬에서 작업 가능.
   - 여러 개발자가 동시에 작업할 수 있도록 분산된 형태로 저장소 관리.
3. **브랜치와 병합**:
   - 새로운 기능이나 버그 수정을 독립된 브랜치(branch)에서 작업 가능.
   - 작업 완료 후 병합(merge)을 통해 메인 코드베이스에 통합 가능.
4. **협업 도구**:
   - 여러 개발자가 동일한 프로젝트에서 작업하면서 충돌을 최소화하고, 효과적으로 협업할 수 있도록 지원.

GitHub

GitHub는 Git 저장소를 호스팅하는 웹 기반 서비스로, 프로젝트 관리와 협업을 위한 다양한 기능을 제공합니다. GitHub의 주요 역할은 다음과 같습니다:

1. **원격 저장소 호스팅**:
   - Git 저장소를 클라우드에서 호스팅하여 어디서든 접근 가능.
   - 원격 저장소를 통해 여러 개발자가 공동으로 작업할 수 있음.
2. **협업 도구**:
   - 풀 리퀘스트(Pull Request): 코드 변경 사항을 검토하고 병합 요청을 관리하는 기능.
   - 이슈 트래킹(Issue Tracking): 버그, 기능 요청 등을 추적하고 관리하는 시스템.
3. **프로젝트 관리**:
   - 프로젝트 보드, 마일스톤, 태스크 관리 등 다양한 프로젝트 관리 도구 제공.
   - 팀 멤버 간의 역할 분담과 진행 상황을 체계적으로 관리.

4. **커뮤니티와 네트워킹**:
    - 오픈 소스 프로젝트를 쉽게 공유하고 협업할 수 있는 플랫폼.
    - 다른 개발자들과의 네트워킹 및 지식 공유 가능.
5. **CI/CD 통합**:
    - GitHub Actions와 같은 기능을 통해 지속적 통합(Continuous Integration)과 지속적 배포(Continuous Deployment) 파이프라인 설정 가능.

Git과 GitHub는 함께 사용되어 소프트웨어 개발 프로세스를 효율적으로 관리하고, 팀원 간의 협업을 원활하게 합니다. Git은 로컬과 분산 환경에서 버전 관리를 담당하고, GitHub는 이를 클라우드 기반으로 호스팅하여 협업과 프로젝트 관리를 용이하게 합니다.

1. Git 설치

먼저 Git이 설치되어 있는지 확인합니다. 만약 설치되어 있지 않다면 Git 공식 웹사이트에서 다운로드 후 설치합니다.

```
git --version
```

2. 프로젝트 디렉토리로 이동 및 Git 초기화

1. **프로젝트 디렉토리 생성 및 이동**:

    ```
    mkdir my-awesome-project
    cd my-awesome-project
    ```

2. **Git 저장소 초기화**:

    ```
    git init
    ```

    이제 `my-awesome-project` 디렉토리는 Git 저장소로 초기화되었습니다.

3. Git 기본 명령어

1. **새 파일 추가 및 첫 커밋**:

    ```
    echo "# My Awesome Project" > README.md
    git add README.md
    git commit -m "Initial commit with README"
    ```

    여기서 `README.md` 파일을 생성하고, Git 스테이징 영역에 추가한 후 첫 커밋을 생성했습니다.

2. **상태 확인**:

   ```
   git status
   ```

   현재 Git 저장소의 상태를 확인할 수 있습니다.
3. **로그 확인**:

   ```
   git log
   ```

   커밋 로그를 확인할 수 있습니다.

4. GitHub 기본 사용법

1. **GitHub 계정 생성 및 새 저장소 생성**:
   - GitHub에 접속하여 계정을 생성합니다.
   - 새로운 저장소를 생성합니다. 예를 들어, 저장소 이름을 `my-awesome-project`로 합니다.
2. **GitHub 원격 저장소 추가**:

   ```
   git remote add origin https://github.com/<your-username>/my-awesome-project.git
   ```

   여기서 `<your-username>`을 자신의 GitHub 사용자 이름으로 바꿉니다.
3. **로컬 커밋을 GitHub 원격 저장소로 푸시**:

   ```
   git push -u origin master
   ```

   이제 로컬 저장소의 내용을 GitHub 원격 저장소에 푸시했습니다.

5. 프로젝트 클론

다른 사람이 GitHub에서 해당 프로젝트를 클론(clone)하여 자신의 로컬 머신에 복사할 수 있습니다.

```
git clone https://github.com/<your-username>/my-awesome-project.git
```

6. 프로젝트 포크

다른 사용자가 자신의 GitHub 계정으로 해당 프로젝트를 포크(fork)하여 독립된 복사본을 생성할 수 있습니다. 포크한 프로젝트는 독립적으로 변경할 수 있으며, 변경 사항을 원래 저장소로 풀 리퀘스트(pull request)할 수 있습니다.

1. **GitHub에서 포크 버튼 클릭**:
   - 원래 프로젝트 페이지에서 `Fork` 버튼을 클릭하여 자신의 GitHub 계정으로 복사합니다.
2. **포크된 프로젝트 클론**:

```
git clone https://github.com/<your-username>/my-awesome-project.git
```

7. 브랜치 및 협업

1. **새 브랜치 생성 및 체크아웃**:

```
git checkout -b feature-branch
```

2. **변경 사항 커밋**:

```
echo "New feature" > feature.txt
git add feature.txt
git commit -m "Add new feature"
```

3. **브랜치 푸시**:

```
git push origin feature-branch
```

4. **풀 리퀘스트 생성**:
   - GitHub 웹사이트에서 해당 브랜치에 대한 풀 리퀘스트(pull request)를 생성합니다.

웹 보안의 기초

웹 보안의 중요성

웹 애플리케이션 보안은 개인 정보 보호와 시스템 무결성 유지를 위해 필수적입니다.

주요 보안 위협

- **SQL 인젝션**: 악의적인 SQL 코드를 주입하여 데이터베이스를 공격.
- **XSS (Cross-Site Scripting)**: 악성 스크립트를 주입하여 사용자 정보를 탈취.
- **CSRF (Cross-Site Request Forgery)**: 사용자가 의도하지 않은 요청을 실행하여 공격.

기본 보안 대책

- **입력 검증**: 사용자 입력 검증, 필터링.
- **인증 및 권한 관리**: 사용자 인증, 권한 부여.
- **데이터 암호화**: 데이터 전송, 저장 시 암호화.

최신 웹 개발 트렌드

프론트엔드 프레임워크

- **React**: 컴포넌트 기반 라이브러리, Facebook 개발. 상태 관리, 가상 DOM 사용.
- **Vue.js**: 경량화, 반응형 UI 개발. 템플릿 기반, 단일 파일 컴포넌트.
- **Angular**: 완전한 프레임워크, Google 개발. 양방향 데이터 바인딩, 모듈 시스템.

백엔드 프레임워크

- **Express**: Node.js용 경량 웹 프레임워크. 미들웨어, 라우팅 기능 제공.
- **Django**: Python용 고수준 웹 프레임워크. MTV(Model-Template-View) 아키텍처, ORM(Object-Relational Mapping) 제공.
- **Flask**: Python용 마이크로 프레임워크. 유연한 라우팅, 확장성 높은 구조.

DevOps

- **CI/CD**: 지속적 통합, 지속적 배포. 개발, 테스트, 배포 자동화.
- **Jenkins**: 오픈 소스 자동화 서버. 빌드, 테스트, 배포 파이프라인 구성.
- **Docker**: 컨테이너 기반 가상화 플랫폼. 애플리케이션의 일관된 실행 환경 제공.

자바스크립트 기본

변수와 데이터 타입

변수 선언과 초기화

var

`var`는 ES6 이전에 변수를 선언할 때 사용하던 방법입니다. 함수 스코프를 가지며, 같은 이름으로 여러 번 선언할 수 있습니다.

```
var x = 5;
console.log(x); // 5

var x = 10;
console.log(x); // 10
```

let

`let`은 ES6 이후에 도입된 키워드로, 블록 스코프를 가집니다. 같은 이름으로 재선언할 수 없지만, 재할당은 가능합니다.

```
let y = 5;
console.log(y); // 5

// let y = 10; // 오류: 재선언 불가
y = 10; // 재할당은 가능
console.log(y); // 10
```

const

`const`는 상수를 선언할 때 사용하며, 블록 스코프를 가집니다. 선언과 동시에 초기화해야 하며, 재할당이 불가합니다.

```
const z = 5;
console.log(z); // 5

// z = 10; // 오류: 재할당 불가
```

기본 데이터 타입

문자열 (String)

문자열은 텍스트 데이터를 나타내며, 작은따옴표('') 또는 큰따옴표("")로 감싸서 표현합니다.

```
let greeting = "Hello, world!";
let name = 'Alice';

console.log(greeting); // "Hello, world!"
console.log(name); // "Alice"
```

숫자 (Number)

숫자는 정수와 부동 소수점 숫자를 포함합니다.

```
let age = 30;
let pi = 3.14;
```

```
console.log(age); // 30
console.log(pi); // 3.14
```

불리언 (Boolean)

불리언은 논리값을 나타내며, `true` 또는 `false` 값을 가집니다.

```
let isStudent = true;
let hasGraduated = false;

console.log(isStudent); // true
console.log(hasGraduated); // false
```

Undefined

`undefined`는 변수가 선언되었지만 값이 할당되지 않은 상태를 나타냅니다.

```
let address;
console.log(address); // undefined
```

Null

`null`은 의도적으로 값이 없음을 나타내기 위해 사용됩니다.

```
let phoneNumber = null;
console.log(phoneNumber); // null
```

변수와 데이터 타입 예제

다양한 예제를 통해 변수와 데이터 타입을 이해해 봅시다.

예제 1: 변수 선언과 데이터 타입

```
var username = "JohnDoe"; // 문자열
let score = 95; // 숫자
const isValid = true; // 불리언

console.log(username); // "JohnDoe"
console.log(score); // 95
console.log(isValid); // true
```

예제 2: 재할당과 스코프

```
let message = "Hello";
console.log(message); // "Hello"

message = "Welcome";
console.log(message); // "Welcome"

// 블록 스코프 예제
if (true) {
    let localMessage = "This is local";
    console.log(localMessage); // "This is local"
}
// console.log(localMessage); // 오류: 블록 스코프 내에서만 접근 가능
```

예제 3: 상수와 데이터 타입

```
const pi = 3.14159;
console.log(pi); // 3.14159

// pi = 3.14; // 오류: 상수는 재할당 불가

const person = {
    name: "Alice",
    age: 25
};
console.log(person);

// 객체의 프로퍼티는 변경 가능
person.age = 26;
console.log(person); // { name: "Alice", age: 26 }
```

자바스크립트 데이터 타입의 동적 특성

자바스크립트는 동적 타입 언어로, 변수의 데이터 타입을 변경할 수 있습니다.

예제 4: 데이터 타입 변경

```
let dynamicVar = "Hello";
console.log(typeof dynamicVar); // "string"

dynamicVar = 42;
console.log(typeof dynamicVar); // "number"
```

```
dynamicVar = true;
console.log(typeof dynamicVar); // "boolean"
```

typeof 연산자

`typeof` 연산자는 변수의 데이터 타입을 확인할 때 사용됩니다.

```
console.log(typeof "Hello"); // "string"
console.log(typeof 123); // "number"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (자바스크립트의 버그로 인해 null의 타입이 object로 표시됨)
```

조건문과 반복문

조건문

if, else if, else

조건문은 특정 조건에 따라 코드의 실행 흐름을 제어하는 데 사용됩니다. `if` 문은 조건이 참일 때 실행할 코드를 정의하며, `else if` 와 `else` 는 각각 추가적인 조건과 모든 조건이 거짓일 때 실행할 코드를 정의합니다.

```
let age = 20;

if (age < 18) {
    console.log("미성년자입니다.");
} else if (age >= 18 && age < 65) {
    console.log("성인입니다.");
} else {
    console.log("노인입니다.");
}
```

switch

`switch` 문은 하나 이상의 경우(case)에 대해 실행할 코드를 선택합니다. 각 경우는 `case` 키워드와 함께 정의되며, 일치하는 경우가 없을 때 실행할 코드는 `default` 키워드와 함께 정의됩니다.

```
let color = "red";

switch (color) {
    case "red":
```

```
        console.log("빨간색입니다.");
        break;
    case "blue":
        console.log("파란색입니다.");
        break;
    default:
        console.log("알 수 없는 색상입니다.");
}
```

반복문

for

`for` 문은 특정 횟수만큼 코드를 반복 실행합니다. 초기식, 조건식, 증감식으로 구성됩니다.

```
for (let i = 0; i < 5; i++) {
    console.log(`i: ${i}`);
}
```

while

`while` 문은 조건식이 참인 동안 코드를 반복 실행합니다. 조건식이 거짓이 되면 반복이 종료됩니다.

```
let j = 0;
while (j < 5) {
    console.log(`j: ${j}`);
    j++;
}
```

do-while

`do-while` 문은 코드를 최소 한 번 실행하고, 조건식이 참인 동안 반복 실행합니다.

```
let k = 0;
do {
    console.log(`k: ${k}`);
    k++;
} while (k < 5);
```

조건문과 반복문 예제

예제 1: if-else 조건문

```
let temperature = 25;

if (temperature > 30) {
    console.log("날씨가 덥습니다.");
} else if (temperature < 15) {
    console.log("날씨가 춥습니다.");
} else {
    console.log("날씨가 적당합니다.");
}
```

예제 2: switch 조건문

```
let day = 3;

switch (day) {
    case 1:
        console.log("월요일");
        break;
    case 2:
        console.log("화요일");
        break;
    case 3:
        console.log("수요일");
        break;
    case 4:
        console.log("목요일");
        break;
    case 5:
        console.log("금요일");
        break;
    case 6:
        console.log("토요일");
        break;
    case 7:
        console.log("일요일");
        break;
    default:
        console.log("잘못된 요일입니다.");
}
```

예제 3: for 반복문

```
for (let i = 1; i <= 10; i++) {
    console.log(`${i} * ${i} = ${i * i}`);
```

```
}
```

예제 4: while 반복문

```
let n = 1;
let factorial = 1;

while (n <= 5) {
    factorial *= n;
    n++;
}

console.log(`5! = ${factorial}`); // 5! = 120
```

예제 5: do-while 반복문

```
let m = 0;
do {
    console.log(`m: ${m}`);
    m++;
} while (m < 3);
```

함수 정의와 호출

함수 선언

함수 선언은 `function` 키워드를 사용하여 함수를 정의합니다. 함수 이름과 매개변수를 지정하고, 함수 본문에 실행할 코드를 작성합니다.

```
function greet(name) {
    return `Hello, ${name}!`;
}

console.log(greet("Alice")); // "Hello, Alice!"
```

함수 표현식

함수 표현식은 함수를 변수에 할당하는 방식입니다. 익명 함수 또는 기명 함수를 사용할 수 있습니다.

익명 함수:

```
const square = function(number) {
    return number * number;
};

console.log(square(5)); // 25
```

기명 함수:

```
const factorial = function fact(n) {
    if (n <= 1) return 1;
    return n * fact(n - 1);
};

console.log(factorial(5)); // 120
```

화살표 함수

화살표 함수는 간결한 문법으로 함수를 정의하는 방식입니다. `function` 키워드 대신 `=>`를 사용합니다. 화살표 함수는 `this` 바인딩이 다르다는 특징이 있습니다.

```
const multiply = (a, b) => a * b;

console.log(multiply(2, 3)); // 6
```

매개변수가 하나일 때는 괄호를 생략할 수 있습니다.

```
const greet = name => `Hello, ${name}!`;

console.log(greet("Bob")); // "Hello, Bob!"
```

본문이 한 줄일 때는 중괄호와 `return` 키워드를 생략할 수 있습니다.

```
const add = (a, b) => a + b;

console.log(add(2, 3)); // 5
```

스코프와 클로저

스코프

전역 스코프

전역 스코프는 코드 전체에서 접근할 수 있는 범위를 의미합니다. 전역 변수는 함수 밖에서 선언된 변수입니다.

```
let globalVar = "I am a global variable";

function displayGlobalVar() {
    console.log(globalVar); // "I am a global variable"
}

displayGlobalVar();
console.log(globalVar); // "I am a global variable"
```

지역 스코프

지역 스코프는 함수나 블록 내부에서만 접근할 수 있는 범위를 의미합니다. 지역 변수는 함수나 블록 내부에서 선언된 변수입니다.

```
function localScopeExample() {
    let localVar = "I am a local variable";
    console.log(localVar); // "I am a local variable"
}

localScopeExample();
// console.log(localVar); // 오류: localVar는 지역 변수이므로 함수 밖에서 접근 불가
```

블록 스코프는 `let`과 `const` 키워드를 사용하여 선언된 변수에 적용됩니다.

```
if (true) {
    let blockScopedVar = "I am block scoped";
    console.log(blockScopedVar); // "I am block scoped"
}
// console.log(blockScopedVar); // 오류: blockScopedVar는 블록 스코프이므로 블록 밖에서 접근 불가
```

클로저

클로저는 함수와 함수가 선언된 어휘적 환경의 조합을 의미합니다. 클로저를 사용하면 함수가 생성될 때의 환경을 기억하여 나중에 참조할 수 있습니다.

클로저의 개념

```
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
```

```
        console.log(`Outer Variable: ${outerVariable}`);
        console.log(`Inner Variable: ${innerVariable}`);
    };
}

const newFunction = outerFunction("outside");
newFunction("inside");
// Outer Variable: outside
// Inner Variable: inside
```

위 예제에서 `innerFunction`은 `outerFunction`이 호출될 때 생성된 환경(스코프)을 기억합니다. 따라서 `innerFunction`이 실행될 때 `outerVariable`에 접근할 수 있습니다.

클로저의 활용

1. **상태 유지**: 클로저는 함수 호출 간에 상태를 유지할 수 있습니다.

```
function counter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}

const increment = counter();
console.log(increment()); // 1
console.log(increment()); // 2
console.log(increment()); // 3
```

2. **데이터 캡슐화**: 클로저를 사용하여 데이터를 캡슐화하고 외부에서 직접 접근하지 못하도록 할 수 있습니다.

```
function createPerson(name) {
    return {
        getName: function() {
            return name;
        },
        setName: function(newName) {
            name = newName;
        }
    };
}

const person = createPerson("Alice");
console.log(person.getName()); // "Alice"
```

```
person.setName("Bob");
console.log(person.getName()); // "Bob"
```

3. **콜백 함수**: 클로저는 콜백 함수 내에서 변수의 상태를 유지할 수 있습니다.

```
function setup() {
    let count = 0;
    document.getElementById("myButton").addEventListener("click",
function() {
        count++;
        console.log(`Button clicked ${count} times`);
    });
}

setup();
// 버튼을 클릭할 때마다 count가 증가합니다.
```

ES6 문법과 함수 예제

기본 매개변수 (Default Parameters)

ES6에서 함수 매개변수에 기본값을 설정할 수 있습니다.

```
function greet(name = "stranger") {
    return `Hello, ${name}!`;
}

console.log(greet()); // "Hello, stranger!"
console.log(greet("Alice")); // "Hello, Alice!"
```

템플릿 리터럴 (Template Literals)

템플릿 리터럴을 사용하여 문자열을 쉽게 연결할 수 있습니다.

```
function greet(name) {
    return `Hello, ${name}!`;
}

console.log(greet("Alice")); // "Hello, Alice!"
```

나머지 매개변수 (Rest Parameters)

나머지 매개변수 문법을 사용하여 함수가 가변 인수를 받을 수 있습니다.

```
function sum( ...numbers) {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
```

스프레드 연산자 (Spread Operator)

스프레드 연산자를 사용하여 배열이나 객체를 펼칠 수 있습니다.

```
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4, 5];

console.log(newNumbers); // [1, 2, 3, 4, 5]
```

객체 리터럴 (Object Literals)

ES6에서는 객체 리터럴을 사용하여 더 간결하게 객체를 정의할 수 있습니다.

```
const name = "Alice";
const age = 25;

const person = {
    name,
    age,
    greet() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age}
years old.`);
    }
};

person.greet(); // "Hello, my name is Alice and I am 25 years old."
```

객체

객체 리터럴

객체는 자바스크립트에서 키-값 쌍으로 데이터를 저장할 수 있는 중요한 데이터 구조입니다. 객체 리터럴은 중괄호 `{}`를 사용하여 객체를 정의합니다.

```
let person = {
    name: "John",
```

```
    age: 30,
    job: "Developer"
};

console.log(person); // { name: "John", age: 30, job: "Developer" }
```

프로퍼티와 메서드

프로퍼티는 객체의 속성을 나타내며, 메서드는 객체의 동작을 정의하는 함수입니다.

프로퍼티 접근

프로퍼티에 접근하는 방법에는 점 표기법과 대괄호 표기법이 있습니다.

```
// 점 표기법
console.log(person.name); // "John"

// 대괄호 표기법
console.log(person["age"]); // 30
```

프로퍼티 추가 및 수정

프로퍼티를 추가하거나 수정할 수 있습니다.

```
// 프로퍼티 추가
person.email = "john@example.com";
console.log(person.email); // "john@example.com"

// 프로퍼티 수정
person.age = 31;
console.log(person.age); // 31
```

메서드 정의

메서드는 객체의 동작을 정의하는 함수입니다.

```
let person = {
    name: "John",
    age: 30,
    job: "Developer",
    greet: function() {
        console.log(`Hello, my name is ${this.name}.`);
```

```
    }
};

person.greet(); // "Hello, my name is John."
```

this 키워드

`this` 키워드는 객체의 현재 인스턴스를 참조합니다. `this`는 함수가 호출될 때 결정됩니다.

```
let car = {
    brand: "Toyota",
    model: "Corolla",
    start: function() {
        console.log(`Starting the ${this.brand} ${this.model}`);
    }
};

car.start(); // "Starting the Toyota Corolla"
```

this의 다양한 사용 사례

1. **전역 컨텍스트에서의 this**
   전역 스코프에서 `this`는 전역 객체(브라우저에서는 `window` 객체)를 참조합니다.

```
console.log(this); // Window {...}
```

2. **객체 메서드에서의 this**
   객체 메서드 내부에서 `this`는 해당 메서드를 호출한 객체를 참조합니다.

```
let user = {
    name: "Alice",
    greet: function() {
        console.log(`Hello, ${this.name}`);
    }
};

user.greet(); // "Hello, Alice"
```

3. **함수에서의 this**
   일반 함수 내부에서 `this`는 전역 객체를 참조합니다. (엄격 모드에서는 `undefined`)

```
function showThis() {
    console.log(this);
```

```
}

showThis(); // Window {...}
```

4. **이벤트 핸들러에서의 this**
   이벤트 핸들러 내부에서 `this`는 이벤트가 발생한 요소를 참조합니다.

```
document.getElementById("myButton").addEventListener("click", function() {
    console.log(this); // <button id="myButton">...</button>
});
```

5. **화살표 함수에서의 this**
   화살표 함수는 자신만의 `this` 바인딩을 가지지 않고, 자신이 정의된 외부 스코프의 `this`를 사용합니다.

```
let person = {
    name: "Bob",
    greet: function() {
        setTimeout(() => {
            console.log(`Hello, ${this.name}`);
        }, 1000);
    }
};

person.greet(); // "Hello, Bob"
```

배열

배열 선언

배열은 자바스크립트에서 순서가 있는 데이터의 컬렉션입니다. 배열은 대괄호 `[]`를 사용하여 선언합니다.

```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits); // ["apple", "banana", "cherry"]
```

배열은 다양한 데이터 타입을 포함할 수 있습니다.

```
let mixedArray = [1, "hello", true, null];
console.log(mixedArray); // [1, "hello", true, null]
```

배열 메서드

push

`push` 메서드는 배열의 끝에 하나 이상의 요소를 추가합니다.

```
let fruits = ["apple", "banana"];
fruits.push("cherry");
console.log(fruits); // ["apple", "banana", "cherry"]
```

pop

`pop` 메서드는 배열의 끝에서 하나의 요소를 제거하고, 제거된 요소를 반환합니다.

```
let fruits = ["apple", "banana", "cherry"];
let lastFruit = fruits.pop();
console.log(lastFruit); // "cherry"
console.log(fruits); // ["apple", "banana"]
```

shift

`shift` 메서드는 배열의 첫 번째 요소를 제거하고, 제거된 요소를 반환합니다.

```
let fruits = ["apple", "banana", "cherry"];
let firstFruit = fruits.shift();
console.log(firstFruit); // "apple"
console.log(fruits); // ["banana", "cherry"]
```

unshift

`unshift` 메서드는 배열의 앞에 하나 이상의 요소를 추가합니다.

```
let fruits = ["banana", "cherry"];
fruits.unshift("apple");
console.log(fruits); // ["apple", "banana", "cherry"]
```

map

`map` 메서드는 배열의 각 요소에 대해 주어진 함수를 호출하고, 그 결과를 모아 새로운 배열을 반환합니다.

```
let numbers = [1, 2, 3];
let doubled = numbers.map(function(number) {
```

```
    return number * 2;
});
console.log(doubled); // [2, 4, 6]
```

화살표 함수로도 사용할 수 있습니다.

```
let doubled = numbers.map(number => number * 2);
console.log(doubled); // [2, 4, 6]
```

filter

`filter` 메서드는 배열의 각 요소에 대해 주어진 함수를 호출하여 조건을 만족하는 요소들만 모아 새로운 배열을 반환합니다.

```
let numbers = [1, 2, 3, 4, 5];
let evenNumbers = numbers.filter(function(number) {
    return number % 2 === 0;
});
console.log(evenNumbers); // [2, 4]
```

화살표 함수로도 사용할 수 있습니다.

```
let evenNumbers = numbers.filter(number => number % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

forEach

`forEach` 메서드는 배열의 각 요소에 대해 주어진 함수를 한 번씩 실행합니다. 반환 값이 없습니다.

```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach(function(fruit) {
    console.log(fruit);
});
// "apple"
// "banana"
// "cherry"
```

화살표 함수로도 사용할 수 있습니다.

```
fruits.forEach(fruit => console.log(fruit));
// "apple"
```

```
// "banana"
// "cherry"
```

find

`find` 메서드는 주어진 함수의 조건을 만족하는 첫 번째 요소를 반환합니다. 조건을 만족하는 요소가 없으면 `undefined`를 반환합니다.

```
let numbers = [1, 2, 3, 4, 5];
let firstEven = numbers.find(function(number) {
    return number % 2 === 0;
});
console.log(firstEven); // 2
```

화살표 함수로도 사용할 수 있습니다.

```
let firstEven = numbers.find(number => number % 2 === 0);
console.log(firstEven); // 2
```

reduce

`reduce` 메서드는 배열의 각 요소에 대해 주어진 함수를 호출하여 단일 값을 생성합니다.

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce(function(accumulator, currentValue) {
    return accumulator + currentValue;
}, 0);
console.log(sum); // 15
```

화살표 함수로도 사용할 수 있습니다.

```
let sum = numbers.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(sum); // 15
```

데이터 포맷 변천사

XML (eXtensible Markup Language)

개요

XML은 1990년대 후반에 등장하여 데이터 저장과 전송을 위한 포맷으로 널리 사용되었습니다. 데이터와 데이터를 설명하는 태그를 사용하여 구조화된 데이터를 표현할 수 있습니다.

예시

```
<person>
    <name>John</name>
    <age>30</age>
    <job>Developer</job>
</person>
```

HTML (HyperText Markup Language)

개요

HTML은 웹 페이지를 작성하기 위한 언어로, XML과 유사하게 태그를 사용하여 문서의 구조와 콘텐츠를 정의합니다. 주로 웹 브라우저에서 웹 페이지를 표시하는 데 사용됩니다.

예시

```
<!DOCTYPE html>
<html>
<head>
    <title>Example</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is an example HTML page.</p>
</body>
</html>
```

.xlsx (Excel Spreadsheet)

개요

.xlsx 파일 형식은 마이크로소프트 엑셀에서 사용되는 스프레드시트 파일 포맷입니다. 데이터를 표 형식으로 저장하며, 각 셀은 수식, 텍스트, 숫자 등을 포함할 수 있습니다.

예시

Name	Age	Job
John	30	Developer
Jane	25	Designer

.csv (Comma-Separated Values)

개요

.csv 파일 형식은 데이터를 쉼표로 구분하여 저장하는 단순한 텍스트 파일 포맷입니다. 엑셀과 같은 스프레드시트 프로그램이나 데이터베이스와 쉽게 호환됩니다.

예시

```
Name,Age,Job
John,30,Developer
Jane,25,Designer
```

JSON (JavaScript Object Notation)

개요

JSON은 2000년대 중반에 등장한 데이터 포맷으로, 경량 데이터 교환 형식입니다. 주로 웹 애플리케이션에서 서버와 클라이언트 간 데이터를 주고받기 위해 사용됩니다. JavaScript 객체 표기법을 기반으로 하여 가독성과 사용이 용이합니다.

예시

```
{
    "person": {
        "name": "John",
        "age": 30,
        "job": "Developer"
    }
}
```

데이터 포맷 별 특징

1. **XML**: 데이터의 계층적 구조를 정의하는 데 사용되었지만, 가독성과 데이터 처리에서 복잡함이 있음.

2. **HTML**: 주로 웹 페이지를 작성하는 데 사용되며, 데이터보다는 문서 구조와 콘텐츠에 집중.
3. **.xlsx**: 구조화된 데이터를 표 형식으로 관리하고 분석하는 데 강력하지만, 파일 크기와 호환성 문제 존재.
4. **.csv**: 간단하고 가벼운 데이터 저장 형식으로, 데이터베이스와 호환성이 높음.
5. **JSON**: 경량화된 데이터 교환 포맷으로, 웹 애플리케이션에서 효율적이고 사용이 간편함.

각 데이터 포맷은 그 당시의 요구에 따라 발전해왔으며, 현재는 JSON이 웹 개발과 데이터 교환에서 널리 사용.

JSON 고급이해

JSON (JavaScript Object Notation)은 데이터를 저장하고 전송하기 위해 널리 사용되는 경량 데이터 교환 형식. JSON은 사람이 읽기 쉽고, 기계가 해석하고 생성하기 쉬운 텍스트 형식으로, JavaScript 객체 표기법을 기반으로 하지만 언어 독립적.

JSON의 특징:

- **텍스트 형식**: 사람이 읽기 쉽고, 편집이 간편합니다.
- **언어 독립적**: 대부분의 프로그래밍 언어에서 JSON을 파싱하고 생성할 수 있는 라이브러리를 제공합니다.
- **구조적**: 데이터 구조를 표현하기 위해 객체(object)와 배열(array)을 사용합니다.

JSON 구조

객체 (Object)

```
{
  "name": "Alice",
  "age": 25,
  "isStudent": false
}
```

배열 (Array)

```
[
  {
    "name": "Alice",
    "age": 25
  },
  {
    "name": "Bob",
```

```
    "age": 30
  }
]
```

JSON의 기본 기능

JSON 문자열로 변환

```
const obj = { name: "Alice", age: 25, isStudent: false };
const jsonString = JSON.stringify(obj);
console.log(jsonString);
// {"name":"Alice","age":25,"isStudent":false}
```

JSON 문자열을 객체로 변환

```
const jsonString = '{"name":"Alice","age":25,"isStudent":false}';
const obj = JSON.parse(jsonString);
console.log(obj);
// { name: 'Alice', age: 25, isStudent: false }
```

구조 분해 할당 (Destructuring)

객체 구조 분해 할당

```
const person = {
  name: 'Alice',
  age: 25,
  occupation: 'Engineer'
};

const { name, age, occupation } = person;

console.log(name); // 'Alice'
console.log(age); // 25
console.log(occupation); // 'Engineer'
```

별칭 사용

```
const { name: fullName, age: years } = person;
```

```
console.log(fullName); // 'Alice'
console.log(years); // 25
```

기본값 설정

```
const { name, age = 30 } = person;

console.log(name); // 'Alice'
console.log(age); // 30
```

중첩 객체 구조 분해

```
const person = {
  name: 'Alice',
  address: {
    city: 'Wonderland',
    zip: '12345'
  }
};

const { name, address: { city, zip } } = person;

console.log(city); // 'Wonderland'
console.log(zip); // '12345'
```

배열 구조 분해 할당

기본 예제

```
const colors = ['red', 'green', 'blue'];

const [first, second, third] = colors;

console.log(first); // 'red'
console.log(second); // 'green'
console.log(third); // 'blue'
```

기본값 설정

```
const [first, second = 'green', third = 'blue'] = ['red'];
```

```
console.log(first); // 'red'
console.log(second); // 'green'
console.log(third); // 'blue'
```

나머지 요소 추출

```
const [first, ...rest] = ['red', 'green', 'blue', 'yellow'];

console.log(first); // 'red'
console.log(rest); // ['green', 'blue', 'yellow']
```

중첩 배열 구조 분해

```
const nestedArray = [1, [2, 3], 4];

const [a, [b, c], d] = nestedArray;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(d); // 4
```

구조화 (Structuring)

객체 구조화

```
const name = 'Alice';
const age = 25;
const occupation = 'Engineer';

const person = { name, age, occupation };

console.log(person);
// { name: 'Alice', age: 25, occupation: 'Engineer' }
```

배열 구조화

```
const first = 'red';
const second = 'green';
const third = 'blue';
```

```
const colors = [first, second, third];

console.log(colors);
// ['red', 'green', 'blue']
```

JSON 활용 예제

API 데이터 처리

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    const { name, age } = data;
    console.log(`Name: ${name}, Age: ${age}`);
  })
  .catch(error => console.error('Error:', error));
```

로컬 스토리지 활용

```
const user = {
  name: 'Alice',
  age: 25
};

// JSON 문자열로 변환하여 저장
localStorage.setItem('user', JSON.stringify(user));

// 저장된 JSON 문자열을 객체로 변환하여 사용
const storedUser = JSON.parse(localStorage.getItem('user'));
console.log(storedUser);
// { name: 'Alice', age: 25 }
```

JSON 게임 데이터 예시

주요 데이터:

1. 캐릭터 **(Character)**
2. 아이템 **(Item)**
3. 퀘스트 **(Quest)**
4. 몬스터 **(Monster)**

캐릭터 데이터


```
{
  "characters": [
    {
      "id": 1,
      "name": "Archer",
      "level": 5,
      "class": "Ranger",
      "stats": {
        "strength": 8,
        "dexterity": 15,
        "intelligence": 7
      },
      "inventory": [
        {
          "itemId": 101,
          "quantity": 1
        },
        {
          "itemId": 102,
          "quantity": 5
        }
      ],
      "currentQuest": 201
    },
    {
      "id": 2,
      "name": "Mage",
      "level": 7,
      "class": "Sorcerer",
      "stats": {
        "strength": 4,
        "dexterity": 10,
        "intelligence": 18
      },
      "inventory": [
        {
          "itemId": 103,
          "quantity": 3
        }
      ],
      "currentQuest": null
    }
  ]
}
```

아이템 데이터


```
{
  "items": [
    {
      "id": 101,
      "name": "Short Bow",
      "type": "Weapon",
      "damage": 10,
      "price": 100
    },
    {
      "id": 102,
      "name": "Health Potion",
      "type": "Consumable",
      "healing": 50,
      "price": 10
    },
    {
      "id": 103,
      "name": "Magic Staff",
      "type": "Weapon",
      "damage": 25,
      "price": 300
    }
  ]
}
```


퀘스트 데이터


```
{
  "quests": [
    {
      "id": 201,
      "name": "Defeat the Goblin King",
      "description": "Defeat the Goblin King in the Dark Forest",
      "rewards": {
        "experience": 500,
        "items": [
          {
            "itemId": 104,
            "quantity": 1
          }
        ]
      }
    },
    {
```

```
      "id": 202,
      "name": "Collect Healing Herbs",
      "description": "Collect 10 healing herbs from the meadow",
      "rewards": {
        "experience": 200,
        "items": [
          {
            "itemId": 102,
            "quantity": 10
          }
        ]
      }
    }
  ]
}
```

몬스터 데이터

```
{
  "monsters": [
    {
      "id": 301,
      "name": "Goblin",
      "level": 3,
      "stats": {
        "strength": 6,
        "dexterity": 5,
        "intelligence": 2
      },
      "loot": [
        {
          "itemId": 102,
          "quantity": 1,
          "dropRate": 0.5
        }
      ]
    },
    {
      "id": 302,
      "name": "Goblin King",
      "level": 10,
      "stats": {
        "strength": 15,
        "dexterity": 10,
        "intelligence": 5
      },
      "loot": [
```

```
        {
          "itemId": 104,
          "quantity": 1,
          "dropRate": 1.0
        }
      ]
    }
  ]
}
```

게임 데이터 설명

1. **캐릭터 데이터**: 각 캐릭터의 정보, 통계, 인벤토리, 현재 진행 중인 퀘스트 등을 포함합니다.
2. **아이템 데이터**: 각 아이템의 ID, 이름, 유형, 공격력, 가격 등의 속성을 포함합니다.
3. **퀘스트 데이터**: 퀘스트의 ID, 이름, 설명, 보상 등을 포함합니다.
4. **몬스터 데이터**: 각 몬스터의 정보, 레벨, 통계, 전리품 및 드랍률 등을 포함합니다.

JSON을 이용한 데이터 구조화의 이점

- **가독성**: JSON은 사람이 읽기 쉬운 형식으로 데이터를 표현합니다.
- **유연성**: 복잡한 데이터 구조를 표현할 수 있으며, 다양한 데이터 유형을 지원합니다.
- **호환성**: 대부분의 프로그래밍 언어에서 JSON을 쉽게 파싱하고 생성할 수 있는 라이브러리를 제공합니다.
- **표준화**: 데이터 교환을 위한 표준 형식으로, REST API, 웹 애플리케이션, 모바일 앱 등 다양한 환경에서 널리 사용됩니다.

데이터 전송: 네트워크 프로토콜

1. Protocol(프로토콜) 개요

프로토콜은 컴퓨터 네트워크에서 데이터를 주고받는 규칙과 약속입니다. 마치 사람들이 서로 소통하기 위해 사용하는 언어와 같습니다. 프로토콜을 통해 컴퓨터들이 서로 이해하고 소통할 수 있습니다.

2. IP와 Port

- **IP (Internet Protocol)**: 인터넷에서 장치들을 식별하는 고유 주소입니다. 예를 들어, 192.168.0.1 같은 숫자 형태로 나타납니다.
- **Port**: 특정 프로그램이나 서비스에 연결되는 통로입니다. 한 컴퓨터에서 여러 서비스가 동시에 동작할 수 있도록 합니다. 예를 들어, 웹 서버는 80번 포트를 사용합니다.

IP 주소는 집 주소, Port는 각 방의 번호라고 생각하면 됩니다. 한 집에 여러 방이 있듯이, 한 IP 주소에 여러 Port가 있을 수 있습니다.

3. TCP/IP

TCP/IP는 인터넷의 핵심 프로토콜입니다.

- **TCP (Transmission Control Protocol)**: 데이터를 신뢰성 있게 전달하기 위한 프로토콜입니다. 데이터가 순서대로, 손실 없이 전달되도록 보장합니다.
- **IP (Internet Protocol)**: 데이터를 목적지까지 전달하기 위한 주소 지정 프로토콜입니다.

TCP는 데이터를 안전하게 전달하는 우체부, IP는 데이터를 목적지까지 안내하는 지도라고 생각할 수 있습니다.

4. HTTP (Hypertext Transfer Protocol)

HTTP는 웹 페이지를 주고받기 위한 프로토콜입니다.

- **REST (Representational State Transfer)**: REST는 자원을 URI로 식별하고, HTTP 메서드(GET, POST, PUT, DELETE)를 사용해 자원을 조작하는 아키텍처 스타일입니다.
- **AJAX (Asynchronous JavaScript and XML)**: 웹 페이지를 새로고침하지 않고 데이터를 주고받을 수 있게 해주는 기술입니다. AJAX를 통해 웹 애플리케이션이 더 인터랙티브해집니다.

간단한 AJAX 예제:

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

HTTP는 우리가 웹 브라우저에서 웹 페이지를 요청하고 응답을 받는 일련의 과정입니다. REST는 주소를 통해 특정 자원을 요청하고, AJAX는 웹 페이지를 새로고침하지 않고도 데이터를 주고받을 수 있게 해줍니다.

5. HTTPS (HTTP Secure)

HTTPS는 HTTP에 보안을 추가한 프로토콜입니다. SSL/TLS를 사용해 데이터를 암호화하여 안전하게 주고받습니다.

HTTPS는 HTTP에 보안 잠금장치를 더한 버전입니다. 데이터를 보내기 전에 자물쇠로 잠가서 안전하게 전송합니다.

6. FTP (File Transfer Protocol)

FTP는 파일을 주고받기 위한 프로토콜입니다. 주로 파일 서버에 파일을 업로드하거나 다운로드할 때 사용됩니다.

FTP는 파일을 전송하는 택배 서비스와 같습니다. 택배 상자를 안전하게 보내고 받습니다.

7. SSH (Secure Shell)

SSH는 원격 컴퓨터에 안전하게 접속하기 위한 프로토콜입니다. 터미널을 통해 서버에 접속하여 명령을 실행할 수 있습니다.

- **SFTP (SSH File Transfer Protocol)**: SSH를 사용해 파일을 전송하는 프로토콜입니다.
- **SCP (Secure Copy Protocol)**: SSH를 사용해 파일을 복사하는 프로토콜입니다.

간단한 SCP 예제:

```
scp localfile.txt user@remotehost:/path/to/destination
```

SSH는 원격으로 다른 컴퓨터에 접속할 수 있는 보안 통로입니다. SFTP와 SCP는 이 통로를 통해 파일을 전송하거나 복사할 수 있게 해줍니다.

8. Websocket

Websocket은 실시간 양방향 통신을 위한 프로토콜입니다. 채팅 애플리케이션이나 실시간 데이터 업데이트가 필요한 경우에 사용됩니다.

간단한 Websocket 예제:

```
const socket = new WebSocket('ws://example.com/socket');

socket.onopen = () => {
    socket.send('Hello Server!');
};

socket.onmessage = (event) => {
    console.log('Message from server:', event.data);
};
```

Websocket은 전화 통화와 같습니다. 양방향으로 실시간 대화를 주고받을 수 있습니다.

9. UDP (User Datagram Protocol)

UDP는 TCP와 달리 신뢰성보다는 속도를 중시하는 프로토콜입니다. 데이터가 손실되거나 순서가 뒤바뀔 수 있지만, 빠르게 전달됩니다. 실시간 스트리밍이나 온라인 게임에서 주로 사용됩니다.

UDP는 메모를 던져주는 것과 같습니다. 빠르게 전달되지만, 메모가 중간에 사라지거나 순서가 뒤바뀔 수 있습니다.

자바스크립트 활용

클래스

클래스 기본 구조

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`안녕하세요, 제 이름은 ${this.name}이고 ${this.age}살입니다.`);
  }
}

const person = new Person("김철수", 30);
person.sayHello(); // "안녕하세요, 제 이름은 김철수이고 30살입니다." 출력
```

정적 메서드

```
class MathOperations {
  static add(x, y) {
    return x + y;
  }
}

console.log(MathOperations.add(5, 3)); // 8 출력
```

상속

```
class Animal {
  constructor(name) {
    this.name = name;
  }
```

```
  speak() {
    console.log(`${this.name}이(가) 소리를 냅니다.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name}이(가) 멍멍 짖습니다.`);
  }
}

const dog = new Dog("뽀삐");
dog.speak(); // "뽀삐이(가) 멍멍 짖습니다." 출력
```

getter와 setter

```
class Rectangle {
  constructor(width, height) {
    this._width = width;
    this._height = height;
  }

  get area() {
    return this._width * this._height;
  }

  set width(value) {
    if (value > 0) {
      this._width = value;
    }
  }
}

const rect = new Rectangle(5, 10);
console.log(rect.area); // 50 출력
rect.width = 7;
console.log(rect.area); // 70 출력
```

프라이빗 필드 (ES2022)

```
class Circle {
  #radius;

  constructor(radius) {
    this.#radius = radius;
```

```
  }

  get area() {
    return Math.PI * this.#radius ** 2;
  }
}

const circle = new Circle(5);
console.log(circle.area); // 약 78.54 출력
// console.log(circle.#radius); // 에러 발생: 프라이빗 필드에 접근 불가
```

자바스크립트 클래스의 주요 특징:

- 클래스는 `class` 키워드로 선언합니다.
- `constructor` 메서드는 객체 초기화를 위해 사용됩니다.
- 메서드를 정의할 때 `function` 키워드를 사용하지 않습니다.
- `extends` 키워드로 상속을 구현합니다.
- `static` 키워드로 정적 메서드를 정의할 수 있습니다.
- `get`과 `set` 키워드로 getter와 setter를 정의할 수 있습니다.
- `#` 접두사로 프라이빗 필드를 선언할 수 있습니다 (ES2022 이상).

클래스는 자바스크립트의 프로토타입 기반 상속을 더 쉽게 사용할 수 있게 해주는 문법. 내부적으로는 여전히 프로토타입을 사용하지만, 다른 객체 지향 언어와 유사한 문법으로 코드를 작성할 수 있게 해줌.

비동기 프로그래밍

콜백 함수

콜백 함수는 다른 함수에 인수로 전달되어 실행되는 함수입니다. 주로 비동기 작업이 완료된 후 실행될 작업을 정의할 때 사용됩니다.

```
function fetchData(callback) {
    setTimeout(() => {
        callback("데이터 로드 완료");
    }, 2000);
}

fetchData((message) => {
    console.log(message); // "데이터 로드 완료" (2초 후 출력)
});
```

Promise

Promise는 비동기 작업의 완료 또는 실패를 나타내는 객체입니다. `then`, `catch` 메서드를 사용하여 작업이 완료되었을 때와 실패했을 때의 동작을 정의할 수 있습니다.

```
const fetchData = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("데이터 로드 완료");
    }, 2000);
});

fetchData
    .then((message) => console.log(message)) // "데이터 로드 완료" (2초 후 출력)
    .catch((error) => console.error(error));
```

async/await

`async` 함수는 항상 Promise를 반환하며, `await` 키워드는 Promise가 처리될 때까지 기다립니다. 이를 통해 비동기 코드를 동기 코드처럼 작성할 수 있습니다.

```
async function fetchData() {
    let response = await new Promise((resolve) => {
        setTimeout(() => resolve("데이터 로드 완료"), 2000);
    });
    console.log(response); // "데이터 로드 완료" (2초 후 출력)
}

fetchData();
```

모듈 시스템

ES6 모듈

ES6 모듈은 `export`와 `import` 키워드를 사용하여 모듈을 정의하고 가져옵니다. 이를 통해 코드의 재사용성과 유지보수성을 높일 수 있습니다.

모듈 정의 (module.js)

```
export const PI = 3.14;

export function add(a, b) {
    return a + b;
}

export default class Calculator {
    constructor() {
```

```
        console.log("Calculator 객체 생성");
    }
    subtract(a, b) {
        return a - b;
    }
}
```

모듈 가져오기 (main.js)

```
import { PI, add } from './module.js';
import Calculator from './module.js';

console.log(PI); // 3.14
console.log(add(2, 3)); // 5

const calc = new Calculator();
console.log(calc.subtract(5, 2)); // 3
```

CommonJS

CommonJS 모듈 시스템은 Node.js에서 사용되며, `require`와 `module.exports`를 사용하여 모듈을 정의하고 가져옵니다.

모듈 정의 (module.js)

```
const PI = 3.14;

function add(a, b) {
    return a + b;
}

class Calculator {
    constructor() {
        console.log("Calculator 객체 생성");
    }
    subtract(a, b) {
        return a - b;
    }
}

module.exports = { PI, add, Calculator };
```

모듈 가져오기 (main.js)

```
const { PI, add, Calculator } = require('./module.js');
```

```
console.log(PI); // 3.14
console.log(add(2, 3)); // 5

const calc = new Calculator();
console.log(calc.subtract(5, 2)); // 3
```

에러 처리

try-catch

`try-catch` 구문은 코드에서 발생하는 예외를 처리하는 방법입니다. `try` 블록 내부에서 예외가 발생하면 `catch` 블록이 실행됩니다.

```
try {
    let result = 10 / 0;
    console.log(result); // Infinity
    throw new Error("임의의 에러 발생");
} catch (error) {
    console.error(error.message); // "임의의 에러 발생"
} finally {
    console.log("이 블록은 항상 실행됩니다.");
}
```

throw

`throw` 키워드를 사용하여 사용자 정의 예외를 발생시킬 수 있습니다.

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("0으로 나눌 수 없습니다.");
    }
    return a / b;
}

try {
    console.log(divide(10, 2)); // 5
    console.log(divide(10, 0)); // 오류 발생
} catch (error) {
    console.error(error.message); // "0으로 나눌 수 없습니다."
}
```

에러 객체

에러 객체는 예외의 정보를 담고 있으며, `name`, `message`, `stack` 등의 속성을 가집니다.

```
try {
    throw new Error("임의의 에러");
} catch (error) {
    console.log(error.name); // "Error"
    console.log(error.message); // "임의의 에러"
    console.log(error.stack); // 에러 발생 위치 스택 추적 정보
}
```

브라우저 객체와 주요 키워드 및 사용법

Window 객체

`Window` 객체는 브라우저 창을 나타내며, 전역 객체로서 모든 전역 변수와 함수의 컨테이너 역할을 합니다.

주요 메서드와 프로퍼티

- `alert(message)` : 경고 메시지 창을 표시합니다.
- `confirm(message)` : 확인 대화 상자를 표시하고, 사용자의 확인 여부를 불리언 값으로 반환합니다.
- `prompt(message, default)` : 입력 대화 상자를 표시하고, 사용자가 입력한 문자열을 반환합니다.
- `window.innerWidth` : 창의 내부 너비를 반환합니다.
- `window.innerHeight` : 창의 내부 높이를 반환합니다.

```
// 경고 메시지 창
window.alert("Hello, World!");

// 확인 대화 상자
let userConfirmed = window.confirm("Are you sure?");
console.log(userConfirmed); // true 또는 false

// 입력 대화 상자
let userInput = window.prompt("What is your name?", "John Doe");
console.log(userInput); // 사용자가 입력한 문자열

// 창의 크기
console.log(window.innerWidth); // 창의 내부 너비
console.log(window.innerHeight); // 창의 내부 높이
```

Document 객체

`Document` 객체는 HTML 문서를 나타내며, DOM(Document Object Model)의 진입점 역할을 합니다.

주요 메서드와 프로퍼티

- `document.getElementById(id)` : 지정된 ID를 가진 요소를 반환합니다.
- `document.querySelector(selector)` : CSS 선택자를 사용하여 일치하는 첫 번째 요소를 반환합니다.
- `document.querySelectorAll(selector)` : CSS 선택자를 사용하여 일치하는 모든 요소를 반환합니다.
- `document.createElement(tagName)` : 지정된 태그 이름의 새로운 요소를 생성합니다.

```
// 요소 선택
let element = document.getElementById("myElement");
let firstParagraph = document.querySelector("p");
let allDivs = document.querySelectorAll("div");

// 새로운 요소 생성
let newDiv = document.createElement("div");
newDiv.textContent = "Hello, World!";
document.body.appendChild(newDiv);
```

Navigator 객체

`Navigator` 객체는 브라우저의 정보를 나타냅니다.

주요 메서드와 프로퍼티

- `navigator.userAgent` : 브라우저의 사용자 에이전트 문자열을 반환합니다.
- `navigator.language` : 브라우저의 기본 언어를 반환합니다.
- `navigator.onLine` : 브라우저가 온라인 상태인지 여부를 불리언 값으로 반환합니다.

```
console.log(navigator.userAgent); // 브라우저 사용자 에이전트 문자열
console.log(navigator.language); // 브라우저의 기본 언어
console.log(navigator.onLine); // 온라인 상태 여부 (true 또는 false)
```

Location 객체

`Location` 객체는 현재 문서의 URL 정보를 나타냅니다.

주요 메서드와 프로퍼티

- `location.href` : 현재 페이지의 URL을 반환하거나 설정합니다.
- `location.reload()` : 현재 페이지를 다시 로드합니다.
- `location.assign(url)` : 지정된 URL로 이동합니다.

```
console.log(location.href); // 현재 페이지의 URL

// 페이지를 다시 로드
location.reload();

// 다른 페이지로 이동
location.assign("https://www.example.com");
```

Node.js에서의 자바스크립트 주요 키워드 및 사용법

Node.js 개요

Node.js는 서버 측에서 자바스크립트를 실행할 수 있게 해주는 런타임 환경입니다. V8 JavaScript 엔진을 기반으로 하며, 비동기 이벤트 주도 방식으로 높은 성능을 제공합니다. Node.js는 주로 웹 서버, API 서버, 실시간 애플리케이션 등을 개발하는 데 사용됩니다.

주요 키워드 및 사용법

모듈 시스템

Node.js는 CommonJS 모듈 시스템을 사용합니다. 모듈을 정의하고 가져오는 방법을 통해 코드를 모듈화하고 재사용할 수 있습니다.

모듈 정의
모듈을 정의하려면 `module.exports`를 사용하여 모듈에서 내보낼 값을 지정합니다.

```
// math.js
const PI = 3.14;

function add(a, b) {
    return a + b;
}

module.exports = { PI, add };
```

모듈 가져오기
`require` 함수를 사용하여 다른 파일의 모듈을 가져옵니다.

```
// app.js
const math = require('./math');

console.log(math.PI); // 3.14
console.log(math.add(2, 3)); // 5
```

비동기 프로그래밍

Node.js는 비동기 프로그래밍을 쉽게 할 수 있도록 콜백 함수, Promise, async/await 등을 지원합니다.

콜백 함수

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});
```

Promise

```
const fs = require('fs').promises;

fs.readFile('example.txt', 'utf8')
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.error(err);
    });
```

async/await

```
const fs = require('fs').promises;

async function readFile() {
    try {
        const data = await fs.readFile('example.txt', 'utf8');
        console.log(data);
    } catch (err) {
        console.error(err);
    }
```

```
}

readFile();
```

이벤트

Node.js는 이벤트 기반 아키텍처를 가지고 있으며, `events` 모듈을 통해 이벤트를 생성하고 처리할 수 있습니다.

이벤트 생성 및 처리

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();

// 이벤트 핸들러 정의
eventEmitter.on('sayHello', () => {
    console.log('Hello, world!');
});

// 이벤트 발생
eventEmitter.emit('sayHello');
```

HTTP 서버

Node.js를 사용하여 간단한 HTTP 서버를 만들 수 있습니다.

HTTP 서버 생성

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello, world!\n');
});

const port = 3000;
server.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

파일 시스템

Node.js의 `fs` 모듈을 사용하여 파일을 읽고 쓸 수 있습니다.

파일 읽기

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});
```

파일 쓰기

```
const fs = require('fs');

const content = 'Hello, world!';
fs.writeFile('example.txt', content, err => {
    if (err) {
        console.error(err);
        return;
    }
    console.log('File written successfully');
});
```

스트림

Node.js의 스트림은 데이터 처리를 효율적으로 할 수 있도록 도와줍니다. 스트림은 읽기, 쓰기, 변환 등의 작업을 비동기적으로 수행합니다.

읽기 스트림

```
const fs = require('fs');

const readStream = fs.createReadStream('example.txt', 'utf8');
readStream.on('data', chunk => {
    console.log(chunk);
});
```

쓰기 스트림

```
const fs = require('fs');

const writeStream = fs.createWriteStream('output.txt');
writeStream.write('Hello, ');
```

```
writeStream.write('world!');
writeStream.end();
writeStream.on('finish', () => {
    console.log('Write completed.');
});
```

패키지 관리

Node.js의 패키지 관리자는 npm(Node Package Manager)입니다. npm을 사용하여 패키지를 설치, 업데이트, 제거할 수 있습니다.

패키지 설치

```
npm install express
```

패키지 사용

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Next.js 개요 및 설치

Next.js란 무엇인가?

Next.js는 React를 기반으로 하는 프레임워크로, 서버 사이드 렌더링(SSR)과 정적 사이트 생성(SSG)을 지원합니다. 이는 SEO 친화적인 웹 애플리케이션을 쉽게 만들 수 있게 해주며, 성능 최적화와 개발 편의성을 높이는 다양한 기능을 제공합니다.

주요 특징

- **서버 사이드 렌더링(SSR)**: 초기 페이지 로드를 서버에서 처리하여 빠른 렌더링과 SEO 최적화를 가능하게 합니다.
- **정적 사이트 생성(SSG)**: 빌드 시점에 HTML을 생성하여 성능을 극대화하고, CDN을 통해 배포할 수 있습니다.

- **자동 코드 분할**: 각 페이지는 필요할 때만 로드되므로 초기 로드 시간이 단축됩니다.
- **API 라우트**: 백엔드 로직을 작성할 수 있는 서버리스 함수(즉, API 엔드포인트)를 쉽게 생성할 수 있습니다.
- **CSS 및 Sass 지원**: CSS 모듈, Sass, Styled JSX 등 다양한 스타일링 옵션을 지원합니다.
- **Typescript 지원**: 타입스크립트를 사용하여 더 안전한 코드 작성이 가능합니다.
- **Fast Refresh**: 코드 변경 시 전체 페이지가 아닌 변경된 부분만을 빠르게 갱신하여 개발자 경험을 향상시킵니다.

Next.js 설치 방법 (v14 이상)

Next.js v14 이상에서는 새로운 `app` 디렉터리를 사용하여 라우트를 정의할 수 있습니다. 이 디렉터리를 통해 더욱 직관적이고 강력한 라우팅 기능을 제공합니다.

사전 준비

- Node.js와 npm이 설치되어 있어야 합니다. Node.js를 설치하면 npm도 함께 설치됩니다.
- Node.js가 설치되어 있지 않다면 Node.js 공식 웹사이트에서 설치합니다.

Next.js 설치

1. **새 프로젝트 디렉터리 생성**

```
mkdir my-next-app
cd my-next-app
```

2. **Next.js 및 React 설치**

```
npx create-next-app@latest --experimental-app
# 또는 특정 버전을 설치하려면
# npx create-next-app@14
```

   위 명령어를 실행하면 프로젝트 이름, 타입스크립트 사용 여부, ESLint 설정 여부 등의 옵션을 선택할 수 있는 프롬프트가 나타납니다.

3. **프로젝트 디렉터리로 이동**

```
cd my-next-app
```

4. **개발 서버 시작**

```
npm run dev
```

기본적으로 개발 서버는 `http://localhost:3000`에서 실행됩니다. 브라우저에서 해당 URL을 열어 Next.js 애플리케이션을 확인할 수 있습니다.

프로젝트 구조

Next.js v14 이상에서는 `app` 디렉터리를 사용하여 라우트를 정의합니다.

```
my-next-app/
├── node_modules/
├── public/
│   └── favicon.ico
├── styles/
│   ├── globals.css
│   └── Home.module.css
├── app/
│   ├── api/
│   │   └── hello.js
│   ├── layout.js
│   ├── page.js
│   └── about/
│       └── page.js
├── .gitignore
├── package.json
├── README.md
└── next.config.js
```

- `app/`: Next.js v14 이상의 새로운 라우팅 시스템을 위한 디렉터리입니다.
  - `page.js`: 기본 라우트(`http://localhost:3000/`)를 위한 파일입니다.
  - `about/page.js`: `/about` 라우트를 위한 파일입니다.
  - `layout.js`: 모든 페이지에 공통적으로 적용되는 레이아웃을 정의하는 파일입니다.
- `public/`: 정적 파일을 저장하는 디렉터리입니다.
- `styles/`: CSS 파일을 저장하는 디렉터리입니다.

예제 코드

app/page.js

```
export default function HomePage() {
  return (
    <div>
      <h1>Welcome to My Next.js App</h1>
      <p>This is the home page.</p>
```

```
    </div>
  );
}
```

app/about/page.js

```
export default function AboutPage() {
  return (
    <div>
      <h1>About Us</h1>
      <p>This is the about page.</p>
    </div>
  );
}
```

app/layout.js

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <header>
          <nav>
            <ul>
              <li><a href="/">Home</a></li>
              <li><a href="/about">About</a></li>
            </ul>
          </nav>
        </header>
        <main>{children}</main>
      </body>
    </html>
  );
}
```

HTML 기초

HTML 문서의 기본 구조

HTML 문서는 웹 페이지의 구조와 콘텐츠를 정의하는 마크업 언어입니다. HTML 문서는 기본적으로 다음과 같은 구조를 가집니다.

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <!-- 콘텐츠가 들어가는 부분 -->
</body>
</html>
```

- **DOCTYPE**: 문서 유형을 선언합니다. HTML5에서는 `<!DOCTYPE html>`을 사용합니다.
- **html 태그**: HTML 문서의 루트 요소입니다. `lang` 속성으로 문서의 언어를 지정할 수 있습니다.
- **head 태그**: 메타데이터, 스타일, 제목 등을 포함합니다.
  - **meta 태그**: 문서의 메타데이터를 정의합니다. 예: `<meta charset="UTF-8">`
  - **title 태그**: 문서의 제목을 정의합니다. 브라우저의 제목 표시줄이나 탭에 표시됩니다.
- **body 태그**: 문서의 콘텐츠를 포함합니다. 사용자가 보게 되는 모든 요소가 이 안에 들어갑니다.

주요 태그 소개 - 헤딩, 문단, 목록

헤딩 태그

헤딩 태그는 문서의 제목이나 섹션의 제목을 정의하는 데 사용됩니다. `<h1>`부터 `<h6>`까지 있으며, 숫자가 클수록 작은 헤딩을 나타냅니다.

```
<h1>제목 1</h1>
<h2>제목 2</h2>
<h3>제목 3</h3>
<h4>제목 4</h4>
<h5>제목 5</h5>
<h6>제목 6</h6>
```

문단 태그

문단 태그는 텍스트의 단락을 정의합니다. `<p>` 태그를 사용합니다.

```
<p>이것은 하나의 문단입니다.</p>
<p>이것은 또 다른 문단입니다.</p>
```

목록 태그

목록 태그는 항목을 나열하는 데 사용됩니다. 순서가 있는 목록과 순서가 없는 목록이 있습니다.

- **순서가 없는 목록**: `<ul>` 태그와 `<li>` 태그를 사용합니다.

```
<ul>
    <li>첫 번째 항목</li>
    <li>두 번째 항목</li>
    <li>세 번째 항목</li>
</ul>
```

- **순서가 있는 목록**: `<ol>` 태그와 `<li>` 태그를 사용합니다.

```
<ol>
    <li>첫 번째 항목</li>
    <li>두 번째 항목</li>
    <li>세 번째 항목</li>
</ol>
```

주요 태그 소개 - 하이퍼링크와 이미지

하이퍼링크 태그

하이퍼링크는 다른 문서나 페이지로 연결되는 링크를 만듭니다. `<a>` 태그를 사용하며, `href` 속성으로 링크할 URL을 지정합니다.

```
<a href="https://www.example.com">Example 사이트로 이동</a>
```

이미지 태그

이미지는 `<img>` 태그를 사용하여 삽입합니다. `src` 속성으로 이미지 파일의 경로를, `alt` 속성으로 이미지가 로드되지 않을 때 표시할 대체 텍스트를 지정합니다.

```
<img src="path/to/image.jpg" alt="설명 텍스트">
```

폼과 입력 요소 - 폼 태그와 입력 요소

폼 태그

폼은 사용자 입력을 받기 위해 사용됩니다. `<form>` 태그를 사용하여 정의합니다.

```
<form action="/submit" method="post">
    <!-- 입력 요소가 여기에 들어갑니다 -->
</form>
```

입력 요소

입력 요소는 사용자로부터 데이터를 입력받기 위해 사용됩니다. 주요 입력 요소로는 `<input>`, `<textarea>`, `<select>`가 있습니다.

- **input 태그**: 다양한 유형의 입력 요소를 생성합니다. `type` 속성으로 입력 유형을 지정합니다.

```
<input type="text" name="username" placeholder="사용자 이름">
<input type="password" name="password" placeholder="비밀번호">
<input type="email" name="email" placeholder="이메일">
<input type="submit" value="제출">
```

- **textarea 태그**: 여러 줄의 텍스트 입력을 받기 위해 사용됩니다.

```
<textarea name="message" rows="4" cols="50" placeholder="메시지를 입력하세요">
</textarea>
```

- **select 태그**: 드롭다운 목록을 생성합니다.

```
<select name="options">
    <option value="option1">옵션 1</option>
    <option value="option2">옵션 2</option>
    <option value="option3">옵션 3</option>
</select>
```

폼 제출과 처리 - 폼의 기본 동작 이해

폼의 method와 action 속성

폼의 `method` 속성은 데이터를 전송하는 방식을 지정하며, `action` 속성은 데이터를 전송할 URL을 지정합니다.

- **method 속성**: 주로 `GET`과 `POST`를 사용합니다.
  - `GET`: URL에 데이터를 포함하여 전송합니다. 데이터가 URL에 노출됩니다.
  - `POST`: 데이터를 요청 본문에 포함하여 전송합니다. 데이터가 URL에 노출되지 않습니다.

```
<form action="/submit" method="post">
    <input type="text" name="username" placeholder="사용자 이름">
    <input type="submit" value="제출">
</form>
```

폼 제출과 처리 - 폼 데이터 처리

서버로 데이터 전송

폼을 제출하면 데이터가 서버로 전송됩니다. 서버는 전송된 데이터를 처리하고 응답을 반환합니다.

GET과 POST 차이

- **GET**: URL에 데이터를 포함하여 전송합니다. 주로 데이터를 조회하는 요청에 사용됩니다.

```
<form action="/search" method="get">
    <input type="text" name="query" placeholder="검색어">
    <input type="submit" value="검색">
</form>
```

- **POST**: 데이터를 요청 본문에 포함하여 전송합니다. 주로 데이터를 생성하거나 업데이트하는 요청에 사용됩니다.

```
<form action="/submit" method="post">
    <input type="text" name="username" placeholder="사용자 이름">
    <input type="password" name="password" placeholder="비밀번호">
    <input type="submit" value="제출">
</form>
```

CSS 기초

CSS의 역할과 필요성

CSS(Cascading Style Sheets)는 HTML 요소의 스타일을 정의하고 제어하는 데 사용됩니다. CSS를 사용하면 웹 페이지의 시각적 표현을 분리하여 HTML 문서의 구조와 콘텐츠를 보다 효율적으로 관리할 수 있습니다.

- **스타일 시트의 개념**: CSS는 여러 HTML 요소에 스타일을 적용할 수 있는 스타일 시트를 통해 웹 페이지의 외형을 제어합니다.
- **CSS의 필요성**: HTML만으로는 웹 페이지의 구조만 정의할 수 있으며, CSS를 사용하여 디자인, 레이아웃, 색상, 폰트 등 시각적 요소를 제어할 수 있습니다.

CSS 문법과 기본 선택자

CSS는 선택자와 속성-값 쌍으로 구성됩니다. 선택자는 스타일을 적용할 HTML 요소를 지정하고, 속성은 스타일의 종류, 값은 스타일의 구체적인 설정을 나타냅니다.

```
선택자 {
    속성: 값;
```

```
}
```

기본 선택자

- **태그 선택자**: 특정 태그에 스타일을 적용합니다.

```
p {
    color: blue;
}
```

- **클래스 선택자**: 클래스 이름으로 요소를 선택합니다. 클래스는 여러 요소에 적용할 수 있습니다.

```
.my-class {
    font-size: 20px;
}
```

- **아이디 선택자**: 아이디 이름으로 요소를 선택합니다. 아이디는 문서 내에서 고유해야 합니다.

```
#my-id {
    background-color: yellow;
}
```

- **속성 선택자**: 특정 속성을 가진 요소를 선택합니다.

```
input[type="text"] {
    border: 1px solid black;
}
```

레이아웃 구성 - 박스 모델

박스 모델은 HTML 요소가 박스로 취급되는 개념으로, 각 요소는 콘텐츠, 패딩, 테두리, 마진으로 구성됩니다.

```
div {
    width: 300px;
    padding: 20px;
    border: 10px solid black;
    margin: 30px;
}
```

- **콘텐츠**: 요소의 실제 내용.

- **패딩**: 콘텐츠 주위의 여백.
- **테두리**: 패딩 주위의 선.
- **마진**: 테두리 바깥의 여백.

레이아웃 구성 - Flexbox 레이아웃

Flexbox는 1차원 레이아웃 모델로, 요소를 행 또는 열 방향으로 정렬할 때 사용됩니다.

Flex 컨테이너

Flex 컨테이너는 `display: flex;`를 사용하여 설정됩니다.

```
.container {
    display: flex;
}
```

Flex 아이템

Flex 컨테이너 내부의 모든 직접 자식 요소는 Flex 아이템이 됩니다.

정렬 방법

- **주축 정렬**: `justify-content`

  ```
  .container {
      justify-content: center; /* flex-start, flex-end, space-between,
  space-around, space-evenly */
  }
  ```

- **교차축 정렬**: `align-items`

  ```
  .container {
      align-items: center; /* flex-start, flex-end, stretch, baseline
  */
  }
  ```

- **여러 줄 정렬**: `align-content`

  ```
  .container {
      align-content: space-between; /* flex-start, flex-end, center,
  ```

```
space-around, stretch */
}
```

레이아웃 구성 - Grid 레이아웃

Grid 레이아웃은 2차원 레이아웃 시스템으로, 행과 열을 사용하여 요소를 배치합니다.

그리드 컨테이너

Grid 컨테이너는 `display: grid;`를 사용하여 설정됩니다.

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows: repeat(2, 100px);
}
```

그리드 아이템

Grid 컨테이너 내부의 모든 직접 자식 요소는 Grid 아이템이 됩니다.

그리드 영역

- **그리드 행과 열 정의**: `grid-template-columns`와 `grid-template-rows`

```
.container {
    grid-template-columns: 200px 1fr 1fr;
    grid-template-rows: 100px 200px;
}
```

- **아이템 배치**: `grid-column`과 `grid-row`

```
.item1 {
    grid-column: 1 / 3; /* 첫 번째에서 세 번째 열까지 */
    grid-row: 1 / 2; /* 첫 번째 행 */
}
```

스타일 적용 - 색상, 글꼴, 배경

색상

- **색상 이름**: `color: red;`
- **HEX 값**: `color: #ff0000;`
- **RGB 값**: `color: rgb(255, 0, 0);`
- **RGBA 값**: `color: rgba(255, 0, 0, 0.5);`

```
p {
    color: blue;
    background-color: #f0f0f0;
}
```

글꼴

- **폰트 설정**: `font-family`, `font-size`, `font-weight`, `font-style`

```
p {
    font-family: 'Arial', sans-serif;
    font-size: 16px;
    font-weight: bold;
    font-style: italic;
}
```

배경

- **배경 이미지**: `background-image`, `background-size`, `background-position`

```
.background {
    background-image: url('image.jpg');
    background-size: cover;
    background-position: center;
}
```

추가적으로 학습할 내용

반응형 디자인

- **미디어 쿼리**: 다양한 화면 크기와 장치에 대응하기 위해 사용됩니다.

```
@media (max-width: 600px) {
    .container {
        flex-direction: column;
    }
}
```

애니메이션과 전환

- **전환**: `transition` 속성을 사용하여 요소의 상태 변화에 애니메이션 효과를 추가합니다.

```
.button {
    background-color: blue;
    transition: background-color 0.3s ease;
}
.button:hover {
    background-color: green;
}
```

- **애니메이션**: `@keyframes`와 `animation` 속성을 사용하여 복잡한 애니메이션을 정의합니다.

```
@keyframes slidein {
    from {
        transform: translateX(-100%);
    }
    to {
        transform: translateX(0);
    }
}
.slide {
    animation: slidein 1s forwards;
}
```

변환

- **변환**: `transform` 속성을 사용하여 요소를 이동, 회전, 크기 조절, 기울이기 등의 변환을 수행합니다.

```
.rotate {
    transform: rotate(45deg);
}
```

가상 클래스와 요소

- **가상 클래스**: `:hover`, `:focus`, `:active` 등

```
a:hover {
    color: red;
}
```

- **가상 요소**: `::before`, `::after`

```
.content::before {
    content: 'Start: ';
}
```

CSS 예제

기본 HTML과 CSS 구조를 하나 두고, 각 챕터별로 기능을 하나씩 추가해 나가면서 설명하겠습니다.

기본 HTML 구조

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS 기초 예제</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>CSS 기초 예제</h1>
        <p class="example">이 예제는 CSS 기초를 설명하기 위해 작성되었습니다.</p>
        <div class="box">박스 모델</div>
        <div class="flex-container">
            <div class="flex-item">Flex Item 1</div>
            <div class="flex-item">Flex Item 2</div>
            <div class="flex-item">Flex Item 3</div>
        </div>
        <div class="grid-container">
            <div class="grid-item">Grid Item 1</div>
            <div class="grid-item">Grid Item 2</div>
            <div class="grid-item">Grid Item 3</div>
        </div>
        <button class="button">전환 버튼</button>
```

```
        <div class="content">가상 요소 예제</div>
    </div>
</body>
</html>
```

CSS 기초

CSS의 역할과 필요성

```
/* styles.css */
body {
    font-family: Arial, sans-serif;
    background-color: #f0f0f0;
    color: #333;
}

.container {
    max-width: 800px;
    margin: 0 auto;
    padding: 20px;
    background-color: white;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
    text-align: center;
}
```

CSS 문법과 기본 선택자

```
/* 기본 선택자 예제 */
p {
    color: blue;
}

.my-class {
    font-size: 20px;
}

#my-id {
    background-color: yellow;
}

input[type="text"] {
```

```
    border: 1px solid black;
}
```

레이아웃 구성 - 박스 모델

```
/* 박스 모델 예제 */
.box {
    width: 300px;
    padding: 20px;
    border: 10px solid black;
    margin: 30px auto;
    text-align: center;
}
```

레이아웃 구성 - Flexbox 레이아웃

```
/* Flexbox 레이아웃 예제 */
.flex-container {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin: 20px 0;
}

.flex-item {
    background-color: lightblue;
    padding: 20px;
    margin: 10px;
}
```

레이아웃 구성 - Grid 레이아웃

```
/* Grid 레이아웃 예제 */
.grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 10px;
}

.grid-item {
    background-color: lightgreen;
    padding: 20px;
```

```
    text-align: center;
}
```

스타일 적용 - 색상, 글꼴, 배경

```
/* 색상, 글꼴, 배경 예제 */
p {
    color: blue;
    background-color: #f0f0f0;
    font-family: 'Arial', sans-serif;
    font-size: 16px;
    font-weight: bold;
    font-style: italic;
}

.background {
    background-image: url('image.jpg');
    background-size: cover;
    background-position: center;
}
```

추가적으로 학습할 내용

```
/* 반응형 디자인 예제 */
@media (max-width: 600px) {
    .flex-container {
        flex-direction: column;
    }
}

/* 전환 예제 */
.button {
    background-color: blue;
    color: white;
    padding: 10px 20px;
    border: none;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

.button:hover {
    background-color: green;
}

/* 애니메이션 예제 */
```

```css
@keyframes slidein {
    from {
        transform: translateX(-100%);
    }
    to {
        transform: translateX(0);
    }
}

.slide {
    animation: slidein 1s forwards;
}

/* 변환 예제 */
.rotate {
    transform: rotate(45deg);
}

/* 가상 클래스와 요소 예제 */
a:hover {
    color: red;
}

.content::before {
    content: 'Start: ';
}
```

React.js 기초

React 개요 및 기본 개념

- **React란 무엇인가?**
  - React는 Facebook에서 개발하고 유지하는 자바스크립트 라이브러리로, 사용자 인터페이스를 구축하기 위해 사용됩니다.
  - 선언적, 컴포넌트 기반, 배우기 쉬운 라이브러리로, 효율적이고 유연한 UI를 만들 수 있습니다.
- **컴포넌트 기반 아키텍처**
  - React 애플리케이션은 여러 개의 독립적이고 재사용 가능한 컴포넌트로 구성됩니다.
  - 각 컴포넌트는 특정 부분의 UI를 정의하며, 서로 결합하여 전체 애플리케이션을 구성합니다.

컴포넌트 기반 구조 이해

- **컴포넌트의 정의**
  - 컴포넌트는 UI를 구성하는 기본 단위입니다.
  - 컴포넌트는 HTML과 JavaScript를 결합하여 UI를 구성하는 기능을 제공합니다.

- **클래스형 컴포넌트**
  - ES6 클래스를 사용하여 정의됩니다.
  - `render` 메서드를 사용하여 UI를 반환합니다.
  - `state`와 `lifecycle` 메서드를 사용할 수 있습니다.

```
import React, { Component } from 'react';

class MyComponent extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}

export default MyComponent;
```

- **함수형 컴포넌트**
  - 함수형 컴포넌트는 함수로 정의됩니다.
  - React Hooks를 사용하여 상태와 생명주기 메서드를 사용할 수 있습니다.

```
import React from 'react';

function MyComponent(props) {
  return <h1>Hello, {props.name}</h1>;
}

export default MyComponent;
```

함수형 컴포넌트 작성

- **함수형 컴포넌트 예제**

```
import React from 'react';

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

export default Welcome;
```

- **props 사용법**
  - `props`는 부모 컴포넌트가 자식 컴포넌트에 데이터를 전달하는 방법입니다.
  - 읽기 전용이며, 컴포넌트는 props를 변경할 수 없습니다.

```jsx
import React from 'react';

function Greeting(props) {
  return <h1>Good {props.timeOfDay}, {props.name}!</h1>;
}

export default Greeting;
```

상태(state)와 props

- **state와 props의 차이**
  - `props` : 부모 컴포넌트로부터 전달받은 데이터, 컴포넌트 내에서 수정할 수 없습니다.
  - `state` : 컴포넌트 내에서 관리되는 데이터, 컴포넌트 내에서 수정 가능합니다.
- **상태 관리**
  - 함수형 컴포넌트에서 `useState` 훅을 사용하여 상태를 관리합니다.

```jsx
import React, { useState } from 'react';

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

이벤트 처리 - 이벤트 핸들링

- **이벤트 종류**
  - `onClick` : 클릭 이벤트
  - `onChange` : 값 변경 이벤트
  - `onSubmit` : 폼 제출 이벤트
- **이벤트 핸들러 작성**

```jsx
import React, { useState } from 'react';
```

```
function ClickButton() {
  const handleClick = () => {
    alert('Button clicked!');
  };

  return <button onClick={handleClick}>Click me</button>;
}

export default ClickButton;
```

폼과 사용자 입력 처리

- 폼 요소
  - 사용자 입력을 받기 위한 다양한 폼 요소들을 포함합니다. (예: `input`, `textarea`, `select`)

```
import React, { useState } from 'react';

function FormExample() {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert('Submitted value: ' + inputValue);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Input:
        <input type="text" value={inputValue} onChange={handleChange}
/>
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormExample;
```

- 입력 데이터 처리

- 상태를 사용하여 입력 데이터 변경을 추적하고 처리합니다.

```
import React, { useState } from 'react';

function TextAreaExample() {
  const [text, setText] = useState('');

  const handleChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <textarea value={text} onChange={handleChange} />
      <p>{text}</p>
    </div>
  );
}

export default TextAreaExample;
```

- **상태 업데이트**
  - 입력 데이터가 변경될 때마다 상태를 업데이트하여 최신 값을 유지합니다.

```
import React, { useState } from 'react';

function SelectExample() {
  const [selectedOption, setSelectedOption] = useState('');

  const handleChange = (event) => {
    setSelectedOption(event.target.value);
  };

  return (
    <div>
      <select value={selectedOption} onChange={handleChange}>
        <option value="">Select an option</option>
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
      </select>
      <p>Selected: {selectedOption}</p>
    </div>
  );
}
```

```
export default SelectExample;
```

추가적으로 학습할 내용

컴포넌트 생명주기

- 클래스형 컴포넌트의 생명주기 메서드(`componentDidMount` , `componentDidUpdate` , `componentWillUnmount`)

클래스형 컴포넌트의 생명주기 메서드

클래스형 컴포넌트는 컴포넌트의 생명주기에 따라 특정 메서드를 자동으로 호출합니다. 생명주기 메서드를 통해 컴포넌트가 생성, 업데이트, 제거될 때 특정 작업을 수행할 수 있습니다. 주요 생명주기 메서드에는 `componentDidMount` , `componentDidUpdate` , `componentWillUnmount` 가 있습니다.

componentDidMount

- **설명**: `componentDidMount` 는 컴포넌트가 처음으로 DOM에 마운트된 직후에 호출됩니다. 이 메서드는 주로 네트워크 요청, DOM 요소에 대한 직접적인 조작, 타이머 설정 등 초기화 작업을 수행하는 데 사용됩니다.
- **예제**:

```
import React, { Component } from 'react';

class DataFetcher extends Component {
  state = {
    data: null,
  };

  componentDidMount() {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        this.setState({ data });
      });
  }

  render() {
    const { data } = this.state;

    if (!data) {
      return <p>Loading...</p>;
```

```
    }

    return (
      <div>
        <h1>Fetched Data</h1>
        <pre>{JSON.stringify(data, null, 2)}</pre>
      </div>
    );
  }
}

export default DataFetcher;
```

componentDidUpdate

- **설명**: `componentDidUpdate`는 컴포넌트가 업데이트된 직후에 호출됩니다. 이 메서드는 이전 프롭스와 상태를 인자로 받아 상태 변화에 따른 작업을 수행할 수 있습니다. 주로 특정 조건에서 네트워크 요청을 다시 하거나 DOM을 업데이트할 때 사용됩니다.
- **예제**:

```
import React, { Component } from 'react';

class Counter extends Component {
  state = {
    count: 0,
  };

  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log(`Count updated: ${this.state.count}`);
    }
  }

  increment = () => {
    this.setState(prevState => ({ count: prevState.count + 1 }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

```
export default Counter;
```

componentWillUnmount

- **설명**: `componentWillUnmount`는 컴포넌트가 DOM에서 제거되기 직전에 호출됩니다. 이 메서드는 주로 타이머를 정리하거나, 네트워크 요청을 취소하거나, 이벤트 리스너를 제거하는 등의 정리 작업을 수행하는 데 사용됩니다.
- **예제**:

```
import React, { Component } from 'react';

class Timer extends Component {
  state = {
    seconds: 0,
  };

  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState(prevState => ({ seconds: prevState.seconds + 1
}));
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        <p>Seconds: {this.state.seconds}</p>
      </div>
    );
  }
}

export default Timer;
```

요약

- **componentDidMount**: 컴포넌트가 처음으로 마운트된 직후에 호출. 초기화 작업 수행.
- **componentDidUpdate**: 컴포넌트가 업데이트된 직후에 호출. 상태 변화에 따른 작업 수행.

- **componentWillUnmount**: 컴포넌트가 DOM에서 제거되기 직전에 호출. 정리 작업 수행.

MongoDB와 Next.js 연동

MongoDB와 Next.js를 연동하여 데이터베이스 작업을 수행하는 방법을 설명합니다. Mongoose를 사용하지 않고, Next.js의 기본 기능과 MongoDB 클라이언트 라이브러리를 사용합니다.

MongoDB 개요 - NoSQL 데이터베이스의 개념

NoSQL과 SQL의 차이

- **SQL 데이터베이스**: 관계형 데이터베이스로, 테이블 기반 구조를 사용하며, 데이터가 고정된 스키마에 따라 저장됩니다. 예: MySQL, PostgreSQL.
- **NoSQL 데이터베이스**: 비관계형 데이터베이스로, 다양한 데이터 모델(문서, 키-값, 그래프 등)을 지원하며, 스키마가 유연합니다. 예: MongoDB, Cassandra.

MongoDB의 특징

- **문서 지향 저장**: BSON(Binary JSON) 형식의 문서로 데이터를 저장합니다.
- **유연한 스키마**: 스키마를 미리 정의하지 않아도 되며, 동적으로 데이터 구조를 변경할 수 있습니다.
- **수평적 확장성**: 샤딩을 통해 데이터베이스를 여러 서버에 분산하여 저장할 수 있습니다.
- **강력한 쿼리 언어**: JSON 기반의 쿼리 언어를 사용하여 다양한 방식으로 데이터를 조회할 수 있습니다.

MongoDB 설치 및 설정 - 로컬 MongoDB 설치

MongoDB 설치 방법

1. **MongoDB 다운로드**: MongoDB 다운로드 페이지에서 운영체제에 맞는 MongoDB를 다운로드합니다.
2. **설치**: 다운로드한 설치 파일을 실행하여 MongoDB를 설치합니다.
3. **MongoDB 실행**: 설치가 완료되면 MongoDB 서버를 실행합니다.

```
# MongoDB 서버 실행 (기본적으로 포트 27017 사용)
mongod --dbpath /path/to/your/db
```

기본 설정

- **MongoDB 설정 파일**: MongoDB 설정 파일(`mongod.conf`)을 수정하여 설정을 변경할 수 있습니다.

- **데이터베이스 경로**: `dbpath` 설정을 통해 데이터베이스 파일이 저장될 경로를 지정합니다.
- **포트 번호**: `port` 설정을 통해 MongoDB 서버가 사용하는 포트 번호를 지정합니다.

MongoDB와 Next.js 연동 - 기본 설정

MongoDB 클라이언트 설치

Next.js 프로젝트에서 MongoDB와 연동하기 위해 MongoDB 클라이언트 라이브러리를 설치합니다.

```
npm install mongodb
```

연결 설정

MongoDB 클라이언트를 사용하여 MongoDB 서버에 연결합니다.

```
// lib/mongodb.js
import { MongoClient } from 'mongodb';

const uri = 'mongodb://localhost:27017/mydatabase';
const options = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
};

let client;
let clientPromise;

if (!global._mongoClientPromise) {
  client = new MongoClient(uri, options);
  global._mongoClientPromise = client.connect();
}
clientPromise = global._mongoClientPromise;

export default clientPromise;
```

MongoDB와 Next.js 연동 - CRUD 연산 구현

데이터 생성(Create)

데이터베이스에 새 문서를 삽입합니다.

```
// pages/api/create.js
import clientPromise from '../../lib/mongodb';
```

```
export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const { name, email } = req.body;

  const result = await db.collection('users').insertOne({ name, email });

  res.status(201).json(result);
};
```

데이터 읽기(Read)

데이터베이스에서 문서를 조회합니다.

```
// pages/api/read.js
import clientPromise from '../../lib/mongodb';

export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const users = await db.collection('users').find({}).toArray();

  res.status(200).json(users);
};
```

데이터 업데이트(Update)

데이터베이스에서 기존 문서를 수정합니다.

```
// pages/api/update.js
import clientPromise from '../../lib/mongodb';
import { ObjectId } from 'mongodb';

export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const { id, name, email } = req.body;

  const result = await db.collection('users').updateOne(
    { _id: new ObjectId(id) },
    { $set: { name, email } }
  );
```

```
  res.status(200).json(result);
};
```

데이터 삭제(Delete)

데이터베이스에서 문서를 삭제합니다.

```
// pages/api/delete.js
import clientPromise from '../../lib/mongodb';
import { ObjectId } from 'mongodb';

export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const { id } = req.body;

  const result = await db.collection('users').deleteOne({ _id: new
ObjectId(id) });

  res.status(200).json(result);
};
```

데이터 모델링 - 스키마 정의

스키마 정의와 데이터 타입 설정

MongoDB는 스키마가 유연하여 데이터 타입을 동적으로 변경할 수 있습니다. 스키마를 정의할 필요가 없지만, 필요하다면 클라이언트 쪽에서 데이터 타입을 미리 정의할 수 있습니다.

```
// 데이터 타입 예시
const userSchema = {
  name: String,
  email: String,
};
```

데이터 모델링 - 예제 데이터 삽입 및 조회

샘플 데이터 삽입

샘플 데이터를 데이터베이스에 삽입하는 API를 구현합니다.

```
// pages/api/seed.js
import clientPromise from '../../lib/mongodb';

export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const sampleData = [
    { name: 'Alice', email: 'alice@example.com' },
    { name: 'Bob', email: 'bob@example.com' },
  ];

  const result = await db.collection('users').insertMany(sampleData);

  res.status(201).json(result);
};
```

데이터 조회

데이터베이스에서 데이터를 조회하는 API를 구현합니다.

```
// pages/api/users.js
import clientPromise from '../../lib/mongodb';

export default async (req, res) => {
  const client = await clientPromise;
  const db = client.db();

  const users = await db.collection('users').find({}).toArray();

  res.status(200).json(users);
};
```

Next.js 추가기능 개발

이미지 업로드 - Next.js에서의 이미지 업로드 처리

이미지 업로드 기능 구현

이미지 업로드 기능을 구현하기 위해 클라이언트 측에서 파일 입력 요소를 사용하고, 서버 측에서 파일을 처리하는 API 라우트를 생성합니다.

1. **클라이언트 측: 이미지 업로드 폼**

```
// pages/upload.js
import { useState } from 'react';

export default function Upload() {
  const [file, setFile] = useState(null);

  const handleChange = (e) => {
    setFile(e.target.files[0]);
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    const formData = new FormData();
    formData.append('file', file);

    const res = await fetch('/api/upload', {
      method: 'POST',
      body: formData,
    });

    const data = await res.json();
    console.log(data);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="file" onChange={handleChange} />
      <button type="submit">Upload</button>
    </form>
  );
}
```

2. **서버 측: 파일 업로드 처리 API**

`formidable` 라이브러리를 사용하여 파일 업로드를 처리합니다.

```
npm install formidable
```

```
// pages/api/upload.js
import formidable from 'formidable';
import fs from 'fs';
import path from 'path';

export const config = {
  api: {
    bodyParser: false,
  },
```

```
};

export default function handler(req, res) {
  const form = new formidable.IncomingForm();
  form.uploadDir = './public/uploads';
  form.keepExtensions = true;

  form.parse(req, (err, fields, files) => {
    if (err) {
      return res.status(500).json({ error: err.message });
    }

    const file = files.file;
    const newFilePath = path.join(form.uploadDir, file.name);
    fs.renameSync(file.path, newFilePath);

    res.status(200).json({ filePath: `/uploads/${file.name}` });
  });
}
```

3. **폴더 생성**

이미지 업로드 폴더를 생성합니다.

```
mkdir -p public/uploads
```

이미지 다운로드 및 표시 - 업로드된 이미지의 관리

1. **이미지 저장**

위의 서버 측 코드에서 이미지를 `public/uploads` 폴더에 저장합니다.

2. **다운로드 URL 생성**

업로드된 이미지에 접근할 수 있는 URL을 생성하여 클라이언트에 반환합니다.

```
// 업로드된 파일의 경로 반환
res.status(200).json({ filePath: `/uploads/${file.name}` });
```

3. **이미지 표시**

이미지를 표시하기 위해 URL을 사용합니다.

```
// pages/show.js
import { useState, useEffect } from 'react';
```

```
export default function Show() {
  const [images, setImages] = useState([]);

  useEffect(() => {
    async function fetchImages() {
      const res = await fetch('/api/images');
      const data = await res.json();
      setImages(data);
    }
    fetchImages();
  }, []);

  return (
    <div>
      {images.map((image, index) => (
        <div key={index}>
          <img src={image.url} alt={`Image ${index}`} width={200} />
        </div>
      ))}
    </div>
  );
}
```

4. **이미지 목록 API**

이미지 목록을 반환하는 API를 작성합니다.

```
// pages/api/images.js
import fs from 'fs';
import path from 'path';

export default function handler(req, res) {
  const directoryPath = path.join(process.cwd(), 'public/uploads');
  fs.readdir(directoryPath, (err, files) => {
    if (err) {
      return res.status(500).json({ error: err.message });
    }
    const images = files.map((file) => ({
      url: `/uploads/${file}`,
    }));
    res.status(200).json(images);
  });
}
```

LightHouse를 사용한 성능분석

Lighthouse는 웹 애플리케이션의 성능, 접근성, SEO 등을 분석할 수 있는 오픈 소스 도구입니다. Chrome DevTools에서 사용할 수 있으며, 웹 애플리케이션을 분석하고 개선점을 제안합니다.

1. **Chrome DevTools에서 Lighthouse 실행**
    1. Chrome 브라우저를 열고, 분석할 웹 페이지를 엽니다.
    2. F12 키를 눌러 DevTools를 엽니다.
    3. DevTools의 상단 메뉴에서 "Lighthouse" 탭을 클릭합니다.
    4. 분석할 항목(Performance, Accessibility, Best Practices, SEO, PWA)을 선택합니다.
    5. "Generate report" 버튼을 클릭하여 Lighthouse 분석을 시작합니다.
2. **Lighthouse 분석 결과**
    - **Performance**: 페이지 로드 시간, 인터랙티브 상태 도달 시간 등 성능 측정 항목.
    - **Accessibility**: 웹 페이지의 접근성 문제와 개선 사항.
    - **Best Practices**: 웹 개발의 모범 사례에 대한 점검.
    - **SEO**: 검색 엔진 최적화 상태.
    - **PWA**: 프로그레시브 웹 애플리케이션의 성능.
3. **Lighthouse CI 설정**

    Lighthouse CI를 사용하여 CI/CD 파이프라인에서 자동으로 성능 분석을 수행할 수 있습니다.

```
npm install -g @lhci/cli
```

```
// lighthouserc.json
{
  "ci": {
    "collect": {
      "url": ["http://localhost:3000"],
      "startServerCommand": "npm run start"
    },
    "assert": {
      "assertions": {
        "categories:performance": ["error", { "minScore": 0.9 }]
      }
    },
    "upload": {
      "target": "temporary-public-storage"
    }
  }
}
```

```
lhci autorun
```

Next.js에서 SSR을 이용한 간단한 사용자 인증 및 권한 관리

1. MongoDB 클라이언트 설정

MongoDB 클라이언트를 설정하여 데이터베이스와 연결합니다.

```
// lib/mongodb.js
import { MongoClient } from 'mongodb';

const uri = 'mongodb://localhost:27017/mydatabase';
const options = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
};

let client;
let clientPromise;

if (!global._mongoClientPromise) {
  client = new MongoClient(uri, options);
  global._mongoClientPromise = client.connect();
}
clientPromise = global._mongoClientPromise;

export default clientPromise;
```

2. 회원가입 API 구현

사용자 정보를 MongoDB에 저장하는 회원가입 API를 구현합니다.

```
// pages/api/register.js
import clientPromise from '../../lib/mongodb';

export default async function handler(req, res) {
  if (req.method !== 'POST') {
    return res.status(405).end();
  }

  const { email, password } = req.body;

  const client = await clientPromise;
  const db = client.db();

  const user = await db.collection('users').findOne({ email });
  if (user) {
    return res.status(400).json({ message: 'User already exists' });
  }
```

```
  await db.collection('users').insertOne({
    email,
    password,
    role: 'user', // 기본 역할은 'user'
  });

  res.status(201).json({ message: 'User registered successfully' });
}
```

3. 로그인 API 구현

사용자 인증을 위한 로그인 API를 구현합니다.

```
// pages/api/login.js
import clientPromise from '../../lib/mongodb';

export default async function handler(req, res) {
  if (req.method !== 'POST') {
    return res.status(405).end();
  }

  const { email, password } = req.body;

  const client = await clientPromise;
  const db = client.db();

  const user = await db.collection('users').findOne({ email, password });
  if (!user) {
    return res.status(401).json({ message: 'Invalid email or password' });
  }

  // 세션 쿠키 설정
  res.setHeader('Set-Cookie', `user=${JSON.stringify(user)}; Path=/;
HttpOnly`);

  res.status(200).json({ message: 'Login successful' });
}
```

4. HOC를 통한 인증 보호 구현

SSR에서 인증 상태를 확인하는 HOC를 구현합니다.

```
// hoc/withAuth.js
import { parseCookies } from 'nookies';
```

```
const withAuth = (WrappedComponent, allowedRoles = []) => {
  const Wrapper = (props) => <WrappedComponent {...props} />;

  Wrapper.getInitialProps = async (ctx) => {
    // req: 서버 측 요청 객체 (Node.js의 http.IncomingMessage 객체)
    // res: 서버 측 응답 객체 (Node.js의 http.ServerResponse 객체)
    // pathname: 현재 페이지의 경로
    // query: URL의 쿼리 매개변수 객체
    // asPath: 브라우저에 표시되는 경로
    // isServer: 서버 측 렌더링인지 클라이언트 측 렌더링인지 여부
    // store: Redux와 같은 상태 관리 라이브러리를 사용할 때의 스토어 객체 (withRedux 사용 시)
    // AppTree: Next.js의 전체 앱 구조를 나타내는 컴포넌트, 서버 측 렌더링 시 HTML을 생성하는 데 사용됩니다.

    const { user } = parseCookies(ctx);

    if (!user) {
      if (ctx.res) {
        ctx.res.writeHead(302, { Location: '/login' });
        ctx.res.end();
      } else {
        document.location.pathname = '/login';
      }
      return {};
    }

    const userData = JSON.parse(user);

    if (allowedRoles.length && !allowedRoles.includes(userData.role)) {
      if (ctx.res) {
        ctx.res.writeHead(302, { Location: '/login' });
        ctx.res.end();
      } else {
        document.location.pathname = '/login';
      }
      return {};
    }

    const componentProps =
      WrappedComponent.getInitialProps &&
      (await WrappedComponent.getInitialProps(ctx));

    return { ...componentProps, user: userData };
  };

  return Wrapper;
};
```

```
export default withAuth;
```

5. 로그인 및 회원가입 페이지 구현

사용자 회원가입 및 로그인 폼을 구현합니다.

```
// pages/register.js
import { useState } from 'react';

export default function Register() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();

    const res = await fetch('/api/register', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ email, password }),
    });

    const data = await res.json();
    console.log(data);
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email</label>
        <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} />
      </div>
      <div>
        <label>Password</label>
        <input type="password" value={password} onChange={(e) =>
setPassword(e.target.value)} />
      </div>
      <button type="submit">Register</button>
    </form>
  );
}
```

```
// pages/login.js
import { useState } from 'react';
import { useRouter } from 'next/router';

export default function Login() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const router = useRouter();

  const handleSubmit = async (e) => {
    e.preventDefault();

    const res = await fetch('/api/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ email, password }),
    });

    const data = await res.json();

    if (data.message === 'Login successful') {
      router.push('/protected');
    } else {
      console.log(data.message);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email</label>
        <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} />
      </div>
      <div>
        <label>Password</label>
        <input type="password" value={password} onChange={(e) =>
setPassword(e.target.value)} />
      </div>
      <button type="submit">Login</button>
    </form>
  );
}
```

6. 보호된 페이지 구현

관리자가 아닌 사용자는 보호된 페이지에 접근하지 못하도록 설정합니다.

```
// pages/protected.js
import withAuth from '../hoc/withAuth';

function ProtectedPage({ user }) {
  return (
    <div>
      <h1>Protected Page</h1>
      <p>Welcome, {user.email}. This page is only accessible to admins.
</p>
    </div>
  );
}

export default withAuth(ProtectedPage, ['admin']);
```

7. 관리자 페이지 구현

관리자 페이지를 구현합니다.

```
// pages/admin.js
import withAuth from '../hoc/withAuth';

function AdminPage({ user }) {
  return (
    <div>
      <h1>Admin Page</h1>
      <p>Welcome, {user.email}. You have admin access.</p>
    </div>
  );
}

export default withAuth(AdminPage, ['admin']);
```