REST API는 클라이언트와 서버 간의 통신을 표준화된 방식으로 처리하는 소프트웨어 아키텍처 스타일입니다. REST API는 웹 애플리케이션이 서버와 통신하여 데이터 또는 자원의 상태를 조회하거나 수정하는데 사용됩니다.

장점 (Positive Effects)

쉬운 사용 (Easy to Use)

- REST API의 가장 큰 장점은 사용의 용이성입니다. REST API 메시지를 읽는 것만으로도 메시지가 의도하는 바를 쉽게 이해할 수 있습니다. 즉, 메뉴얼을 일일이 읽지 않아도 된다는 점에서 엄청난 가독성을 제공합니다.
- HTTP 인프라를 그대로 사용하기 때문에 별도의 인프라 구축이 필요하지 않으며, Stateless한 특징 덕분에 클라이언트는 서버에서 진행된 내용에 대한 문맥을 알 필요가 없습니다. 즉, URI와 원하는 메소드만 이해하면 됩니다.

클라이언트와 서버 간의 완전한 분리 (Complete Separation between Client and Server)

- 클라이언트는 REST API를 통해 서버와 정보를 주고받습니다. 서버는 클라이언트의 문맥을 유지할 필요가 없어 서로의 역할이 명확히 분리됩니다.
- 이러한 분리는 플랫폼의 독립성을 확장하여 다양한 플랫폼에서 서비스 개발과 배포를 쉽게 할 수 있도록 합니다. 예를 들어, 리눅스 웹 서버에서 윈도우 플랫폼 기반의 웹 브라우저를 통해 빠르게 서비스를 운영할 수 있습니다.

특정 데이터 타입에 대한 세부 표현 (Detail Expression for Specific Data Type)

- REST API는 헤더에서 URI 처리 메소드를 명시함으로써 실제 데이터를 페이로드에 표현할 수 있는 기능을 제공합니다.
- 이는 JSON, XML 등의 다양한 형식으로 세부 표현이 가능하여 가독성을 높일 수 있습니다.

단점 (Negative Effects)

HTTP 메서드의 제약 (Restriction of HTTP Method)

- REST API는 HTTP 메서드를 사용하여 URI를 표현합니다. 이는 인프라에서 편리하게 사용할 수 있는 장점을 주지만, 메서드 형태가 제한적이라는 문제점을 야기합니다.
- 예를 들어, 메일을 보내는 기능을 구현할 때 단순히 보내는 기능 외에도 여러 세부 기능을 작성해야 하는데, 이러한 여러 기능 구현에 제약이 발생할 수 있습니다.

표준의 부재 (Absence of Standard)

- REST API의 큰 단점 중 하나는 표준이 존재하지 않는다는 점입니다. 이는 관리의 어려움과 좋은 API 디자인 가이드가 없음을 의미합니다.
- REST API는 많은 사람들이 각자 정당화된 약속으로 구성되고 움직이게 됩니다. 표준의 부재는 기대감과 별개로 매우 중요한 요소로, 개발자에게는 혼란을 줄 수 있습니다.

REST API의 구조

자원 (Resources)

- REST API의 중심은 자원입니다. 자원은 웹에서 URI(Uniform Resource Identifier)로 식별되며, 데이터베이스의 엔티티와 일치할 수 있습니다.
- 예를 들어, 사용자, 제품, 주문 등이 자원이 될 수 있습니다. 각 자원은 고유한 URI로 접근할 수 있습니다.
  - 예: `/users`, `/products`, `/orders`

HTTP 메서드 (HTTP Methods)

- REST API는 HTTP 프로토콜을 사용하며, 다양한 HTTP 메서드를 통해 자원에 대한 작업을 수행합니다:
  - **GET**: 자원 조회 (ex: `/users` -> 모든 사용자 정보 조회)
  - **POST**: 자원 생성 (ex: `/users` -> 새 사용자 생성)
  - **PUT**: 자원 전체 수정 (ex: `/users/1` -> ID가 1인 사용자 정보를 전체 수정)
  - **PATCH**: 자원 부분 수정 (ex: `/users/1` -> ID가 1인 사용자 정보를 일부 수정)
  - **DELETE**: 자원 삭제 (ex: `/users/1` -> ID가 1인 사용자 삭제)

요청 및 응답 (Request and Response)

- 클라이언트는 서버에 HTTP 요청을 보내고, 서버는 이에 대한 응답을 반환합니다. 요청 및 응답에는 다음과 같은 요소가 포함됩니다:
  - **요청 헤더 (Request Headers)**: 클라이언트의 정보, 인증 토큰, 요청 형식 등을 포함합니다.
  - **요청 본문 (Request Body)**: POST나 PUT 요청 시 전송하는 데이터가 포함됩니다.
  - **응답 헤더 (Response Headers)**: 서버의 정보, 데이터 형식 등을 포함합니다.
  - **응답 본문 (Response Body)**: 요청에 대한 결과 데이터가 포함됩니다. 일반적으로 JSON 또는 XML 형식으로 제공됩니다.

상태 코드 (Status Codes)

- 서버는 HTTP 응답에 상태 코드를 포함하여 요청의 결과를 클라이언트에게 알려줍니다. 주요 상태 코드는 다음과 같습니다:
  - **200 OK**: 요청이 성공적으로 처리됨

- **201 Created**: 새 자원이 성공적으로 생성됨
- **204 No Content**: 요청이 성공했으나 반환할 데이터가 없음
- **400 Bad Request**: 클라이언트의 요청이 잘못됨
- **401 Unauthorized**: 인증이 필요함
- **404 Not Found**: 요청한 자원이 존재하지 않음
- **500 Internal Server Error**: 서버에서 오류가 발생함

URI 구조

- REST API의 URI는 자원에 대한 경로를 명확하게 표현해야 합니다. 일반적으로 다음과 같은 패턴을 따릅니다:
  - `/resource` -> 전체 자원
  - `/resource/{id}` -> 특정 자원 (예: `/users/1`)
  - `/resource/{id}/subresource` -> 특정 자원의 하위 자원 (예: `/users/1/orders`)

현대의 많은 API들은 REST의 제약 조건을 완전히 지키지 않음

1. 하이퍼미디어(HATEOAS)를 통한 상태 전이

HATEOAS (Hypermedia as the Engine of Application State)는 REST API에서 중요한 원칙 중 하나입니다. 이 원칙에 따르면, 클라이언트는 서버로부터 하이퍼미디어를 제공받아 애플리케이션 상태를 전이해야 합니다. 쉽게 말해, **클라이언트는 서버의 응답에 포함된 링크를 통해 다음에 어떤 작업을 할 수 있을지 알아야** 합니다. 이 방법을 통해 클라이언트는 사전 정의된 URI 패턴이나 규칙에 의존하지 않고, 서버가 제공하는 하이퍼링크를 통해 상태 전이를 관리할 수 있습니다.

HATEOAS의 역할:

- 서버는 클라이언트에게 요청을 통해 전달된 리소스뿐만 아니라, **해당 리소스와 상호작용할 수 있는 링크를 포함**시킵니다. 예를 들어, 사용자의 정보를 조회하는 API가 있을 때, 해당 사용자의 세부 정보를 조회하거나 업데이트할 수 있는 URI 링크가 응답에 포함되어야 합니다.
- 클라이언트는 이러한 링크를 따라가면서 애플리케이션의 상태를 전이하고, 추가적인 요청을 보냅니다.

하지만, 많은 REST API는 이러한 **HATEOAS**의 원칙을 준수하지 않습니다. 일반적으로 API 문서나 사전 정의된 URI 패턴에 따라 클라이언트가 직접 다음 요청을 형성하게 됩니다. 이는 사실 REST의 원래 정신에서 벗어난 부분이며, REST의 본질인 **하이퍼미디어의 동적 탐색 기능**을 상실하게 됩니다.

예시:

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "_links": {
    "self": { "href": "/users/1" },
    "update": { "href": "/users/1/update" },
    "delete": { "href": "/users/1/delete" }
  }
}
```

위와 같이, 응답에 포함된 링크들은 클라이언트가 다음 가능한 행동을 직접 알 수 있도록 도와줍니다. 하지만 오늘날 많은 API는 이러한 방식이 아닌, 단순히 리소스 정보만 제공하고 클라이언트는 URI 패턴을 기억하거나 별도의 문서를 참고해야만 합니다.

2. 자기 설명적 메시지

자기 설명적 메시지(self-descriptive messages)는 REST의 또 다른 중요한 제약 조건입니다. 이 원칙에 따르면, **각 메시지(요청과 응답)는 그것만으로도 완전한 의미를 가지고 있어야** 하며, 클라이언트가 해당 메시지를 해석할 때 추가적인 문서를 참조하지 않고도 모든 정보를 이해할 수 있어야 합니다.

REST에서의 자기 설명적 메시지:

- 요청에는 필요한 모든 정보가 포함되어야 합니다. 즉, **헤더, URI, HTTP 메서드, 페이로드 등을 통해 해당 요청이 무엇을 하려는지 명확하게 알 수 있어야** 합니다.
- 응답 메시지 역시 상태 코드와 함께 응답 본문이 무엇을 나타내는지, 어떤 추가 행동을 할 수 있는지를 명확히 해야 합니다.

그러나 실제로 많은 API는 메시지 안에 충분한 정보를 담지 않거나, 응답이 불완전하게 제공되는 경우가 많습니다. 예를 들어, 상태 코드만으로는 그 이유를 명확하게 알 수 없는 상황들이 있습니다. 이 경우, 클라이언트는 별도의 API 문서를 참고하거나 상태 코드만으로 해석해야 하기 때문에, **메시지가 자기 설명적이지 않은 상태**가 됩니다.

문제점:

- 클라이언트가 서버로부터 받은 응답이 충분히 **자기 설명적이지 않으면** 해당 요청의 의미나 처리 결과를 명확히 알기 어렵습니다. 예를 들어, 400 Bad Request 응답이 왔을 때, 그 이유가 무엇인지에 대한 추가 설명이 없으면 클라이언트는 문제를 파악하기 어렵습니다.

해결 방안:

응답 메시지에 상태 코드는 물론, 명확한 에러 메시지나 다음 가능한 행동을 설명하는 정보를 제공해야 합니다. 예를 들어:

```
{
  "status": 400,
  "error": "Bad Request",
  "message": "Invalid email format",
  "hint": "The email should follow standard format: example@example.com"
}
```

REST의 이상적인 목표는 클라이언트와 서버가 각각 독립적으로 진화할 수 있도록 하는 것입니다. 그러나 많은 API는 이러한 목표를 달성하지 못하고 클라이언트와 서버가 강하게 결합된 상태에서 함께 업데이트되어야만 정상 작동하는 상황이 발생하고 있습니다.

References

- https://youtu.be/lsMQRaeKNDk?si=tnc-fkMlWUXAiI7I
- https://youtu.be/RP_f5dMoHFc?si=Vz3dc4SVsW8CpREg
- https://wallees.wordpress.com/2018/04/19/rest-api-%EC%9E%A5%EB%8B%A8%EC%A0%90/