REST(Representational State Transfer)는 네트워크 기반 소프트웨어 아키텍처, 특히 분산된 하이퍼미디어 시스템(예: 웹)의 설계를 안내하는 아키텍처 스타일입니다.

REST는 2000년 Roy Fielding의 박사 학위 논문에서 제안되었으며, 웹 상호작용의 **확장성**(Scalability), **성능**(Performance), **단순성**(Simplicity), **변경 용이성**(Modifiability)을 개선하기 위한 목적으로 만들어졌습니다. REST의 주요 목표는 이 아키텍처 스타일을 사용하여 구축된 애플리케이션이 인터넷과 같은 대규모 네트워크에서 효과적으로 확장될 수 있도록 보장하는 것입니다.

초기 WWW 아키텍처 설계의 문제 및 개선점

HTTP의 성능 문제

- 초기 HTTP는 각 요청에 대해 단일 응답을 제공하는 구조로 설계되었습니다. 그러나 웹 페이지에 여러 이미지나 동적 콘텐츠가 포함될 경우 성능 저하가 발생했습니다.
- **개선점**: HTTP/1.1에서는 지속적인 연결을 유지하고, 요청-응답 간에 캐시를 사용하여 리소스 전송 효율을 높였습니다.

확장성의 한계

- 초기 아키텍처는 사용자 증가와 상업적 사용에 맞춰 확장하기 어려웠습니다.
- **개선점**: 캐시 및 프록시 사용을 통해 네트워크 부하를 분산하고, HTTP 프로토콜을 확장하여 확장성 문제를 해결했습니다.

캐싱과 중간자 문제

- 중간자(프록시, 캐시)의 효과적인 사용이 어려웠고, 표준화된 확장 방식이 부족했습니다. 결과적으로 오래된 캐시 데이터의 신뢰성이 문제가 되었습니다.
- **개선점**: 캐시 가능한 응답을 명시하는 HTTP 헤더 필드 도입으로 문제를 개선했습니다.

분산 하이퍼미디어 시스템의 확장성 문제

- 분산 하이퍼미디어 시스템에서 규모의 확장이 어려웠으며, 네트워크 환경의 다양성으로 인해 상호 호환성이 부족했습니다.
- **개선점**: REST 아키텍처는 리소스 간의 일관된 인터페이스를 유지하며, 독립적인 배포가 가능하도록 제약을 추가해 문제를 해결했습니다.

아키텍처 요소

데이터 요소

- **리소스(Resource)**:
  - REST 아키텍처에서 리소스는 서버가 관리하는 모든 정보를 의미합니다. 이는 웹 페이지, 문서, 이미지, 동영상뿐만 아니라 데이터베이스의 특정 레코드, 사용자 정보, 그리고 시스템의 현재 상태 등 다양한 형태를 포함할 수 있습니다.
  - 리소스는 추상적이며 고유의 식별자를 가지고, 리소스의 특정 상태를 표현으로 전달합니다.
- **리소스 식별자(Resource Identifier)**:
  - 각 리소스는 고유한 URI(Uniform Resource Identifier)를 통해 식별됩니다. URI는 웹 상에서 리소스를 참조하는 중요한 요소로, REST의 기본적인 제약 사항 중 하나는 모든 리소스는 그 URI로 접근 가능해야 한다는 것입니다.
  - 예시: `https://example.com/resource/123`는 특정 리소스를 가리키는 URI입니다.
- **표현(Representation)**:
  - 클라이언트와 서버 간에 교환되는 데이터는 리소스 자체가 아니라 리소스의 **표현**입니다. 표현은 리소스의 특정 상태를 나타내며, 주로 JSON, XML, HTML과 같은 형식으로 제공됩니다.
  - 리소스와 표현을 분리하는 이 구조는 클라이언트와 서버 간의 확장성을 높여, 서버는 동일한 리소스를 다양한 표현 형식으로 제공할 수 있습니다.

커넥터(Connectors)

REST 아키텍처에서는 **커넥터**가 클라이언트와 서버 간의 상호작용을 담당하는 중요한 역할을 합니다. 커넥터는 서버와 클라이언트 사이의 통신을 중재하며, 다양한 유형의 커넥터가 존재합니다.

- **클라이언트(Client)**:
  - 클라이언트는 리소스의 표현을 요청하는 역할을 합니다. 웹 브라우저나 API 소비자 등이 클라이언트의 예입니다. 클라이언트는 서버로 요청을 보내고, 서버로부터 받은 응답을 처리합니다.
- **서버(Server)**:
  - 서버는 클라이언트로부터 받은 요청에 대해 리소스를 제공하는 역할을 합니다. 서버는 요청된 리소스의 표현을 찾아 클라이언트에 응답합니다.
- **프록시(Proxy)**:
  - 프록시는 클라이언트와 서버 사이에 위치하여 요청과 응답을 중계하거나 캐싱하는 역할을 합니다. 이를 통해 성능을 개선하거나 보안을 강화할 수 있습니다.
- **게이트웨이(Gateway)**:
  - 게이트웨이는 서버 또는 클라이언트와 중계 역할을 하는 컴포넌트로, 주로 다른 프로토콜로 변환하는 역할을 수행합니다. 예를 들어, SOAP 요청을 RESTful 방식으로 변환할 수 있습니다.
- **캐시(Cache)**:
  - 캐시는 클라이언트와 서버 사이에 위치하여 자주 요청되는 리소스의 표현을 저장하고 관리하는 역할을 합니다. 이를 통해 성능을 향상시키고 서버의 부하를 줄일 수 있습니다.

REST의 주요 원칙:

5.1.2. 클라이언트-서버 (Client-Server)

- **기본 개념**
  - **클라이언트-서버 아키텍처**는 클라이언트와 서버 간의 책임을 분리하는 것을 의미합니다. 클라이언트는 사용자 인터페이스를 담당하고, 서버는 데이터를 관리하고 처리하는 역할을 합니다. 이로 인해 두 요소는 독립적으로 동작하고 발전할 수 있습니다.
  - 클라이언트는 서버에 요청을 보내고, 서버는 요청에 대한 응답을 보냅니다. 클라이언트는 데이터를 처리하는 기능을 서버로부터 분리하여, 사용자와 직접 상호작용하는 데 집중합니다.
- **장점**
  - **인터페이스와 데이터의 분리**: 클라이언트는 사용자와 상호작용하고, 서버는 데이터를 처리함으로써 두 요소가 독립적으로 발전하고 확장될 수 있습니다.
  - **독립적 발전**: 클라이언트와 서버는 서로의 구현 세부 사항에 의존하지 않아, 각각 독립적으로 변경 및 업그레이드가 가능합니다.
  - **확장성**: 서버는 여러 클라이언트의 요청을 처리할 수 있으며, 클라이언트는 다양한 서버에 연결 가능해 시스템 확장성을 극대화할 수 있습니다.
  - **보안 및 관리 용이성**: 서버가 중앙에서 데이터를 관리하므로 보안성 강화와 데이터 관리를 더욱 쉽게 할 수 있습니다.
- **클라이언트의 역할**
  - 클라이언트는 사용자와 상호작용하며, 서버에 요청을 보내고 응답을 받아 데이터를 처리하는 역할을 합니다. 클라이언트는 서버의 내부 상태에 의존하지 않고 독립적으로 동작합니다.
- **서버의 역할**
  - 서버는 클라이언트의 요청에 따라 리소스를 제공하며, 데이터를 관리하고 클라이언트에 응답하는 역할을 합니다. 서버는 클라이언트의 상태를 저장하지 않고 요청에 따라 독립적으로 동작합니다.

5.1.3. 무상태 (Stateless)

- **무상태성의 개념**
  - 무상태(Stateless)란 서버가 클라이언트의 이전 요청 상태를 기억하지 않는다는 것을 의미합니다. 각 클라이언트 요청은 독립적으로 처리되며, 모든 필요한 정보는 해당 요청에 포함되어야 합니다. 서버는 각 요청을 독립적으로 판단하고 처리할 수 있으며, 이전의 상호작용을 기억할 필요가 없습니다.
  - 클라이언트가 서버에 요청할 때, 요청에는 서버가 작업을 완료하는 데 필요한 모든 정보가 포함되어 있어야 합니다. 이는 클라이언트가 이전에 어떤 작업을 했는지와는 관계없이 매번 새로운 상태로 처리된다는 의미입니다.
- **무상태성의 장점**
  - **확장성 향상**: 서버는 상태를 유지할 필요가 없으므로 여러 클라이언트 요청을 처리할 때 더 많은 자원을 할당할 수 있습니다. 서버는 클라이언트의 상태를 기억할 필요가 없으므로, 각 요청이 별개로 처리되어 서버 확장성에 긍정적인 영향을 미칩니다.

    - **서버 부하 감소**: 서버가 클라이언트의 상태를 저장하지 않기 때문에, 클라이언트 상태를 유지하거나 관리하는 데 드는 리소스를 절약할 수 있습니다. 이로 인해 서버 부하가 감소하고 성능이 향상됩니다.
    - **가시성**: 각 요청은 독립적이고 자체적으로 충분한 정보를 포함하기 때문에, 요청을 추적하고 모니터링하기가 용이합니다. 외부에서 들어오는 각 요청을 분석하는 것이 더 쉬워집니다.
    - **신뢰성**: 클라이언트 요청의 실패가 발생할 경우, 상태 정보가 저장되지 않으므로 요청을 재시도하는 데 더 안전하고 효율적입니다. 서버는 특정 클라이언트와 관련된 상태가 없기 때문에, 시스템 복구나 장애 처리도 용이합니다.
- **무상태성의 단점**
    - **부가적인 정보 전달 필요**: 무상태성을 유지하기 위해 클라이언트는 요청마다 서버가 처리에 필요한 모든 정보를 포함해야 합니다. 이는 네트워크 트래픽이 증가할 수 있으며, 요청당 데이터 양이 많아질 수 있습니다.
    - **복잡한 상호작용 처리의 어려움**: 복잡한 상태 관리가 필요한 애플리케이션에서는 무상태 구조가 적합하지 않을 수 있습니다. 특히, 클라이언트의 상태나 세션을 기억하고 상호작용이 긴밀하게 연결된 서비스에서는 상태를 유지하는 것이 더 유리할 수 있습니다.
- **무상태성과 REST의 관계**
    - REST(Representational State Transfer) 아키텍처에서 무상태성은 매우 중요한 제약 조건 중 하나입니다. RESTful 시스템에서는 모든 요청이 독립적으로 처리되며, 서버는 클라이언트의 상태를 유지하지 않습니다. 이로 인해 RESTful API는 대규모 확장에 유리하고, 클라이언트와 서버 간의 상호작용을 단순화할 수 있습니다.
    - 클라이언트는 서버와의 각 상호작용에서 필요한 데이터를 함께 전송해야 하므로, 요청을 보낼 때마다 필요한 상태 정보를 포함하게 됩니다. 이러한 방식은 클라이언트-서버 간의 결합을 낮추고, 서버는 더 효율적으로 작동할 수 있습니다.

5.1.4. 캐시 (Cache)

- **캐시의 개념**
    - 캐시는 클라이언트와 서버 간의 상호작용에서 중복된 요청을 줄이고 성능을 향상시키기 위해, 응답을 저장하고 재사용하는 기술입니다. 캐시된 응답은 서버에 다시 요청을 보내지 않고도 클라이언트에서 사용할 수 있도록 하여, 네트워크 트래픽과 서버 부하를 줄입니다.
    - 클라이언트는 서버로부터 받은 응답을 캐시에 저장하고, 같은 요청이 다시 발생할 경우 캐시된 응답을 사용함으로써 성능을 개선할 수 있습니다. 캐시는 REST 아키텍처의 중요한 요소로, 클라이언트와 서버 간의 상호작용을 최적화하는 데 도움을 줍니다.
- **캐시의 장점**
    - **성능 향상**: 자주 요청되는 리소스를 캐시하면 서버에 다시 요청하지 않고도 클라이언트가 응답을 재사용할 수 있습니다. 이를 통해 응답 시간이 단축되고, 사용자 경험이 향상됩니다.
    - **서버 부하 감소**: 서버는 클라이언트의 동일한 요청에 대해 반복적으로 응답할 필요가 없어집니다. 캐시를 사용하면 중복된 요청을 줄여 서버의 처리 부하를 낮출 수 있습니다.
    - **네트워크 트래픽 감소**: 캐시된 응답을 사용하면 클라이언트가 서버로 보내는 네트워크 요청을 줄

일 수 있으므로, 네트워크 대역폭 사용량이 줄어듭니다.
    - **비용 절감**: 서버 부하와 네트워크 트래픽을 줄임으로써, 운영 비용을 절감할 수 있습니다. 특히, 대규모 웹 애플리케이션에서 캐시는 필수적인 성능 최적화 도구입니다.
- **캐시 가능한 데이터**
    - 모든 응답이 캐시 가능한 것은 아닙니다. 캐시는 주로 정적 데이터(예: 이미지, 스타일시트, 정적 HTML 페이지)에 적합하며, 서버의 상태를 자주 변경하는 동적 데이터는 캐시하기에 적합하지 않습니다.
    - 캐시 가능한 응답은 HTTP 응답 헤더를 통해 명시적으로 설정됩니다. 이를 통해 클라이언트와 중간 프록시는 응답을 캐시할 수 있는지, 그리고 캐시된 응답이 얼마나 오래 유효한지 알 수 있습니다.
- **HTTP에서의 캐시 제어**
    - HTTP는 캐시를 제어하기 위해 여러 가지 헤더를 제공합니다. 대표적으로 `Cache-Control`, `Expires`, `ETag` 등이 있습니다.
        - **Cache-Control**: 클라이언트나 프록시가 리소스를 캐시할 수 있는지 여부와 캐시의 유효 기간을 설정하는 데 사용됩니다. 예: `Cache-Control: max-age=3600`은 리소스를 1시간 동안 캐시할 수 있음을 의미합니다.
        - **Expires**: 응답이 만료되는 시점을 지정합니다. 이 시점이 지나면 클라이언트는 서버에서 새로운 응답을 요청해야 합니다.
        - **ETag**: 서버에서 생성한 리소스의 고유 식별자로, 클라이언트는 ETag를 사용하여 캐시된 리소스가 서버의 최신 버전인지 확인할 수 있습니다.
- **캐시의 유효성 검사**
    - 캐시는 유효 기간이 지나면 만료되며, 클라이언트는 만료된 캐시 데이터를 사용할 수 없습니다. 그러나 클라이언트는 서버에 조건부 요청을 보내서 캐시된 리소스가 여전히 유효한지 확인할 수 있습니다.
    - **조건부 요청**은 클라이언트가 서버에 새로운 요청을 보내기 전에, 캐시된 리소스가 최신 상태인지 서버에 확인하는 방식입니다. 서버는 클라이언트의 요청에 따라 리소스가 변경되지 않았음을 확인하면, 새로 다운로드하지 않고 기존 캐시를 사용할 수 있도록 응답합니다.
- **캐시와 REST의 관계**
    - REST 아키텍처에서는 캐시를 중요한 제약 조건으로 포함하고 있으며, 응답이 캐시 가능하도록 설정하는 것이 원칙입니다. 클라이언트와 서버 간의 상호작용을 최적화하기 위해, 캐시 가능한 리소스는 가능한 한 캐시를 활용하는 것이 권장됩니다.
    - 캐시를 사용함으로써 RESTful 시스템의 확장성을 높일 수 있으며, 특히 대규모 애플리케이션에서 자주 사용되는 리소스에 대해 빠른 응답을 제공할 수 있습니다.

5.1.5. 균일한 인터페이스 (Uniform Interface)

- **균일한 인터페이스의 개념**
    - **균일한 인터페이스(Uniform Interface)**는 REST 아키텍처 스타일의 가장 중요한 제약 조건 중 하나입니다. 이는 클라이언트와 서버가 일관된 방식으로 상호작용할 수 있도록 인터페이스를 표준

화하는 것을 의미합니다. 인터페이스가 균일하다는 것은 모든 클라이언트가 동일한 방식으로 서버와 소통하며, 동일한 원칙을 기반으로 요청과 응답을 처리한다는 것을 뜻합니다.
    - 균일한 인터페이스를 통해 시스템의 복잡성이 줄어들고, 다양한 클라이언트와 서버가 상호작용할 수 있도록 표준화된 방법을 제공합니다. 클라이언트는 서버의 내부 구현에 상관없이, 일관된 인터페이스를 통해 서버의 리소스와 상호작용할 수 있습니다.
- **균일한 인터페이스의 구성 요소**
  균일한 인터페이스를 구성하는 네 가지 주요 요소가 있습니다.
    - **리소스 식별(Resource Identification)**:
        - 모든 리소스는 URI(Uniform Resource Identifier)를 통해 고유하게 식별됩니다. 이는 클라이언트가 서버에 있는 리소스에 접근할 수 있는 명확한 경로를 제공합니다. URI는 리소스가 무엇인지를 나타내는 고유 식별자로, 클라이언트는 이를 통해 서버의 특정 리소스를 요청할 수 있습니다.
        - 예시: `https://example.com/users/123` 는 특정 사용자 리소스를 식별하는 URI입니다.
    - **리소스 조작(Resource Manipulation) 기반의 표현**:
        - 리소스의 상태는 클라이언트에 제공되는 **표현(Representation)**을 통해 전달됩니다. 클라이언트는 이 표현을 사용하여 리소스를 조작할 수 있으며, 리소스의 상태를 업데이트하거나 변경된 상태를 서버에 전송합니다.
        - 표현은 주로 JSON, XML, HTML 등의 형식으로 이루어지며, 이를 통해 클라이언트는 서버의 리소스와 상호작용하게 됩니다.
    - **자가 설명적 메시지(Self-descriptive Messages)**:
        - 클라이언트와 서버 간의 모든 요청과 응답은 필요한 모든 정보를 포함하는 자가 설명적(self-descriptive)이어야 합니다. 즉, 클라이언트가 서버에 요청을 보낼 때 그 요청만으로도 해당 요청을 처리하기에 충분한 정보가 포함되어 있어야 합니다.
        - 예를 들어, 요청 메시지에는 어떤 동작을 수행해야 하는지, 어떤 리소스를 참조해야 하는지에 대한 정보가 포함됩니다. 이러한 자가 설명적 메시지를 통해 서버는 이전 요청에 대한 상태를 기억하지 않더라도, 각 요청을 독립적으로 처리할 수 있습니다.
    - **하이퍼미디어(Hypermedia)를 통한 애플리케이션 상태 전이**:
        - 클라이언트는 서버로부터 받은 **하이퍼미디어(Hypermedia)**를 통해 다음에 수행할 수 있는 액션(상태 전이)을 결정합니다. 서버는 응답 메시지에 리소스 간의 링크를 포함하여 클라이언트가 어떤 작업을 수행할 수 있는지를 알려줍니다.
        - 이 개념을 HATEOAS(Hypermedia As The Engine Of Application State)라고 하며, 클라이언트는 하이퍼미디어에 제공된 링크를 통해 다른 리소스에 접근하거나 애플리케이션의 상태를 전이할 수 있습니다.
- **균일한 인터페이스의 장점**
    - **상호작용의 단순화**: 균일한 인터페이스를 통해 클라이언트는 서버의 내부 구조에 대해 알 필요 없이, 명확한 규칙을 따라 리소스에 접근하고 상호작용할 수 있습니다. 이로 인해 클라이언트-서버 간 상호작용이 단순해지고 일관성이 유지됩니다.

    - **독립적 진화 가능**: 서버와 클라이언트는 각각 독립적으로 발전할 수 있습니다. 서버는 리소스의 URI와 표현을 통해 클라이언트와 소통하므로, 서버의 내부 구현을 변경하더라도 클라이언트는 이를 알 필요가 없습니다. 반대로, 클라이언트는 서버의 API만 일관성을 유지하면 자유롭게 발전할 수 있습니다.
    - **확장성**: 균일한 인터페이스를 통해 다수의 클라이언트와 서버 간의 상호작용이 일관되게 유지되므로, 시스템 확장 시에도 클라이언트와 서버 간의 상호작용 방식을 쉽게 관리할 수 있습니다.
    - **캐시 가능성**: 자가 설명적 메시지와 균일한 인터페이스 덕분에, 클라이언트와 서버 간의 상호작용은 캐시 가능하여 성능을 더욱 향상시킬 수 있습니다. 클라이언트는 이전에 받은 자가 설명적 응답을 재사용할 수 있습니다.
- **균일한 인터페이스의 단점**
    - **유연성의 제한**: 인터페이스를 일관되게 유지하는 것이 시스템 확장과 상호작용에 도움을 주지만, 특정 애플리케이션에 맞춘 커스텀 동작을 정의하는 데 있어 제한적일 수 있습니다. 모든 리소스와 상호작용 방식이 표준화되어 있으므로, 복잡한 상호작용을 필요로 하는 애플리케이션에서는 유연성이 부족할 수 있습니다.
    - **표현 형식의 제한**: 서버와 클라이언트 간의 리소스 전달을 위한 표현 형식은 특정 표준에 의존하는 경향이 있습니다. 이를 확장하려면 클라이언트와 서버 간의 협력과 표준 준수가 필수적입니다.
- **REST에서 균일한 인터페이스의 역할**
    - REST의 균일한 인터페이스는 웹 아키텍처에서 확장성, 독립성, 일관성을 제공하는 핵심 요소입니다. 클라이언트와 서버가 독립적으로 발전하고 유지보수할 수 있게 하며, 이를 통해 대규모 분산 시스템에서 다수의 구성 요소가 동일한 방식으로 상호작용할 수 있습니다.
    - 이는 RESTful API가 간단하고 확장 가능한 이유 중 하나로, 다양한 클라이언트와 서버가 함께 동작하면서도 인터페이스의 일관성을 유지할 수 있도록 도와줍니다.

5.1.6. 계층형 시스템 (Layered System)

- **계층형 시스템의 개념**
    - **계층형 시스템(Layered System)**은 REST 아키텍처의 제약 조건 중 하나로, 클라이언트와 서버 사이의 상호작용을 여러 계층으로 분리하여 각 계층이 독립적으로 기능할 수 있도록 하는 방식입니다. 이러한 계층화는 클라이언트와 서버 간의 직접적인 상호작용을 제한하고, 중간 계층들이 추가적인 기능을 수행할 수 있게 해줍니다.
    - 클라이언트는 여러 계층을 거쳐 서버에 도달하지만, 각 계층의 존재에 대해 알 필요는 없습니다. 이는 클라이언트가 다른 중간 서버나 프록시를 통해 서버에 접근할 수 있도록 하고, 보안, 로드 밸런싱, 캐시 등의 기능을 중간 계층에서 수행할 수 있게 합니다.
- **계층형 시스템의 특징**
    - **계층 간 독립성**: 각 계층은 독립적으로 동작할 수 있으며, 각 계층은 자신이 의도한 역할에만 집중할 수 있습니다. 예를 들어, 캐싱 계층은 클라이언트의 요청을 캐시하여 서버 부하를 줄이고, 보안 계층은 요청을 필터링하여 불법적인 접근을 차단합니다.
    - **클라이언트의 투명성**: 클라이언트는 자신이 직접 서버와 상호작용하는지, 아니면 중간 계층을 통해 상호작용하는지 알지 못합니다. 이는 클라이언트가 중간 계층의 존재에 의존하지 않고 동일한 방식으로 서버와 상호작용할 수 있게 해줍니다.

    - **보안 및 관리 용이성**: 중간 계층을 통해 보안 관련 기능을 강화할 수 있습니다. 예를 들어, 방화벽을 중간 계층에 두어 서버로 가는 요청을 필터링하고, 보안 정책을 적용할 수 있습니다.
- **계층형 시스템의 장점**
    - **확장성**: 계층을 추가함으로써 시스템은 더 쉽게 확장될 수 있습니다. 예를 들어, 로드 밸런서를 추가하여 여러 서버에 클라이언트 요청을 분산하거나, 프록시 서버를 사용해 대규모 트래픽을 처리할 수 있습니다. 이 계층 구조는 서버나 클라이언트 자체를 변경하지 않고도 확장할 수 있는 유연성을 제공합니다.
    - **보안 강화**: 중간 계층은 보안을 강화하는 데 중요한 역할을 합니다. 방화벽, 인증 서버, 암호화 계층 등을 추가하여 요청이 서버에 도달하기 전에 인증 및 필터링 작업을 수행할 수 있습니다.
    - **관리성 향상**: 계층 간의 분리를 통해 각 계층이 특정 역할만을 수행하게 되어 관리가 용이해집니다. 예를 들어, 캐시 계층은 서버의 상태를 유지할 필요 없이 단순히 캐시된 응답을 제공하는 역할만 수행할 수 있습니다.
    - **성능 최적화**: 캐시 계층을 두어 서버로 가는 요청 수를 줄이거나, 로드 밸런서를 사용해 트래픽을 여러 서버로 분산시킴으로써 성능을 최적화할 수 있습니다. 이러한 방식은 서버의 부하를 줄이고 응답 시간을 단축하는 데 기여합니다.
- **계층형 시스템의 단점**
    - **지연 시간 증가**: 여러 계층을 통과하면서 클라이언트의 요청이 서버에 도달하기까지 시간이 더 걸릴 수 있습니다. 각 계층이 처리하는 시간이 추가되기 때문에, 계층이 많아질수록 지연 시간이 증가할 가능성이 있습니다.
    - **복잡성 증가**: 각 계층이 독립적으로 동작하기 때문에 시스템의 구조가 복잡해질 수 있습니다. 특히, 여러 계층을 관리하고 모니터링하는 데 있어서 추가적인 노력이 필요할 수 있습니다.
    - **계층 간 의존성 문제**: 계층 간의 독립성은 유지되지만, 잘못된 구성이나 상호작용에 의존하게 되면 계층 간의 문제가 발생할 수 있습니다. 따라서 각 계층의 역할을 명확히 정의하고, 이들 간의 의존성을 최소화하는 것이 중요합니다.
- **계층형 시스템과 REST의 관계**
    - 계층형 시스템은 REST 아키텍처에서 확장성, 보안성, 성능 향상 등 여러 측면에서 중요한 역할을 합니다. 클라이언트와 서버 사이에 중간 계층을 둠으로써, 시스템은 더 큰 유연성과 확장성을 확보할 수 있습니다.
    - 예를 들어, 캐시 계층을 추가하여 자주 사용되는 데이터에 대한 요청을 처리함으로써 서버의 부하를 줄일 수 있으며, 보안 계층을 통해 인증된 요청만 서버에 도달하도록 할 수 있습니다. 또한 로드 밸런서를 통해 대규모 트래픽을 여러 서버로 분산시킬 수 있습니다.
    - REST의 계층형 시스템은 대규모 분산 시스템에서도 일관성을 유지하면서도 유연한 구조를 제공하며, 이는 RESTful 웹 서비스의 성능과 안정성을 보장하는 데 중요한 역할을 합니다.

5.1.7. 요청 시 코드 (Code-On-Demand)

- **요청 시 코드의 개념**
    - **요청 시 코드(Code-On-Demand)**는 REST 아키텍처에서 선택적인 제약 조건으로, 클라이언트가 서버로부터 동적으로 실행 가능한 코드를 요청할 수 있는 기능을 제공합니다. 이 기능은 클라이

언트의 기능을 확장하고, 서버에서 필요할 때마다 클라이언트로 특정 로직을 전달하여 동작할 수 있게 합니다.
    - 클라이언트는 서버로부터 필요한 기능을 다운로드받아 실행할 수 있으며, 이를 통해 애플리케이션의 복잡성을 줄이고 유연성을 높일 수 있습니다. 주로 자바스크립트나 플래시 같은 스크립트 언어가 이 방식으로 사용됩니다.
- **요청 시 코드의 특징**
    - **동적 기능 확장**: 클라이언트는 고정된 기능만 사용하는 것이 아니라, 서버로부터 추가적인 기능을 다운로드받아 동적으로 실행할 수 있습니다. 이를 통해 클라이언트는 처음부터 모든 기능을 내장할 필요 없이, 서버의 지시에 따라 필요한 기능만을 동적으로 가져와 사용할 수 있습니다.
    - **유연성 증가**: 서버가 클라이언트에게 제공할 수 있는 코드를 통해 클라이언트는 다양한 동작을 수행할 수 있으며, 이를 통해 애플리케이션의 유연성과 확장성을 높일 수 있습니다. 클라이언트는 서버에서 제공하는 코드를 통해 동적으로 기능을 변화시킬 수 있습니다.
- **요청 시 코드의 장점**
    - **클라이언트의 기능 확장**: 서버는 필요에 따라 클라이언트에 특정 기능을 제공하고, 클라이언트는 이를 실행함으로써 애플리케이션의 기능을 확장할 수 있습니다. 예를 들어, 웹 페이지에서 서버로부터 자바스크립트 코드를 전송받아 실행하면, 클라이언트는 해당 페이지에서 동적 기능을 수행할 수 있습니다.
    - **유연한 유지보수**: 클라이언트가 동적으로 코드를 다운로드하여 실행할 수 있기 때문에, 클라이언트 측 소프트웨어의 업데이트나 유지보수가 더 용이해집니다. 서버에서 새로운 기능을 추가하거나 버그를 수정할 때 클라이언트는 이를 서버로부터 자동으로 제공받아 사용할 수 있습니다.
    - **효율적인 리소스 사용**: 클라이언트는 처음부터 모든 기능을 로드할 필요가 없으며, 서버에서 제공하는 필요한 코드만 동적으로 다운로드받아 실행할 수 있습니다. 이로 인해 클라이언트의 초기 로드 시간이 단축되고, 필요한 기능만을 사용할 수 있어 리소스 사용이 최적화됩니다.
- **요청 시 코드의 단점**
    - **가시성 저하**: 요청 시 코드를 사용할 경우 클라이언트의 동작이 명확하지 않을 수 있습니다. 서버에서 동적으로 제공되는 코드가 클라이언트에서 실행되기 때문에, 클라이언트의 전체적인 동작을 사전에 예측하기 어렵습니다. 이는 애플리케이션 모니터링이나 디버깅을 복잡하게 만들 수 있습니다.
    - **보안 위험**: 서버에서 제공하는 코드가 클라이언트에서 실행되기 때문에, 보안적인 문제가 발생할 수 있습니다. 신뢰할 수 없는 서버에서 제공된 코드가 클라이언트에서 실행될 경우, 클라이언트 시스템에 악영향을 미칠 수 있습니다. 따라서 요청 시 코드의 보안 관리는 매우 중요합니다.
    - **성능 저하**: 서버로부터 코드를 다운로드하여 실행하는 과정에서 지연이 발생할 수 있으며, 특히 네트워크 상태가 불안정하거나 서버의 응답 속도가 느릴 경우 성능 저하가 발생할 수 있습니다. 클라이언트는 코드를 실행하기 위해 서버와의 추가적인 통신을 필요로 하기 때문에, 요청 시 코드 사용은 성능 문제를 유발할 수 있습니다.
- **요청 시 코드의 사용 예시**
    - **자바스크립트**: 웹 브라우저에서 클라이언트가 서버로부터 자바스크립트를 다운로드하여 실행하는 방식이 대표적인 요청 시 코드의 예입니다. 웹 페이지는 기본적인 HTML 구조만을 포함하고, 자바스크립트를 통해 동적 기능을 추가하여 사용자와의 상호작용을 제공합니다.

    - **API 응답에서 스크립트 제공**: 서버가 클라이언트에게 API 응답으로 자바스크립트 같은 스크립트를 포함하여 제공할 수 있습니다. 클라이언트는 이 스크립트를 실행하여 특정 기능을 수행하거나 동적 변화를 처리할 수 있습니다.
- **요청 시 코드와 REST의 관계**
    - 요청 시 코드는 REST 아키텍처의 선택적인 제약 사항으로, 반드시 모든 REST 시스템에서 사용되는 것은 아닙니다. REST의 다른 제약 조건들과 달리 필수적으로 적용되지 않으며, 필요한 경우에만 선택적으로 적용됩니다.
    - REST에서의 요청 시 코드는 클라이언트가 서버로부터 전달받은 코드를 실행하는 방식으로, 클라이언트와 서버 간의 유연성을 높이는 데 기여합니다. 그러나 요청 시 코드가 가시성과 보안 문제를 야기할 수 있기 때문에, 이를 사용할 때는 신중한 고려가 필요합니다.

References

- Architectural Styles and the Design of Network-based Software Architectures