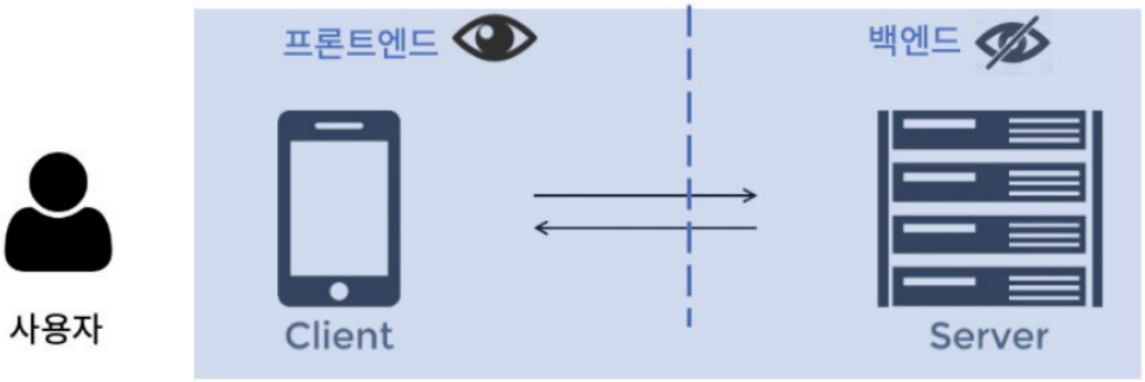
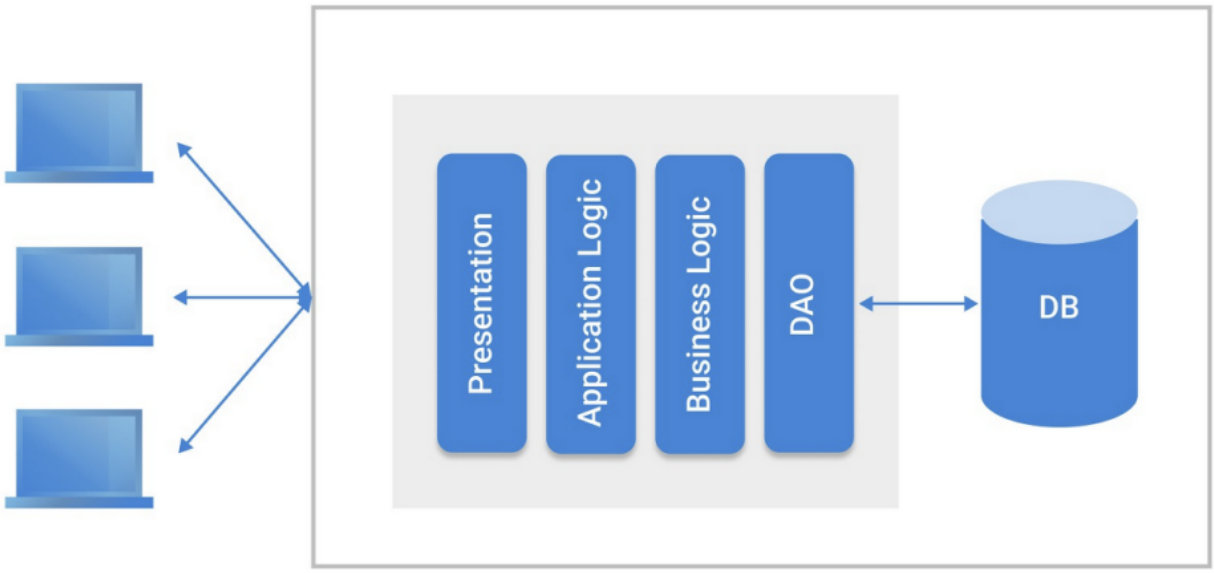
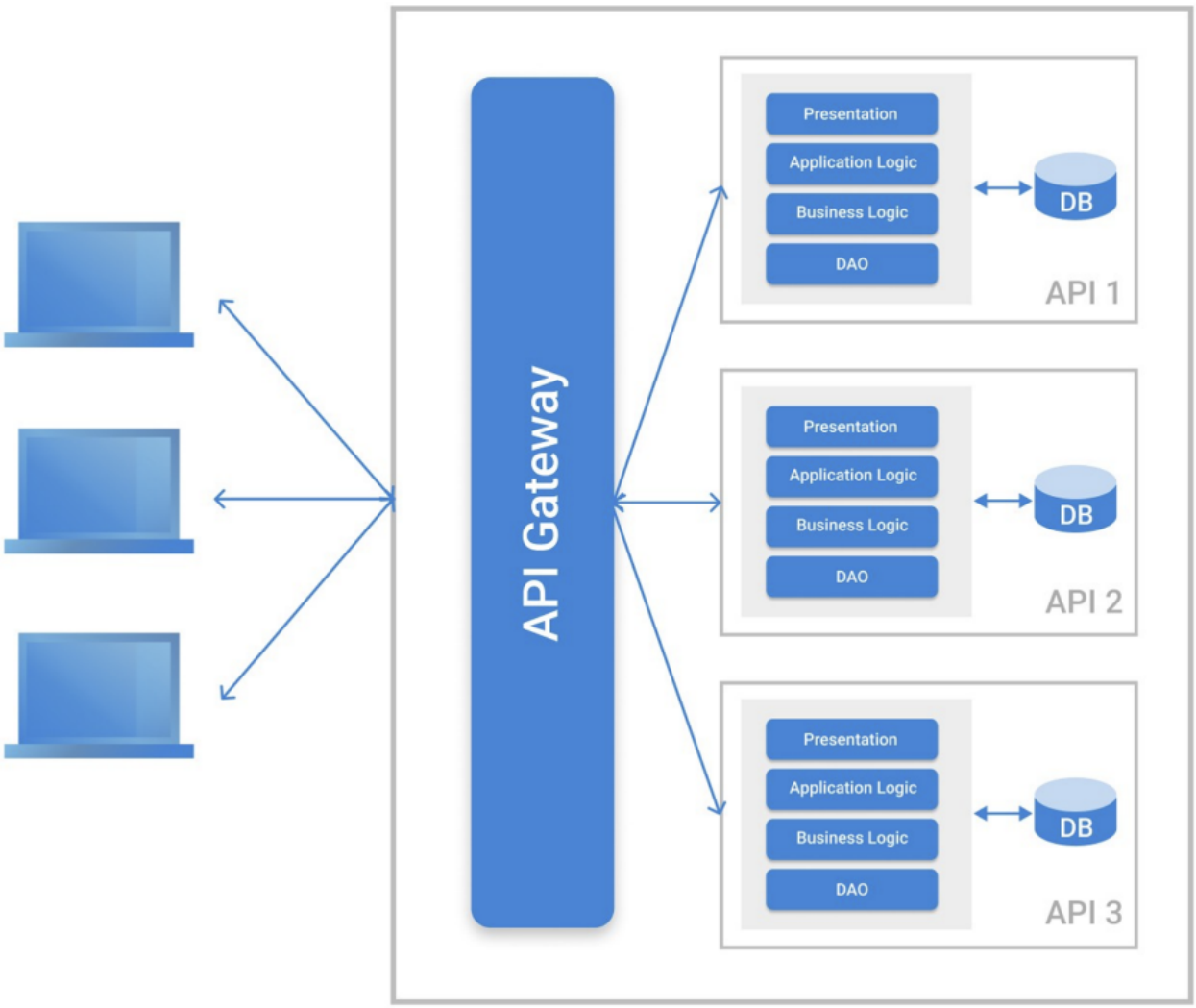
- **응용 프로그램 아키텍처**
    - 설계와 구축 패턴
    - 체계적 구성 지원
    - 로드맵과 모범 사례 제공
- **소프트웨어 설계 패턴**
    - 응용 프로그램 구축 도움
    - 반복 가능한 솔루션 제공
    - 패턴 연결을 통한 일반적인 아키텍처 제작
- **프런트엔드**
    - 사용자 경험(UX)에 초점
    - UI 구현
- **백엔드**
    - 데이터 및 서비스에 대한 액세스 제공
    - 기존 시스템의 동작 지원



- **Client**
    - 서비스나 응용 프로그램을 이용하는 사용자 혹은 시스템
- **Server**
    - 서비스나 응용 프로그램을 제공하는 시스템
- **프로그래밍 언어 선택**
    - 응용 프로그램 유형에 따른 선택 필요
    - 개발 리소스와 요건에 따른 선택 필요
    - 대표적인 언어: Kotlin, Javascript, Ruby, Python, Swift, TypeScript, Java, PHP, SQL
- **모놀리식 아키텍처**
    - 모든 요소가 동일한 리소스와 메모리 공간을 공유하는 전통적인 아키텍처 스타일



- **현대적인 응용 프로그램 아키텍처**
- 마이크로서비스와 응용 프로그래밍 인터페이스를 사용
- 탄력적으로 결합되어 서비스 연결, 클라우드 네이티브 응용 프로그래밍의 기반



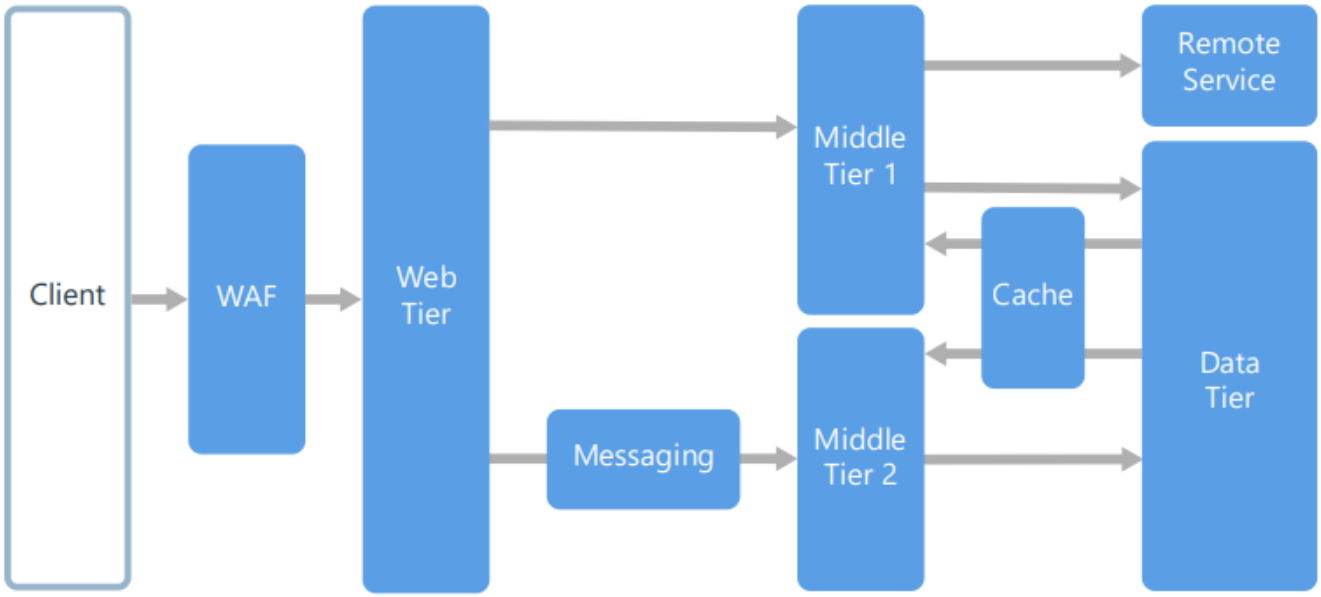
N-Tier Architecture와 모놀리식 아키텍처에 대한 추가적인 설명

N-Tier Architecture

- N-Tier Architecture는 일반적으로 각 계층이 개별 VM 집합에서 실행되는 응용 프로그램으로 구현됩니다.
- 계층을 물리적으로 분리하면, 네트워크 통신이 증가함에 따라 대기 시간이 증가할 수 있습니다.

모놀리식 아키텍처

- 모놀리식 응용 프로그램의 일부를 업데이트하거나 확장하려면 전체 응용 프로그램과 기반 인프라에 영향을 미칩니다.
- 모놀리식 아키텍처는 하나의 변경이 전체 응용 프로그램에 영향을 미치므로 업데이트와 새로운 릴리스는 자주 발생하지 않습니다. 이로 인해 새로운 기능보다는 일반적인 유지 관리가 더 자주 이루어집니다.
- **N-Tier Architecture**
  - 응용 프로그램을 논리적 레이어와 물리적 계층으로 구분
  - 레이어: 책임 구분, 종속성 관리, 특정 책임 부여
  - 계층: 물리적 분리, 별도 시스템에서 실행, 확장성 및 복원성 향상, 추가 네트워크 통신으로 인한 대기시간 증가



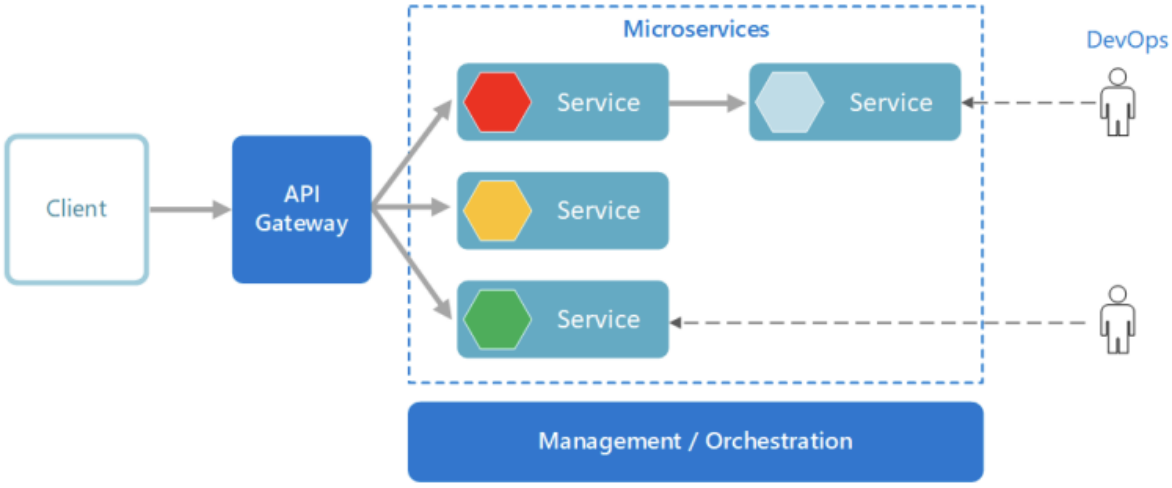
- **3-Tier Application**
  - 구성: Presentation Tier, Middle Tier, Data Tier
  - 복잡한 응용 프로그램: 4개 이상의 계층으로 구성
- **Closed Layer Architecture**
  - 각 레이어가 바로 아래에 있는 다음 레이어만 호출
  - 레이어간의 종속성 제한
- **Open Layer Architecture**
  - 각 레이어가 아래에 있는 모든 레이어 호출 가능
- **N-Tier Architecture 사용 케이스**

- 단순 웹 응용 프로그램
- 온 프레미스 애플리케이션
- 레거시 온 프레미스 응용 프로그램

- **모놀리식 아키텍처**

  - 모든 기능을 단일 응용 프로그램 스택 내에 포함
  - 서비스 상호작용 및 개발/제공 방식이 긴밀하게 연결
  - 한 부분 업데이트 또는 확장 시 전체 응용 프로그램 및 기반 인프라에 영향
  - 일부만 변경하더라도 전체 응용 프로그램을 다시 릴리스해야 함

Microservice Architecture (마이크로서비스 아키텍처)



마이크로서비스 아키텍처는 작은 자율적인 서비스 컬렉션으로 구성됩니다. 각 서비스는 독립적이며 제한된 컨텍스트 내에서 단일 비즈니스 기능을 구현합니다. 제한된 컨텍스트는 비즈니스 내의 자연스러운 분할을 표현하며 도메인 모델이 존재하는 명확한 경계를 제공합니다.

- **특징**

  - 마이크로 서비스는 작고, 독립적이며, 느슨하게 결합됩니다.
  - 하나의 소규모 개발자 팀이 작성하고 유지 관리할 수 있습니다.
  - 각 서비스는 작은 개발 팀이 관리할 수 있는 개별 코드베이스입니다.
  - 서비스를 독립적으로 배포할 수 있습니다. 팀이 전체 애플리케이션을 다시 빌드한 후 재배치하지 않고도 기존 서비스를 업데이트할 수 있습니다.
  - 서비스가 해당 데이터 또는 외부 상태를 유지해야 합니다. 이는 별도의 데이터 레이어가 데이터 지속성을 처리하는 기존 모델과의 차이점입니다.
  - 서비스가 잘 정의된 API를 사용하여 서로 통신합니다. 각 서비스의 내부 구현 세부 정보는 다른 서비스에서 숨겨집니다.
  - 다중 저장소 프로그래밍을 지원합니다. 예를 들어 서비스가 동일한 기술 스택, 라이브러리 또는 프레임워크를 공유할 필요가 없습니다.

- **추가 구성 요소**

  - **관리/오케스트레이션**: 이 구성 요소는 노드에 서비스 배치, 실패 식별, 노드 간에 서비스 부하 조정 등의 작업을 담당합니다. 일반적으로 이 구성 요소는 사용자 지정 빌드가 아니라 Kubernetes와 같은 기성 기술입니다.

    - **API 게이트웨이**: API 게이트웨이는 클라이언트의 진입점입니다. 클라이언트는 서비스를 직접 호출하는 대신, 호출을 백 엔드의 적절한 서비스에 전달하는 API 게이트웨이를 호출합니다. API 게이트웨이를 사용할 경우의 장점은 다음과 같습니다.
        - 클라이언트와 서비스가 분리됩니다. 모든 클라이언트를 업

데이트하지 않고도 서비스 버전을 관리하거나 서비스를 리팩터링할 수 있습니다. - 서비스가 웹 우호적이 아닌 AMQP 등의 메시징 프로토콜을 사용할 수 있습니다. - API 게이트웨이는 인증, 로깅, SSL 종료, 부하 분산 등의 다른 교차 기능을 수행할 수 있습니다. - 제한, 캐싱, 변환 또는 유효성 검사와 같은 즉시 사용 가능한 정책.

마이크로서비스 아키텍처의 이점

1. **민첩성**: 마이크로서비스는 독립적으로 배포되므로 버그 수정 및 기능 릴리스를 더 쉽게 관리할 수 있습니다. 전체 애플리케이션을 다시 배포하지 않고 서비스를 업데이트하거나, 필요하다면 롤백할 수 있습니다.

2. **집중화된 소규모 팀**: 마이크로서비스는 한 기능 팀에서 구축, 테스트 및 배포할 수 있을 만큼 작습니다. 이러한 소규모 팀은 높은 민첩성을 보이며, 대규모 팀에 비해 커뮤니케이션의 속도가 빠르고 관리 오버헤드가 줄어들어 생산성이 향상됩니다.

3. **소규모 코드 기준**: 마이크로서비스 아키텍처는 코드나 데이터 저장소를 공유하지 않으므로 종속성이 최소화되며, 그 결과 새로운 기능을 추가하는 것이 더욱 쉽습니다.

4. **기술의 혼합**: 팀은 혼합된 기술 스택을 적절하게 사용하여 각 서비스에 가장 적합한 기술을 선택할 수 있습니다.

5. **결함 격리**: 개별 마이크로서비스가 사용할 수 없게 되더라도, 잘 설계된 다른 서비스가 문제를 제대로 처리하도록 할 수 있습니다.

6. **확장성**: 각 서비스는 독립적으로 확장될 수 있어, 리소스가 더 많이 필요한 하위 시스템의 규모를 확장하면서 전체 애플리케이션 규모를 확장하지 않아도 됩니다.

7. **데이터 격리**: 단일 마이크로서비스에만 영향을 미치므로, 스키마 업데이트를 수행하는 것이 더욱 쉽습니다. 이는 모놀리식 애플리케이션에서 스키마 업데이트가 훨씬 어려울 수 있는데, 애플리케이션의 다양한 부분이 모두 동일한 데이터에 영향을 미칠 수 있어서 스키마를 변경하는 것이 위험하다는 점에서 차이가 있습니다.

마이크로서비스 아키텍처를 시작하기 전에 고려해야 할 사항들

1. **복잡성**: 마이크로서비스 애플리케이션은 모놀리식 애플리케이션보다 작동 부분이 많을 수 있습니다. 각 서비스는 더 단순하지만 전체 시스템은 더 복잡해질 수 있습니다.

2. **개발 및 테스트**: 다른 서비스에 의존하는 서비스를 개발하려면 통상적인 모놀리식 또는 계층화된 애플리케이션을 작성하는 것과는 다른 접근 방식이 필요합니다. 이로 인해 리팩터링이나 테스트의 어려움이 생길 수 있습니다.

3. **통제 부족**: 마이크로서비스 빌드에 대한 분산 접근 방식은 장점이 있지만, 이로 인해 언어와 프레임워크의 다양성이 증가하면서 애플리케이션 유지 관리가 어려워질 수 있습니다. 로깅과 같은 교차 기능에 대한 프로젝트 전체 표준을 적용하는 것이 유용할 수 있습니다.

4. **네트워크 정체 및 대기 시간**: 여러 개의 작고 세분화된 서비스를 사용하면 서비스 간 통신이 증가하고, 서비스 종속성 체인이 길어지면 추가 대기 시간이 문제가 될 수 있습니다. 이런 문제를 피하기 위해 API 디자인, 통신량 관리, 비동기 통신 패턴 등을 신중하게 고려해야 합니다.

5. **데이터 무결성**: 각 마이크로서비스가 자체 데이터를 관리하므로, 데이터 일관성 유지가 어려울 수 있습니다. 가능한 한 결과적 일관성을 유지하는 것이 중요합니다.

6. **관리**: 마이크로서비스를 성공적으로 운영하려면 DevOps 문화가 필수적입니다. 전체 서비스의 로깅 및 모니터링이 어려울 수 있습니다.

7. **버전 관리**: 서비스를 업데이트할 때 종속된 서비스에 문제가 발생하지 않도록 관리해야 합니다. 이전 버전이나 이후 버전과의 호환성 문제를 방지하기 위해 신중한 설계가 필요합니다.

8. **기술 수준**: 마이크로서비스는 분산 시스템입니다. 팀이 이

마이크로서비스 아키텍처의 장점:

1. **민첩성**: 마이크로서비스는 독립적으로 배포되므로 버그 수정 및 기능 릴리스 관리가 용이합니다. 전체 애플리케이션을 다시 배포할 필요 없이 개별 서비스를 업데이트하고, 문제가 발생하면 롤백할 수 있습니다.

2. **소규모 팀**: 마이크로서비스는 작은 규모의 팀이 각자의 서비스를 구축, 테스트 및 배포할 수 있도록 합니다. 이는 팀의 민첩성을 높이고, 대규모 팀에서 발생할 수 있는 커뮤니케이션 속도 저하와 관리 오버헤드를 줄여 생산성을 향상시킵니다.

3. **종속성 최소화**: 마이크로서비스 아키텍처는 각 서비스가 독립적인 코드베이스와 데이터 저장소를 가지므로 코드 종속성이 최소화됩니다. 이는 새로운 기능 추가가 용이해집니다.

4. **기술 다양성**: 마이크로서비스는 적절한 기술 스택을 선택하여 서비스에 가장 적합한 기술을 사용할 수 있습니다. 팀은 혼합된 기술 스택을 조화롭게 활용할 수 있습니다.

5. **결함 격리**: 개별 마이크로서비스의 장애가 전체 애플리케이션에 영향을 주지 않도록 업스트림 서비스를 설계할 수 있습니다. 예를 들어 회로 차단기 패턴이나 비동기 메시징 패턴을 활용하여 장애 처리를 강화할 수 있습니다.

6. **확장성**: 마이크로서비스는 별도로 확장될 수 있어 전체 애플리케이션 규모를 확장하지 않고도 필요한 하위 시스템의 규모를 늘릴 수 있습니다. 오케스트레이터를 사용하여 서비스를 높은 밀도로 패킹하여 리소스 활용도를 향상시킬 수 있습니다.

7. **데이터 격리**: 각 마이크로서비스는 자체 데이터를 관리하므로 스키마 업데이트가 쉬워집니다. 이는 모놀리식 애플리케이션에서 발생할 수 있는 데이터 일관성 문제를 완화합니다.

마이크로서비스 아키텍처를 고려할 때 고려해야 할 사항:

1. **복잡성**: 마이크로서비스 애플리케이션은 모놀리식 애플리케이션보다 더 많은 작동 부분으로 인해 복잡성이 증가할 수 있습니다.

2. **개발 및 테스트**: 다른 서비스에 의존하는 서비스를 개발하고 테스트하는 것은 독립적인 모놀리식 애플리케이션과 다른 접근 방식을 요구할 수 있습니다.

3. **통제 부족**: 마이크로서비스의 분산 접근 방식은 언어와 프레임워크의 다양성을 증가시킬 수 있으며, 이에 대한 통제가 필요합니다. 일부 교차 기능에 대해 표준을 적용하는 것이 유용할 수 있습니다.

4. **네트워크 정체 및 대기 시간**: 서비스 간 통신량이 증가하고, 긴 서비스 종속성 체인이 있을 경우 네트워크 정체와 대기 시간 문제가 발생할 수 있습니다. 이를 해결하기 위해 API 디자인, 통신량 관리, 비동기 통신 패턴 등을 고려해야 합니다.

5. **데이터 무결성**: 각 마이크로서비스가 자체 데이터를 관리하기 때문에 데이터 일관성 유지가 중요합니다.

6. **관리**: 마이크로서비스를 성공적으로 운영하기 위해 성숙한 DevOps 문화가 필요합니다. 전체 서비스의 로깅과 모니터링은 도전적일 수 있습니다.

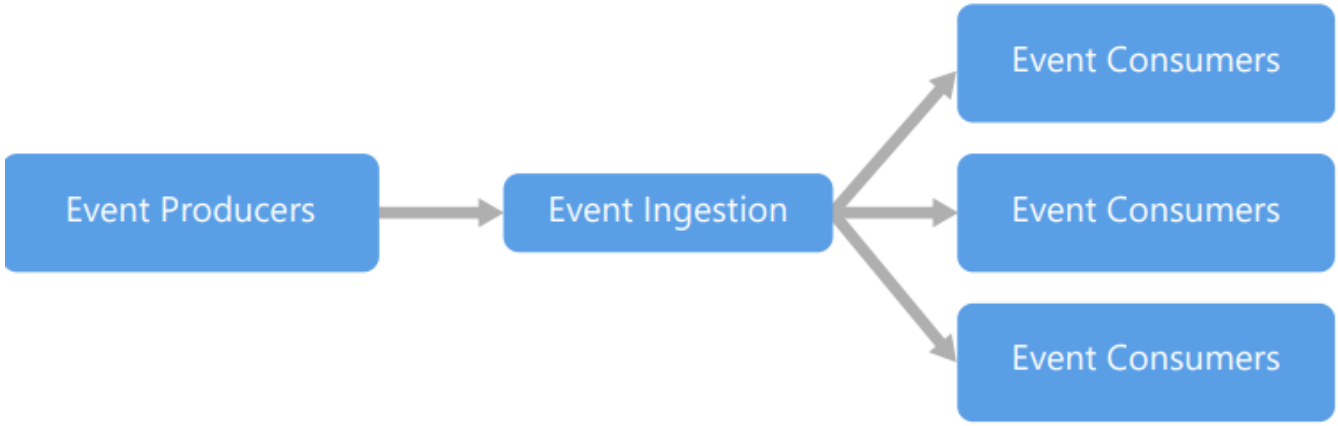
7. **버전 관리**: 서비스 업데이트 시 종속된 서비스에 영향을 주지 않도록 주의해야 합니다.

8. **기술 수준**: 마이크로서비스는 분산 시스템이므로 팀이 필요한 기술과 경험을 가지고 있는지 평가해야 합니다.

이벤트 기반 아키텍처(Event-Driven Architecture)

이벤트 기반 아키텍처는 시스템의 핵심 구조로서 이벤트의 캡처, 커뮤니케이션, 처리, 그리고 지속성에 초점을 둡니다. 이는 전통적인 요청 기반 모델과는 다른 접근 방식입니다. 이벤트는 시스템 하드웨어 또는 소프트웨어 상태의 변화나 중요한 사건 발생을 나타냅니다. 이벤트 소스는 내부나 외부 입력이 될 수 있습니다.

이벤트 기반 아키텍처는 이벤트를 생성하는 이벤트 생산자와 이벤트를 수신 대기하는 이벤트 소비자로 구성됩니다.



이벤트는 거의 실시간으로 전달되기 때문에 이벤트가 발생하면 즉시 소비자가 이벤트에 응답할 수 있습니다. 생산자와 소비자는 분리되어 있기 때문에 생산자는 수신 대기 중인 소비자를 알 수 없습니다. 또한 소비자도 서로 분리되어 각자의 이벤트를 볼 수 있습니다. 이는 소비자가 메시지 큐에서 메시지를 가져와 처리하는 경쟁 소비자 패턴과는 다릅니다. 일부 시스템(예: IoT)에서는 매우 높은 이벤트 수를 처리해야 하는 경우도 있습니다.

이벤트 기반 아키텍처는 게시/구독(pub/sub) 모델이나 이벤트 스트림 모델을 사용할 수 있습니다.

- 게시/구독 모델: 메시징 인프라에서 구독을 추적하고, 이벤트가 게시되면 각 구독자에게 이벤트를 보냅니다. 이벤트를 받은 후에는 재생할 수 없으며, 새로운 구독자는 이전 이벤트를 볼 수 없습니다.

- 이벤트 스트리밍: 이벤트가 로그에 기록되며, 엄격한 순서로 파티셔닝되고 지속 가능합니다. 클라이언트는 스트림을 구독하지 않고, 대신 스트림의 일부에서 읽을 수 있습니다. 클라이언트는 스트림에서 해당 위치를 진행해야 합니다. 즉, 클라이언트는 언제든지 연결하여 이벤트를 재생할 수 있습니다.

이벤트 소비자의 관점에서 몇 가지 일반적인 변형이 있을 수 있습니다:

1. 단순 이벤트 처리: 이벤트가 발생하면 즉시 소비자에서 작업을 트리거합니다. 예를 들어, 서비스 버스 토픽에 메시지를 게시할 때마다 Azure Functions와 같은 서비스를 트리거할 수 있습니다.

2. 복합 이벤트 처리: 소비자는 Azure Stream Analytics와 같은 기술을 사용하여 연속적인 이벤트 데이터의 패턴을 찾아 처리합니다. 예를 들어, 특정 기간 동안 장치에서 읽은 값을 집계하고 이동 평균이 특정 임계값을 초과하면 알림을 생성할 수 있습니다.

3. 이벤트 스트림 처리: Azure IoT Hub나 Apache Kafka와 같은 데이터 스트리밍 플랫폼을 사용하여 이벤트를 수집하고 스트림 프로세서에 공급합니다. 스트림 프로세서는 스트림을 처리하거나 변환하며, 각 하위 시스템에는 여러 개의 스트림 프로세서가 사용될 수 있습니다. 이 접근 방식은 주로 IoT 워크로드에 적합합니다.

이벤트의 소스가 시스템 외부에 있는 경우(예: IoT 솔루션의 물리적 디바이스), 시스템은 필요한 볼륨과 처리량으로 데이터를 수집할 수 있어야 합니다.

논리 다이어그램에서는 각 소비자 유형을 단일 상자로 표시했지만, 실제로는 단일 소비자의 여러 인스턴스가 있어야 합니다. 이벤트의 볼륨과 빈도를 처리하기 위해 여러 인스턴스가 필요할 수 있습니다. 또한 단일 소비자가 여러 스레드에서 이벤트를 처리할 수도 있습니다. 이벤트가 순차적으로 처리되어야 하거나 정확히 한 번만 처리되어야 하는 경우 이를 고려해야 합니다.

이벤트 기반 아키텍처는 여러 하위 시스템이 동일한 이벤트를 처리해야 하는 상황이 있거나, 최소한의 지연 시간으로 실시간 처리가 필요하거나, 복합 이벤트 처리나 높은 볼륨 및 데이터 개발 속도(예: IoT)가 요구되는 경우에 유용합니다.

아키텍처를 사용하는 경우에는 다음과 같은 기능을 구현해야 합니다:

1. 전달 보장 (Guaranteed Delivery): 시스템에서는 이벤트가 안전하게 전달되어야 합니다. 특히 IoT 시나리오와 같은 경우, 이벤트의 안정적인 전달이 매우 중요합니다. 이벤트 손실이나 중복을 방지하기 위해 메시지 큐 또는 메시지 브로커 등의 기술을 사용하여 전달 보장을 구현할 수 있습니다.

2. 이벤트 순서와 중복 처리 (Event Ordering and Deduplication): 이벤트가 순서대로 처리되거나 중복 처리되지 않도록 해야 합니다. 일반적으로 각 소비자 유형은 여러 인스턴스에서 실행되어 복원성과 확장성을 갖추도록 설계됩니다. 이를 위해 이벤트에 고유한 식별자를 부여하거나 중복 이벤트를 제거하는 방법을 사용하여 순서와 중복 처리를 관리할 수 있습니다.

이러한 아키텍처의 사용은 여러 가지 이점을 제공합니다:

1. 생산자와 소비자의 분리: 생산자는 이벤트를 생성하고 전송하기만 하면 되며, 소비자는 독립적으로 이벤트를 처리할 수 있습니다. 이를 통해 시스템의 확장성과 유연성이 향상됩니다.

2. 지점 간 통합 없음: 새로운 소비자를 쉽게 추가할 수 있습니다. 이벤트를 구독하는 새로운 하위 시스템을 간단하게 구축하고 연결할 수 있습니다.

3. 실시간 처리의 최소 시간 지연: 이벤트가 거의 실시간으로 전달되므로 소비자는 이벤트에 대한 즉각적인 응답이 가능합니다. 이는 실시간 처리 요구사항을 충족시키는 데 도움이 됩니다.

4. 확장성과 배포 가능성: 이벤트 기반 아키텍처는 확장성이 용이하며, 클러스터링, 파티셔닝, 분산 처리 등을 통해 대량의 이벤트 처리를 처리할 수 있습니다. 또한, 이벤트 기반 시스템은 여러 서버 또는 컴퓨팅 리소스에 분산되어 배포될 수 있습니다.

5. 독립적인 이벤트 스트림 확인: 각 하위 시스템은 독립적으로

이벤트 스트림을 확인하고 처리할 수 있습니다. 이는 하위 시스템 간의 결합도를 낮추고, 각 하위 시스템이 자체적으로 이벤트를 처리하고 활용할 수 있는 유연성을 제공합니다.

이러한 기능과 이점을 고려하여 이벤트 기반 아키텍처를 구현할 수 있습니다.