How to design a web application: software architecture 101

목차

1. What is software architecture?
2. Why is software architecture important?
3. The difference between software architecture and software design.
4. Software architecture patterns.
5. How to decide on the number of tiers your app should have.
6. Horizontal or vertical scaling — which is right for your app?
7. Monolith or microservice?
8. When should you use NoSQL or SQL?
9. Picking the right technology for the job.
10. How to become a software architect.
11. Where to go from here.

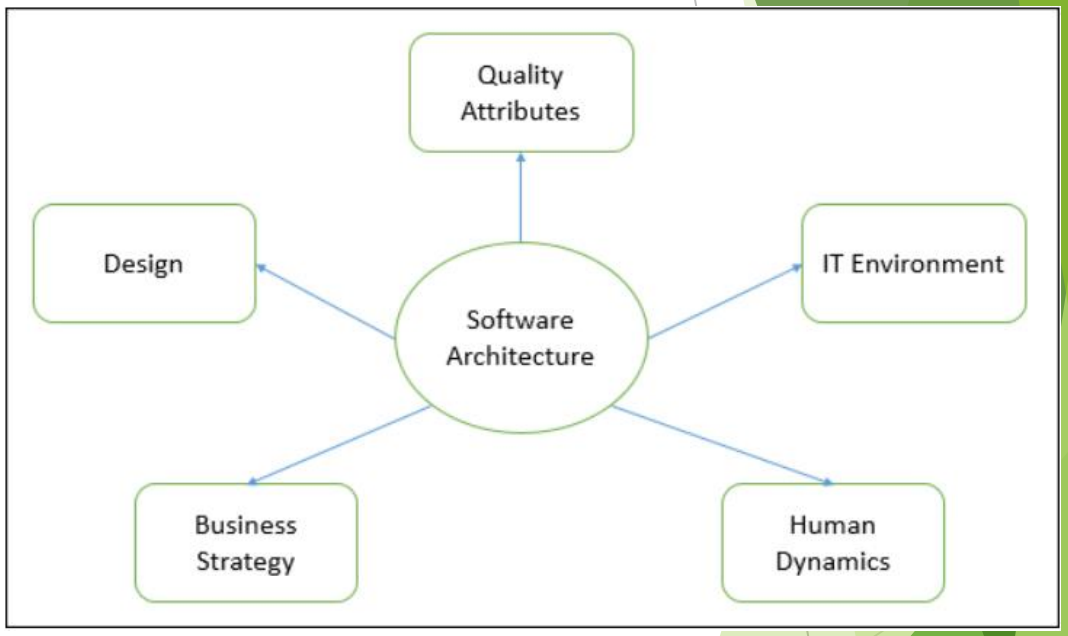
1. What is software architecture?

시스템의 소프트웨어 아키텍처는 주요 구성 요소, 해당 관계 및 상호 작용 방식을 설명

기본적으로 청사진 역할을 함

시스템 복잡성을 관리하기 위한 추상화를 제공

구성 요소 간의 통신 및 조정을 설정



아키텍처는 성능 및 보안을 최적화한다는 공통의 목표와 함께 모든 기술 및 운영 요구 사항을 충족하는 솔루션을 정의하는데 도움이 된다.

아키텍처를 설계하려면 조직의 요구와 개발팀의 요구가 교차된다. 각 결정은 품질, 유지 관리, 성능 등에 상당한 영향을 줄 수 있다.

2. Why is software architecture important?

무엇이든 성공적으로 만들려면 베이스를 제대로 확보해야 한다.

베이스를 제대로 하지 못하면 무엇인가 잘못 될 것이고, 이때에는 다시 처음부터 시작하는 방법밖에 없습니다.

웹 애플리케이션 또한 마찬가지입니다. 나중에 중요한 디자인 변경 및 코드 리팩토링을 피하기 위해 아키텍처는 신중하게 고려되어야 한다.

초기 설계 단계에서 성급한 결정을 내리면 개발 프로세스의 모든 단계에서 문제가 발생할 수 있다.

소프트웨어 개발은 반복적이고 진화적인 과정이고, 항상 완벽한 제품을 얻을 수 있는 것은 아니다. 하지만 우리는 이것을 그냥 지나치면 안된다.

3. The difference between software architecture and software design

소프트웨어 아키텍처는 시스템의 기본 구성 요소와 상위 구성 요소 및 이것들이 함께 작동하는 방식을 정의한다.

소프트웨어 아키텍처가 한 소프트웨어의 뼈대나 고수준의 기반을 담당하는 데에 반해, 소프트웨어 디자인은 각각의 모듈들이 어떤 것을 하는지, 클래스의 범위, 함수의 목적 등 코드 수준의 디자인을 담당한다.

4. Software architecture patterns

아키텍처 패턴이란?

아키텍처 패턴이란 주어진 상황에서의 소스트웨어 아키텍처에서 일반적으로 발생하는 문제점들에 대한 일반화되고 재사용 가능한 솔루션이다.

아키텍처 패턴은 소프트웨어 디자인 패턴과 유사하지만 더 큰 범주에 속한다.

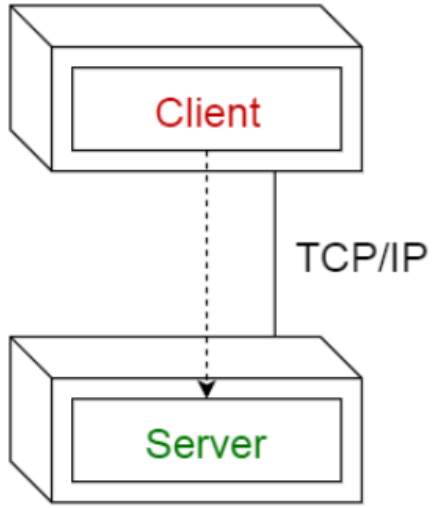
참고 링크

https://mingrammer.com/translation-10-common-software-architectural-patterns-in-a-nutshell/#8-%EB%AA%A8%EB%8D%B8-%EB%B7%B0-%EC%BB%A8%ED%8A%B8%EB%A1%A4%EB%9F%AC-%ED%8C%A8%ED%84%B4-model-view-controller-pattern

4. Software architecture patterns

클라이언트-서버 패턴 (Client-server pattern)

이 패턴은 하나의 서버와 다수의 클라이언트, 두 부분으로 구성된다.
서버 컴포넌트는 다수의 클라이언트 컴포넌트로 서비스를 제공한다.
클라이언트가 서버에 서비스를 요청하면 서버는 클라이언트에게 적절한 서비스를 제공한다.
또한 서버는 계속 클라이언트로부터의 요청을 대기한다.



활용

- 이메일, 문서 공유 및 은행 등의 온라인 애플리케이션

4. Software architecture patterns

계층화 패턴 (Layered pattern)

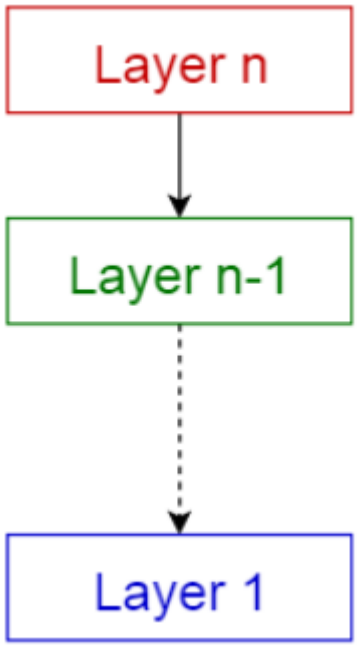
이 패턴은 n-티어 아키텍처 패턴이라고도 불린다.

이는 하위 모듈들의 그룹으로 나눌 수 있는 구조화된 프로그램에서 사용할 수 있다.

각 하위 모듈들은 특정한 수준의 추상화를 제공한다.

각 계층은 다음 상위 계층에 서비스를 제공한다.

- 프레젠테이션 계층 (Presentation layer) - UI 계층 (UI layer) 이라고도 함
- 애플리케이션 계층 (Application layer) - 서비스 계층 (Service layer) 이라고도 함
- 비즈니스 논리 계층 (Business logic layer) - 도메인 계층 (Domain layer) 이라고도 함
- 데이터 접근 계층 (Data access layer) - 영속 계층 (Persistence layer) 이라고도 함



활용

- 일반적인 데스크톱 애플리케이션
- E-commerce 웹 애플리케이션

4. Software architecture patterns

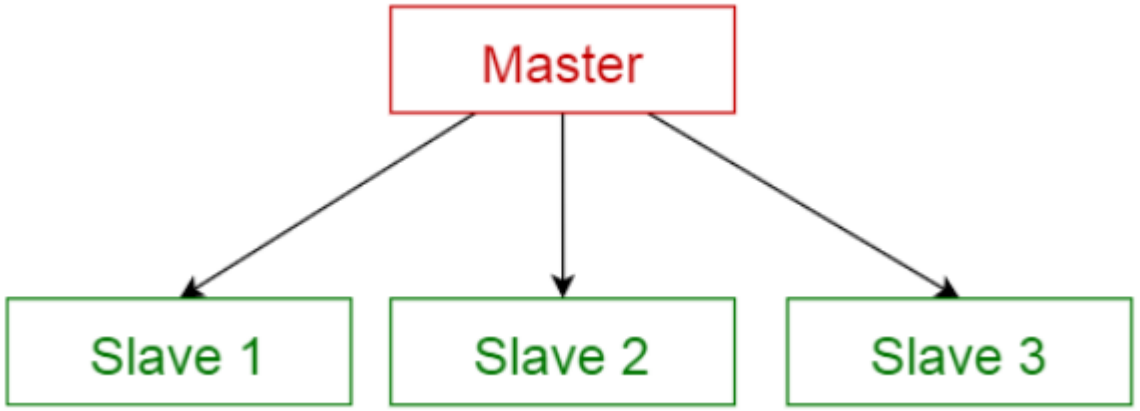
마스터-슬레이브 패턴 (Master-slave pattern)

이 패턴은 마스터와 슬레이브 두 부분으로 구성된다.

마스터 컴포넌트는 동등한 구조를 지닌 슬레이브 컴포넌트들로 작업을 분산하고, 슬레이브가 반환한 결과값으로부터 최종 결과값을 계산한다.

활용

- 데이터베이스 복제에서, 마스터 데이터베이스는 신뢰할 수 있는 데이터 소스로 간주되며 슬레이브 데이터베이스는 마스터 데이터 베이스와 동기화된다.
- 컴퓨터 시스템에서 버스와 연결된 주변장치 (마스터 드라이버와 슬레이브 드라이버)



4. Software architecture patterns

파이프-필터 패턴 (Pipe-filter pattern)

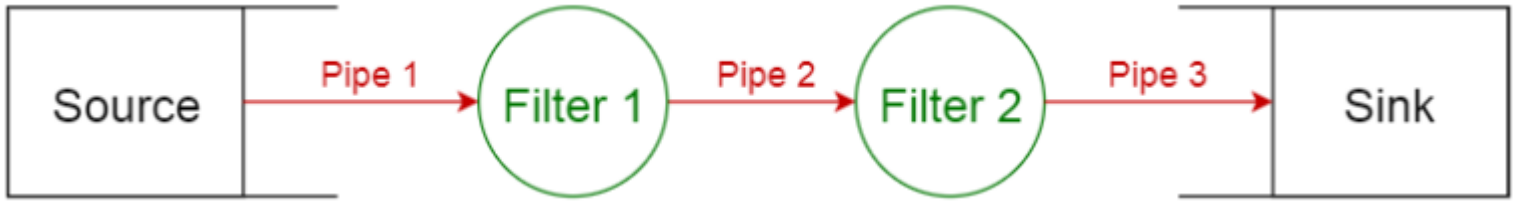
이 패턴은 데이터 스트림을 생성하고 처리하는 시스템에서 사용할 수 있다.

각 처리 과정은 필터 (filter) 컴포넌트에서 이루어지며, 처리되는 데이터는 파이프 (pipes) 를 통해 흐른다.

이 파이프는 버퍼링 또는 동기화 목적으로 사용될 수 있다.

활용

- 컴파일러, 연속한 필터들은 어휘 분석, 파싱, 의미 분석 그리고 코드 생성을 수행한다.
- 생물정보학에서의 워크플로우



4. Software architecture patterns

브로커 패턴 (Broker pattern)

이 패턴은 분리된 컴포넌트들로 이루어진 분산 시스템에서 사용된다.

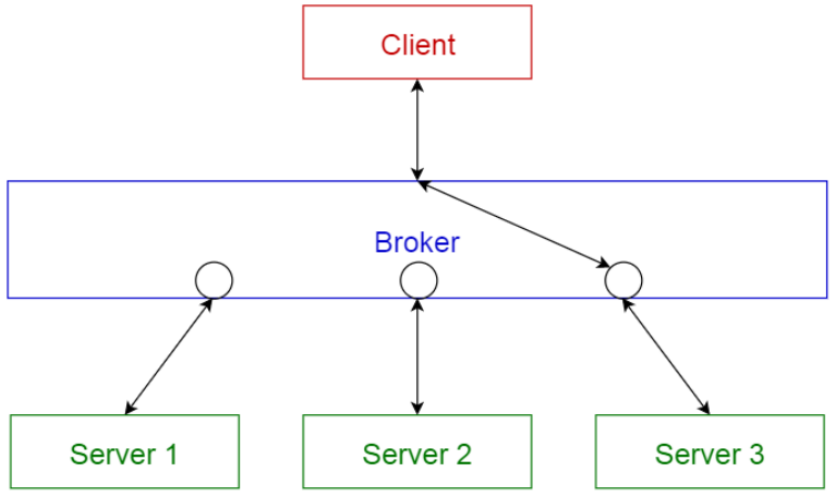
이 컴포넌트들은 원격 서비스 실행을 통해 서로 상호 작용을 할 수 있다.

브로커 (broker) 컴포넌트는 컴포넌트 (components) 간의 통신을 조정하는 역할을 한다.

서버는 자신의 기능들(서비스 및 특성)을 브로커에 넘겨주며 (publish), 클라이언트가 브로커에 서비스를 요청하면 브로커는 클라이언트를 자신의 레지스트리에 있는 적합한 서비스로 리디렉션한다.

활용

- Apache AcriveMQ, apache Kafka, RabbitMQ 및 Jboss Messaging 와 같은 메시지 브로커 소프트웨어



4. Software architecture patterns

이벤트-버스 패턴 (Event-bus pattern)

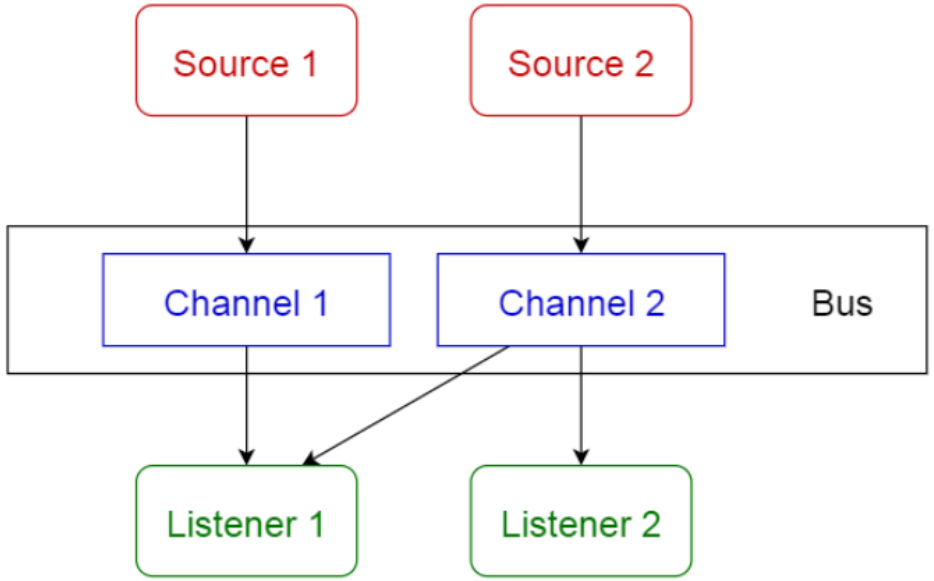
이 패턴은 주로 이벤트를 처리하며 이벤트 소수 (event source), 이벤트 리스너 (event listener), 채널 (channel) 그리고 이벤트 버스 (event bus) 의 4가지 주요 컴포넌트들을 갖는다.

소스는 이벤트 버스를 통해 특정 채널로 메시지를 발행하며 (publish), 리스너는 특정 채널에서 메시지를 구독 (subscribe) 한다.

리스너는 이전에 구독한 채널에 발행된 메시지에 대해 알림을 받는다.

활용

- 안드로이드 개발
- 알림 서비스



4. Software architecture patterns

블랙보드 패턴 (Blackboard pattern)

이 패턴은 결정 가능한 해결 전략이 알려지지 않은 문제에 유용하다.

이 패턴은 3가지 주요 컴포넌트로 구성된다.

- 블랙보드 (blackboard) - 솔루션의 객체를 포함하는 구조화된 전역 메모리
- 지식 소스 (knowledge source) - 자체 표현을 가진 특수 모듈
- 제어 컴포넌트 (control component) - 모듈 선택, 설정 및 실행을 담당

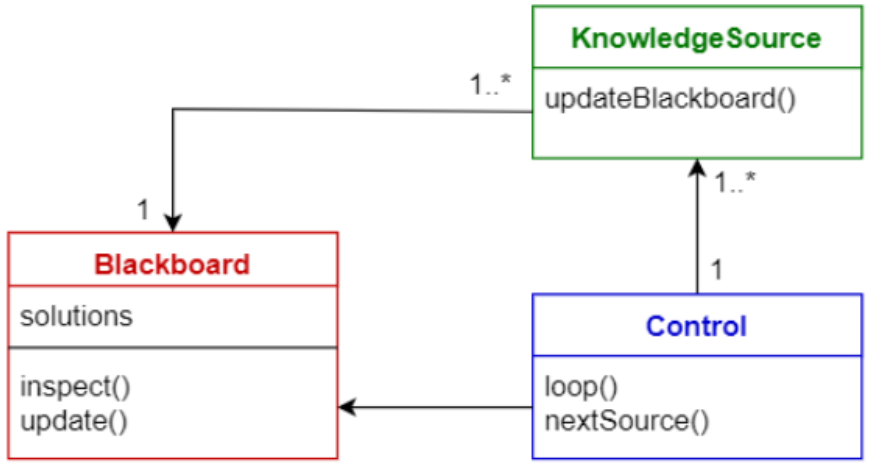
모든 컴포넌트는 블랙보드에 접근한다.

컴포넌트는 블랙보드에 추가되는 새로운 데이터 객체를 생성할 수 있다.

컴포넌트는 블랙보드에서 특정 종류의 데이터를 찾으며, 기존의 지식 소스와의 패턴 매칭으로 데이터를 찾는다.

활용

- 음성 인식
- 차량 식별 및 추적
- 단백질 구조 식별
- 수중 음파 탐지기 신호 해석



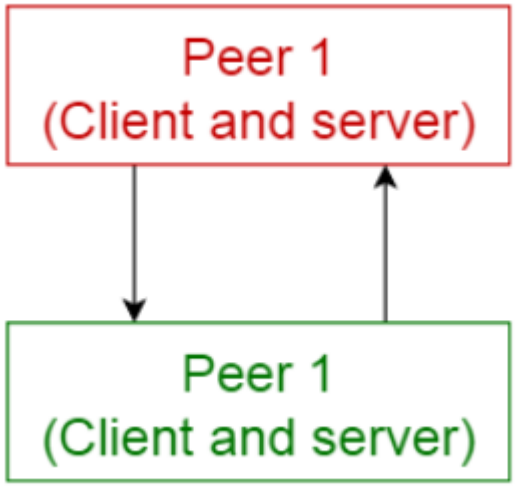
4. Software architecture patterns

피어 투 피어 패턴 (Peer-to-peer pattern)

이 패턴에서는, 각 컴포넌트를 피어 (peers) 라고 부른다. 피어는 클라이언트로서 피어에게 서비스를 요청할 수도 있고, 서버로서 각 피어에게 서비스를 제공할 수도 있다. 피어는 클라이언트 또는 서버 혹은 둘 모두로서 동작할 수 있으며, 시간이 지남에 따라 역할이 유동적으로 바뀔 수 있다.

활용

- Gnutella 나 G2 와 같은 파일 공유 네트워크
- P2PTV 나 PDTP 와 같은 멀티미디어 프로토콜
- Spotify 와 같은 독점적 멀티미디어 애플리케이션



4. Software architecture patterns

Model-View-Controller (MVC)

MVC 패턴이라고도 하는 이 패턴은 대화형 애플리케이션 (interactive application)을 다음의 3 부분으로 나눈 것이다.

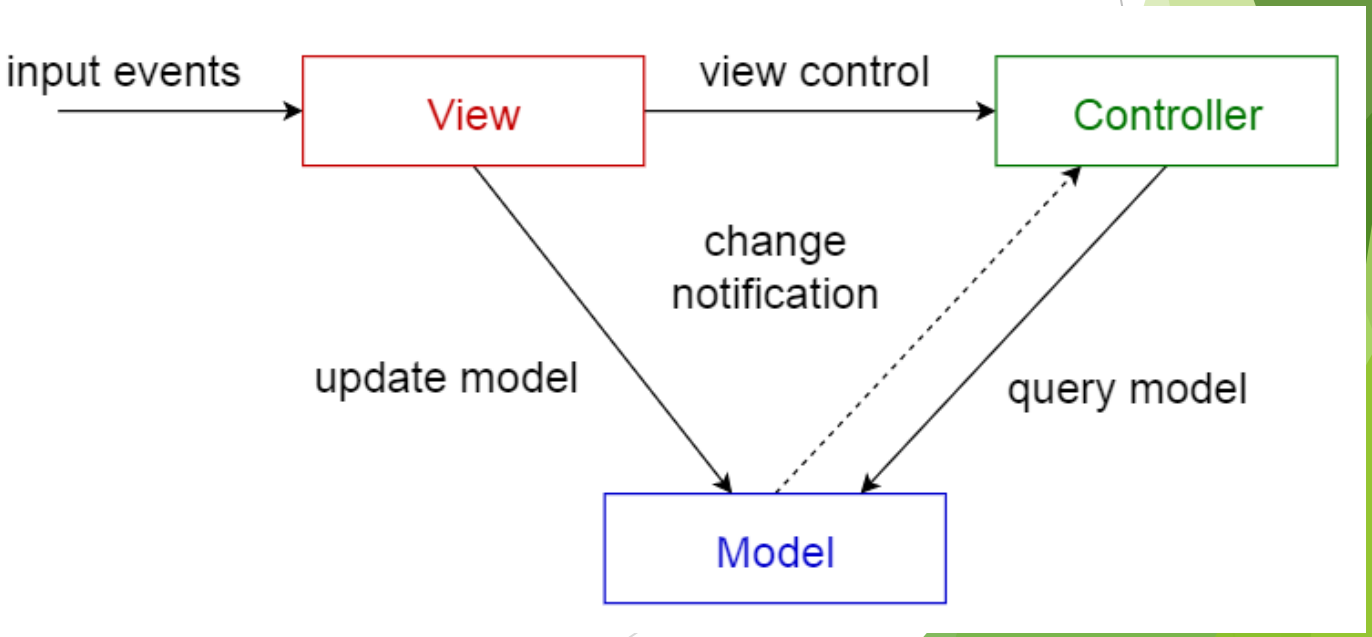
1. 모델 (model) - 핵심 기능과 데이터를 포함한다.
2. 뷰 (view) - 사용자에게 정보를 표시한다. (하나 이상의 뷰가 정의될 수 있음)
3. 컨트롤러 (controller) - 사용자로부터의 입력을 처리한다.

이는 정보가 사용자에게 제공되는 방식과 사용자로부터 받아 들여지는 방식에서 정보의 내부적인 표현을 분리하기 위해 나뉘어진다.

이는 컴포넌트를 분리하며 코드의 효율적인 재사용을 가능케한다.

활용

- 일반적인 웹 애플리케이션 설계 아키텍처
- Django 나 Rails 와 같은 웹 프레임워크



4. Software architecture patterns

인터프리터 패턴 (Interpreter pattern)

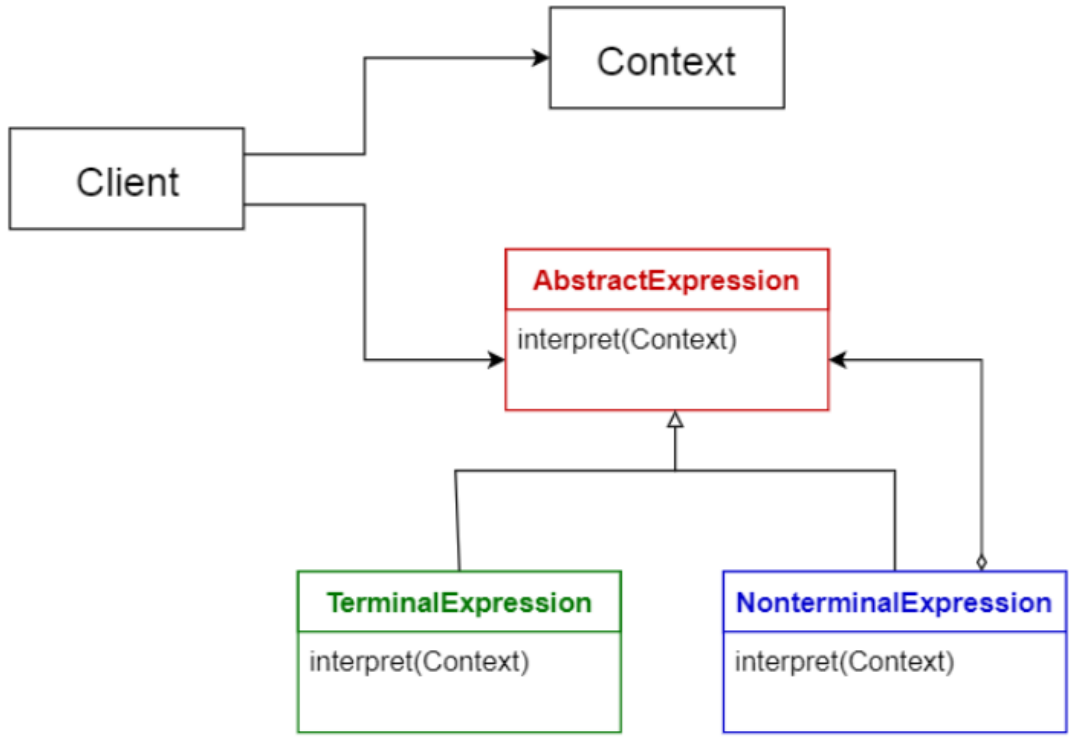
이 패턴은 특정 언어로 작성된 프로그램을 해석하는 컴포넌트를 설계할 때 사용된다.

이는 주로 특정 언어로 작성된 문장 혹은 표현식이라고 하는 프로그램의 각 라인을 수행하는 방법을 지정한다.

기본 아이디어는 언어의 각 기호에 대해 클래스를 만드는 것이다.

활용

- SQL과 같은 데이터베이스 쿼리 언어
- 통신 프로토콜 정의하기 위한 언어



5. How Many Tiers Should Your App Have?

Single tier application

장점 :

- 네트워크 대기시간이 없다.
- 데이터를 쉬고 빠르게 이용할 수 있다.
- 데이터는 네트워크를 통해 전송되지 않으므로 데이터 안전이 보장된다.

단점 :

- 응용 프로그램에 대한 통제력이 거의 없다. - 새로운 기능이나 코드 변경 사항이 새롭게 나오면 구현하기가 힘들다.
- 최소한의 실수로 테스트를 철저히 수행해야한다.
- 단일 계층 응용 프로그램은 조정 또는 역설계에 취약하다.

5. How Many Tiers Should Your App Have?

Two-tier application

장점 :

- 이 패턴은 특정 언어로 작성된 프로그램을 해석하는 컴포넌트를 설계할 때 사용된다.
- 데이터베이스 서버와 비즈니스 로직은 물리적으로 가깝고 더 높은 성능을 제공한다.

단점 :

- 클라이언트는 대부분의 응용 프로그램 논리를 보유하므로 소프트웨어 버전을 제어하고 새 버전을 다시 배포 할 때 문제가 발생한다.
- 제한된 수의 사용자만 지원하므로 확장 성이 부족하다. 여러 개의 클라이언트 요청이 증가하면 클라이언트가 별도의 연결과 CPU 메모리를 진행해야 하므로 응용 프로그램 성능이 저하 될 수 있다.
- 응용 프로그램 논리가 클라이언트와 연결되어 있으므로 논리를 재사용하기가 어렵다.

5. How Many Tiers Should Your App Have?

Three-tier application

장점 :

- 데이터베이스 업데이트를 위해 중간 계층으로 전달 된 데이터가 유효성을 보장하므로 클라이언트 응용 프로그램을 통한 데이터 손상을 제거 할 수 있다.
- 중앙 집중식 서버에 비즈니스 로직을 배치하면 데이터가 더 안전해 진다.
- 응용 프로그램 서버의 분산 배포로 인해 각 클라이언트와 별도의 연결이 필요하지 않지만 일부 응용 프로그램 서버의 연결이 충분하기 때문에 시스템의 확장 성이 향상된다.

단점 :

- 일반적으로, 통신 포인트가 증가하고(클라이언트에서 서버로 직접 전송하는 대신 middle-tier 서버로), Visual Basic, PowerBuilder, Delphi와 같은 도구로 성능이 향상됨에 따라 Three-tier application 프로그램을 만들 때 더 많은 노력을 기울여야 한다.

5. How Many Tiers Should Your App Have?

N-tier application

장점 :

- 모든 Three-tier 아키텍처의 장점으 가지고 있다.
- 데이터베이스 계층과 클라이언트 계층의 오프로드로 인해 성능이 향상 되어 중소 규모 산업에 적합하다.

단점 :

- 계층의 구성 요소로 인해 복잡한 구조를 구현하거나 유지하기가 어렵다.

5. How Many Tiers Should Your App Have?

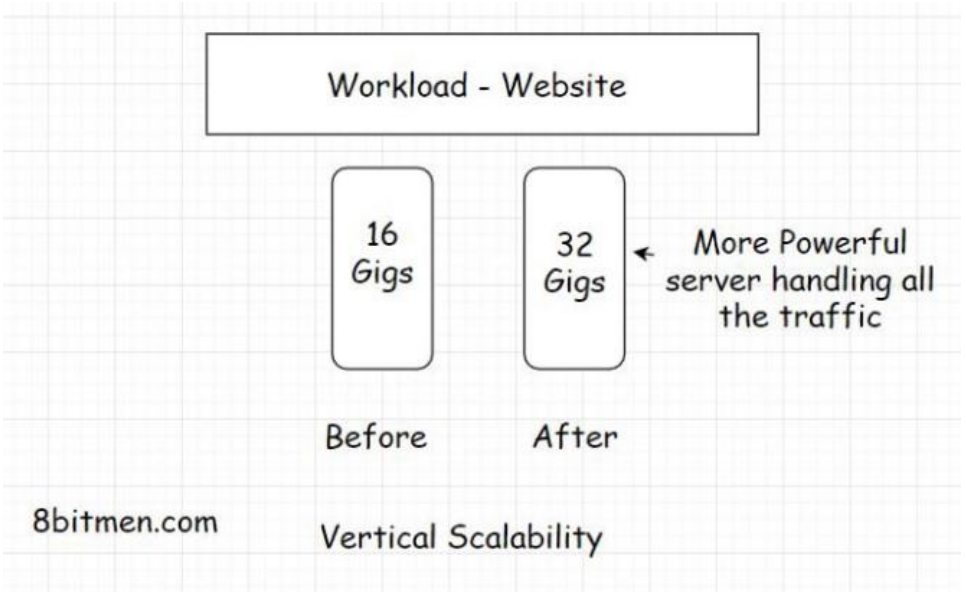
Conclusion

- 네트워크 대기 시간을 원하지 않는 경우는 single-tier application 을 사용하면 된다.
- 네트워크 대기 시간을 최소화 해야 하고 응용 프로그램 내에서 더 많은 데이터를 제어 해야 하는 경우 Two-tier application 을 사용하면 된다.
- 응용 프로그램의 코드 / 비즈니스 논리를 제어 해야 하고 보안을 유지하고 응용 프로그램의 데이터를 제어 해야 하는 경우 Three-tier application 을 사용하면 된다.
- 많은 양의 데이터를 확장하고 처리하기 위해 응용 프로그램이 필요할 때 N 계층 아키텍처를 선택해야 한다.

6. Horizontal or vertical scaling — which is right for your app?

Vertical scaling

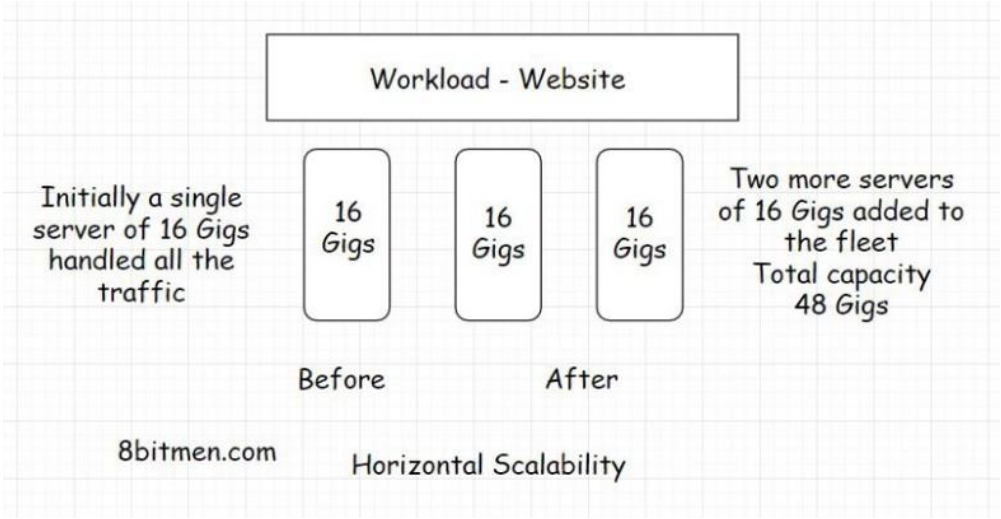
- 앱이 최소한의 일관된 트래픽(예 : 조직의 내부 도구)을 받을 것으로 예상되는 유틸리티 또는 도구인 경우 분산 환경에서 호스팅 하지 않아도 된다.
- 단일 서버로 트래픽을 관리하기에 충분하며 트래픽로드가 크게 증가하지 않음을 알 수 있다.



6. Horizontal or vertical scaling — which is right for your app?

Horizontal

- 앱이 소셜 네트워크, 피트니스 앱 또는 이와 유사한 것과 같은 공개 소셜 앱인 경우 가까운 시일 내에 트래픽이 기하 급수적으로 급증 할 것으로 예상 된다.
- 이 경우 고가용 성과 수평 확장 성이 모두 중요하다.

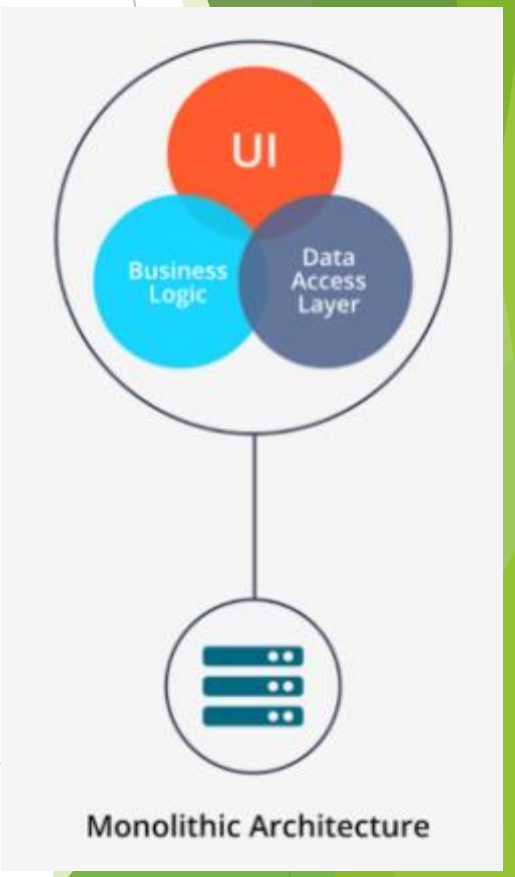


✓ 클라우드에 구축하고 처음부터 항상 수평적 확장성을 염두해 주고 구축해야 한다.

7. Monolith or microservice?

When to use monolithic architecture

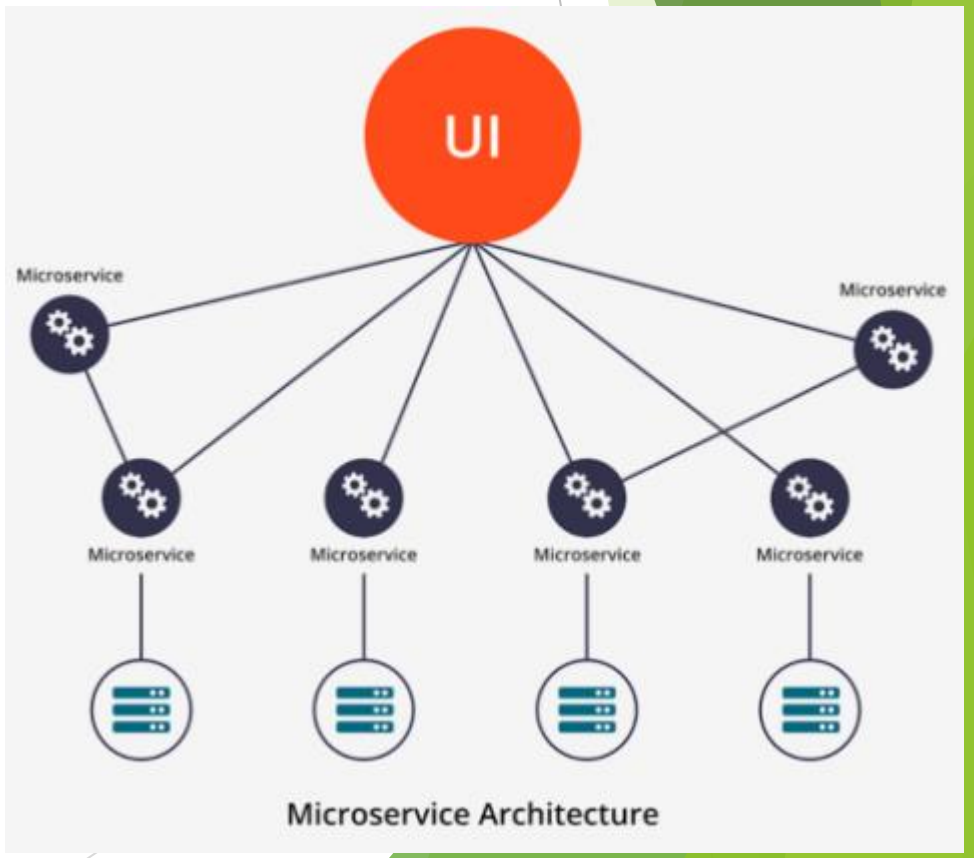
- Monolithic 애플리케이션은 요구 사항이 단순하고 앱이 제한된 양의 트래픽을 처리할 것으로 예상되는 경우에 가장 적합하다.
- 예를 들어, 조직의 내부 세금 계산 앱 또는 이야 유사한 공개 도구이다.
- 비즈니스가 시간이 지남에 따라 사용자 기반과 트래픽이 기하 급수적으로 증가 하지 않을 것으로 확신하는 사용 사례이다.
- 개발자 팀이 monolithic 아키텍처로 시작하여 나중에 분산 마이크로 서비스 아키텍처로 확장한 사례도 있다.
- 이는 필요할 때마다 단계별로 응용 프로그램의 복잡성을 처리하는 데 도움이 된다.
- LinkedIn에서 monolithic 아키텍처를 사용한다.



7. Monolith or microservice?

When to use microservice architecture

- Microservice 아키텍처는 복잡한 소셜 네트워크 애플리케이션과 같이 복잡한 사용 사례와 향후 트래픽이 기하 급수적으로 증가 할 것으로 예상되는 앱에 적합하다.
- 일반적인 소셜 네트워킹 응용 프로그램에는 메시징, 실시간 채팅, 라이브 비디오 스트리밍, 이미지 업로드, 좋아요 및 공유 기능 등과 같은 다양한 구성 요소가 있다.
- 이 경우 각 구성 요소를 별도로 개발하여 단일 책임과 우려 사항을 분리하여 유지하는 것이 좋다.
- 단일 코드베이스로 작성된 모든 기능은 mess로 될 시간이 없다.



7. Monolith or microservice?

Conclusion

세 가지 접근 방식

- Monolithic architecture
- Microservice architecture
- Monolithic architecture 로 시작한 수 나중에 microservice architecture 로 확장

Monolithic 또는 microservice 를 선택하는 것은 주로 사용 사례에 달려 있다.

일은 단순하게 유지하고 요구 사항은 철저히 이해하는것이 좋다.

8. When should you use NoSQL or SQL?

When to pick a SQL database?

주식 거래, 은행 또는 금융 기반 앱을 작성하거나 Facebook 과 같은 소셜 네트워킹 앱을 작성할 때 많은 관계를 저장 해야 하는 경우 관계형 데이터베이스를 선택하는 것이 좋다.

- Transactions and Data Consistency
  돈이나 숫자와 관련이 있거나 거래를 하거나 ACID를 준수해야하는 소프트웨어를 작성하는 경우 데이터 일관성이 매우 중요하다.
  관계형 DB는 트랜잭션 및 데이터 일관성과 관련하여 빛을 발한다.
  ACID 규칙을 준수하고 오랜 세월 테스트를 거쳐 왔다.
- Storing Relationship
  데이터가 특정 소시에 사는 친구와 같은 많은 관계가 있다면?
  오늘 방문 할 식당에서 어느 친구가 이미 음식을 먹었는지?
  이런 종류의 데이터를 저장하는 관계형 데이터베이스보다 더 좋은 것은 없다.

Popular relational databases

- MySQL
- Microsoft SQL Server
- PostgreSQL
- MariaDB

8. When should you use NoSQL or SQL?

When to pick a NoSQL database?

NoSQL 데이터베이스를 선택해야하는 몇 가지 이유가 있다.

- Handling a large number of read-write operations
  빠르게 확장해야하는 경우 NoSQL 데이터베이스를 사용하는게 적합하다.
  예를 들어 웹 사이트에 많은 수의 읽기 / 쓰기 작업이 있고 많은 양의 데이터를 처리 할 때 NoSQL 데이터베이스가 이러한 시나리오에 가장 적합하다.
  노드를 즉시 추가 할 수 있으므로 대기 시간을 최소화하면서 더 많은 동시 트래픽과 대량의 데이터를 처리 할 수 있다.
- Running data analytics
  NoSQL 데이터베이스는 또한 대량의 데이터 유입을 처리해야하는 데이터 분석 사용 사례에 가장 적합하다.

Popular NoSQL databases

- MongoDB
- Redis
- Cassandra
- HBASE

9. Picking the right technology for the job

Real-time data interaction

필요한 앱을 빌드하는 경우 :

- 메시징 응용 프로그램 또는 Spotify, Netflix 등과 같은 오디오-비디오 스트리밍 응용 프로그램과 같은 백엔드 서버와 실시간으로 상호 작용
- 클라이언트와 서버 간의 지속적인 연결 및 백엔드의 non-blocking technology

이러한 앱을 작성할 수 있는 인기 기술 중 일부는 node.js 및 python 프레임워크인 Tornado 이다.

Peer-to-peer web application

P2P 분산 검색 엔진 또는 P2P, 라이브 TV, 라디오 서비스 (예: Microsoft 의 LiveStation 과 유사한 것)와 같은 P2P 웹 응용 프로그램을 구축하려는 경우 DAT 및 IPFS와 같은 JavaScript 프로토콜을 살펴봐야 한다.

CRUD-based regular application

일반 CRUD 기반 앱과 같은 간단한 사용 사례가 있는 경우 사용할 수 있는 기술 중 일부는 Spring MVC, Python Django, Ruby on Ralis, PHP Larvavel 및 ASP .NET MVC 가 있다.

9. Picking the right technology for the job

Simple, small scale applications

블로그나 간단한 온라인 양식 도는 포털의 Iframe 내에서 소셜 미디어와 통합되는 간단한 앱과 같이 복잡하지 않은 앱을 작성하려면 PHP를 사용하면 된다.
또한, 스프링 부트, Ruby on Rails 와 같은 다른 웹 프레임워크를 고려할 수도 있다.
이 프레임워크는 세부 사항, 구성, 개발 시간을 노치별로 줄이고 바른 개발을 촉진시킨다.
그러나 PHP 호스팅은 다른 기술을 호스팅하는 것에 비해 훨씬 저렴하다.
매우 간단한 사례에 이상적이다.

CPU and memory-intensive applications

성능이 뛰어나고 확장 가능한 분산 시스템을 작성하기 위해 업계에서 일반적으로 사용되는 기술은 C++ 이다.
저수준 메모리 조작을 용이하게 하는 기능이 있어 분산 시스템을 작성할 때 개발자에게 메모리를 보다 강력하게 제어 할 수 있다.
암호 화폐의 대부분은 이 언어를 사용하여 작성되었다.
Rust는 C++ 과 유사한 프로그래밍 언어이다.
고성능 및 안전한 동시성을 위해 만들어진 언어이다.
최근 개발자들 사이에서 인기가 높아지고 있다.
Java, Scala 및 Erlang도 좋은 언어이고, 대부분의 대규모 엔터프라이즈 시스템은 Java로 작성된다.
Go 는 멀티 코어 머신 용 앱을 작성하고 많은 양의 데이터를 처리하기 위한 Google 의 프로그래밍 언어이다.
Julia 는 고성능 및 실행 계산 및 수치 분석을 위해 동적으로 사용하는 프로그래밍 언어이다.

10. How to become a software architect.

대부분 소프트웨어 아키텍처로 일을 시작하지는 않고, 몇 년간 경험을 쌓고 시작한다.

소프트웨어 아키텍처에 익숙해지는 가장 좋은 방법은 자신의 웹 응용 프로그램을 디자인 하는 것이다.
이를 통해 로드 밸런싱, 메시지 큐잉, 스트림 처리, 캐싱 등 애플리케이션의 다양한 측면을 생각할 수 있다.
이러한 개념이 앱에 어떻게 적용되는지 이해할 수 있다면 소프트웨어 아키텍처가 될 수 있다.

주목받는 소프트웨어 아키텍처가 되고 싶다면 지속적으로 지식을 확장해야 하고 최신 업계 동향을 파악해야 한다.
하나 이상의 프로그래밍 언어를 배우는 것으로 시작하여 소프트웨어 개발자로 일하면서 역량을 키울 수 있다.

대학에서는 소프트웨어 아키텍처 학위를 취득할 수는 없지만 유용한 다른 과정들이 많다.

11. Where to go from here.

클라이언트-서버, P2P 분산 아키텍처, 마이크로 서비스, 웹 애플리케이션의 데이터 흐름 기본 사항, 관련 계층, 확장성, 고 가용성 등의 개념을 같은 다양한 아키텍처 스타일에 대해 배워야 합니다.

올바른 아키텍처와 기술 스택을 선택하여 사용 사례를 구현하는 기술을 경험한다.

소프트웨어 개발 분야에서 경력을 쌓기 시작한 초보자라면 How to design a web application: software architecture 101 과정을 권장합니다.