Design Concepts 조사

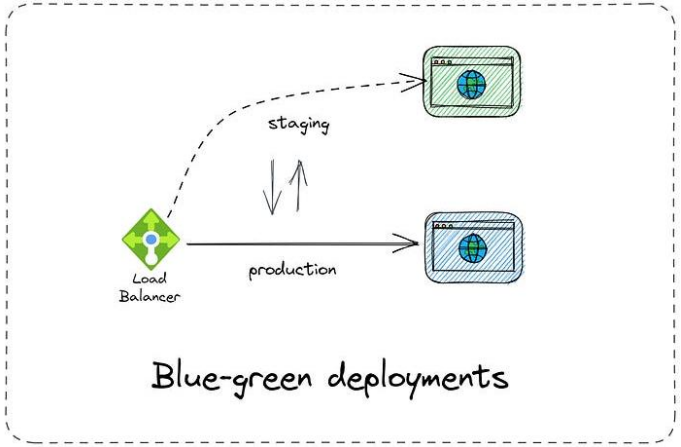
허윤아
한범진
위기화
진항우

Deployment Pattern

Blue-green Deployment Pattern

정의: 동일한 어플리케이션을 두 개의 서로 다른 환경에서 운영하여, 하나의 환경에서만 트래픽을 처리하고 나머지 환경은 새로운 버전을 배포하는 방식.

- Blue 환경: 현재 어플리케이션이 운영 중인 환경. 기존 버전의 어플리케이션이 배포된 상태이며, 모든 트래픽이 이 환경으로 라우팅되고 있음.
- Green 환경: 새 버전의 어플리케이션이 배포될 환경. Blue 환경과 동일한 인프라 구성으로 설정되지만, 새 버전이 배포되어 테스트와 검증이 진행됨.



과정

1. Green 환경에 새 버전 배포: green 환경에 새 버전의 어플리케이션 배포. Blue 환경은 계속해서 기존 버전으로 트래픽을 처리하고 있는 상태.
2. Green 환경 테스트: 새 버전의 기능 및 성능 테스트를 green 환경에서 수행.
3. 트래픽 전환: 모든 트래픽을 blue에서 green으로 전환. 로드 밸런서 또는 DNS 설정 변경으로 수행됨.
4. 롤백 준비: 새 버전에 문제가 발생하면 트래픽을 다시 blue환경으로 되돌릴 수 있음 (신속한 롤백 → 운영 중단 최소화)

장점

- 무중단 배포
- 빠른 롤백
- 검증된 배포
- 테스트 환경 재사용 (green → blue)

단점

- 많은 리소스 필요 (비용 증가)
- 데이터 동기화 문제
- 운영 복잡성

A/B Testing Pattern

정의: 사용자에게 두 가지 이상의 변형된 버전을 노출하여 어떤 버전이 더 높은 성과를 내는지 평가하는 방법

특징

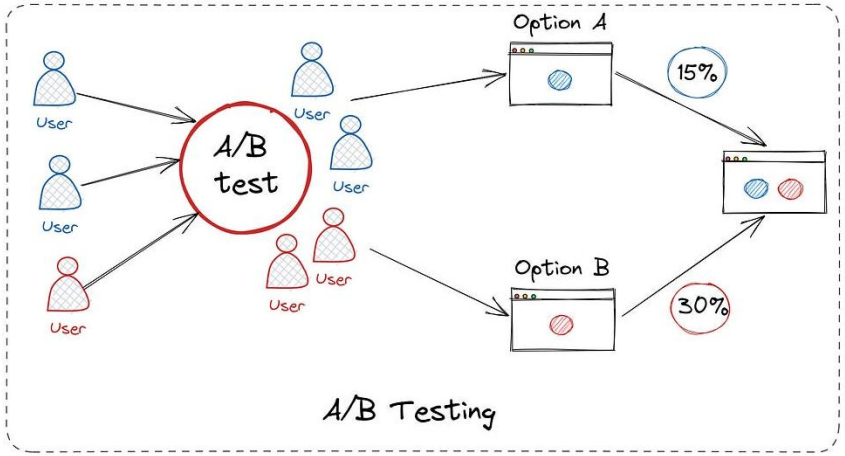
- 사용자를 무작위로 나누어 각 그룹에 서로 다른 버전을 제공하고, 테스트의 목적을 명확히 설정한 후 각 버전이 이 목표를 얼마나 잘 달성하는지 평가함
- 실험 데이터에 통계 분석을 적용하여 각 버전의 성과를 비교하고, 유의미한 차이가 있으면 더 나은 버전을 선택함.

장점

- 데이터 기반 의사결정
- 빠른 개선
- 위험 감소

단점

- 데이터 신뢰성 문제
- 실행 비용
- 시간 소모
- 사용자 경험에 혼란을 초래할 수 있음



Client-Server Pattern

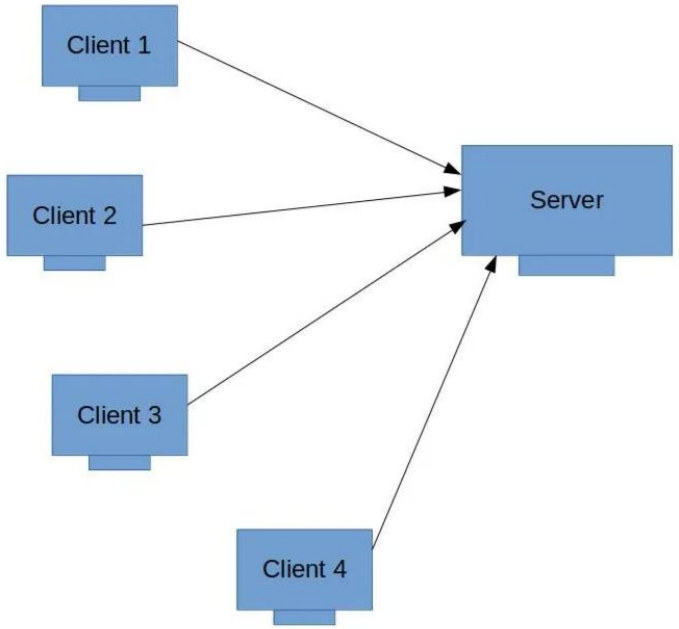
정의: 클라이언트와 서버 간의 상호작용을 기반으로 한 구조
클라이언트는 서비스나 리소스를 요청하는 주체, 서버는 이러한 요청을 처리하고 응답하는 역할을 수행

장점

- 확장성이 뛰어남
- 데이터 일관성과 무결성 유지가 쉬움
- 유지보수가 용이함.
- 일관된 접근 제어와 보안 관리 가능

단점

- 서버가 다운되면 모든 클라이언트가 영향을 받음
- 클라이언트 수가 많아질수록 서버 부하가 커지며 성능 저하가 발생할 수 있음
- 서버 성능과 네트워크 대역폭에 따라 확장 한계가 존재할 수 있음



구성요소: 클라이언트 (데이터 요청), 서버 (데이터 처리 및 응답)

Master-Slave Pattern

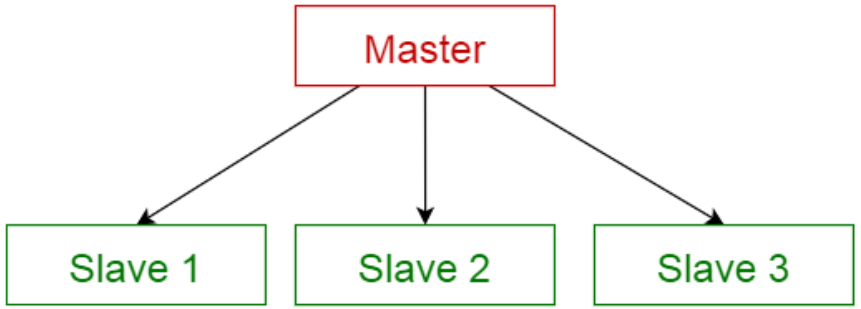
정의: 마스터가 작업을 여러 슬레이브에 분배하고, 슬레이브가 이를 처리한 후 마스터에게 결과를 반환하는 구조.

장점

- 작업을 분산 처리함으로써 성능을 향상시킬 수 있음.
- 마스터가 결과를 통합하기 때문에 데이터 일관성 유지가 쉬움.
- 슬레이브를 추가해 확장성 확보 가능.

단점

- 마스터에 과부하가 발생할 수 있음.
- 마스터가 다운되면 전체 시스템이 중단될 수 있음.
- 슬레이브 간 데이터 동기화가 필요할 때 관리가 복잡해짐.



구성요소: 마스터[Master](작업을 분배, 결과 수집), 슬레이브[Slave](작업 할당받아 처리, 반환)

Load-Balanced Cluster

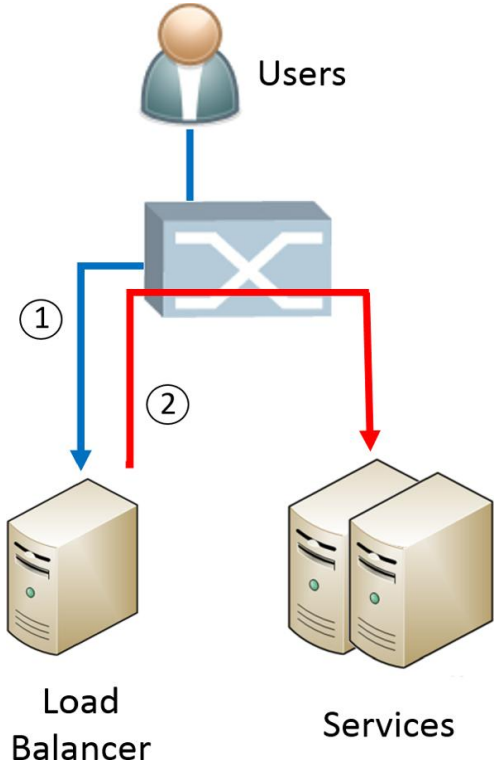
정의: 여러 노드에 걸쳐 트래픽을 균등하게 분산하는 방식.
로드 밸런서가 사용자 요청을 각 노드에 분산하며, 노드들은 할당받은 작업을 개별적으로 처리.

장점

- 트래픽이 균등하게 분산되어 성능이 향상됨.
- 장애가 발생해도 남은 노드가 요청을 처리해 가용성이 높음.
- 추가 노드를 쉽게 연결할 수 있어 확장성이 뛰어남.

단점

- 로드 밸런서가 단일 실패 지점으로 작용할 수 있음.
- 각 노드의 상태를 지속적으로 모니터링하고 관리해야 함.



구성요소: 로드 밸런서[Load Balancer](트래픽 분산, 요청 할당), 노드[Node](작업 수행, 처리 결과 반환)

Architecture Style

Event-driven architecture (EDA) style

구조: 비동기 처리에 기반하여 서로 강하게 결합되지 않은 이벤트 프로세서를 사용해 시스템 내 이벤트를 발생시키고 이에 반응하는 구조

고려되는 상황

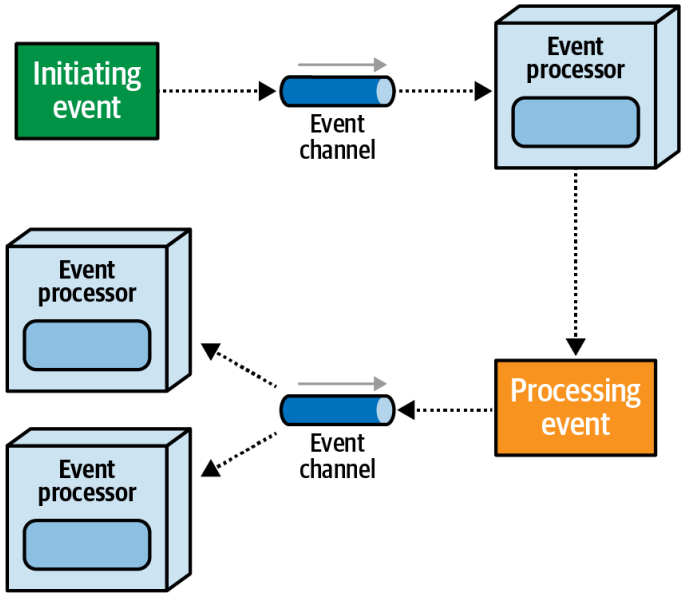
- 복잡한 비결정적 워크플로우가 있거나 반응성이 중요한 경우
- 비즈니스의 흐름이 트리거나 이벤트에 따라 작동되는 경우
- 결정 트리를 만들기 어려운 복잡한 워크플로우를 가진 경우

장점

- 고성능
- 고확장성
- 높은 내결함성

단점

- 오류 처리나 복구가 어려움
- 각 서비스가 자체적으로 오류를 처리해야 함
- 구조가 단순하지 않으므로 많은 개발 비용 발생



Space-based architecture style

시스템의 transaction 처리에서 DB를 제거하여 어플리케이션 확장의 한계를 해결함
(공유 메모리가 있는 여러 병렬 프로세서의 개념인 컴퓨터 과학 용어 tuple space에서 비롯된 이름)

고려되는 상황

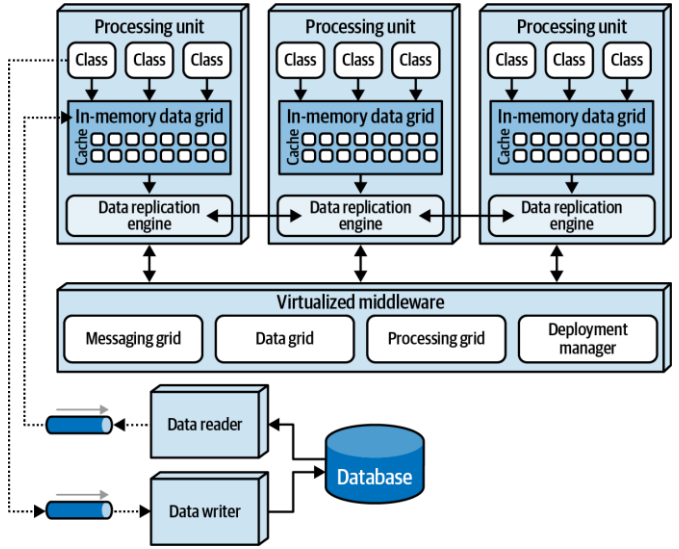
- 동시 확장성 또는 탄력성 요구사항이 매우 높은 상황에 적합함
- 높은 성능과 반응성을 필요로 하는 경우

장점

- 높은 확장성
- DB의 병목현상 제거

단점

- 기술적 복잡도가 높음
- 업데이트 이전에 매우 많은 시간을 필요로 함



Processing unit (서비스):

- 비즈니스 기능을 포함하며, 단일 목적 기능부터 전체 어플리케이션 기능까지 세분성 (granularity)이 다양함
- 비즈니스 로직, transaction 데이터를 포함하는 인메모리 데이터 그리드, (선택적으로) 웹 기반 구성요소 포함
- 가상화된 미들웨어의 프로세싱 그리드를 통해 직접/서로 통신 가능

Microservices Architecture

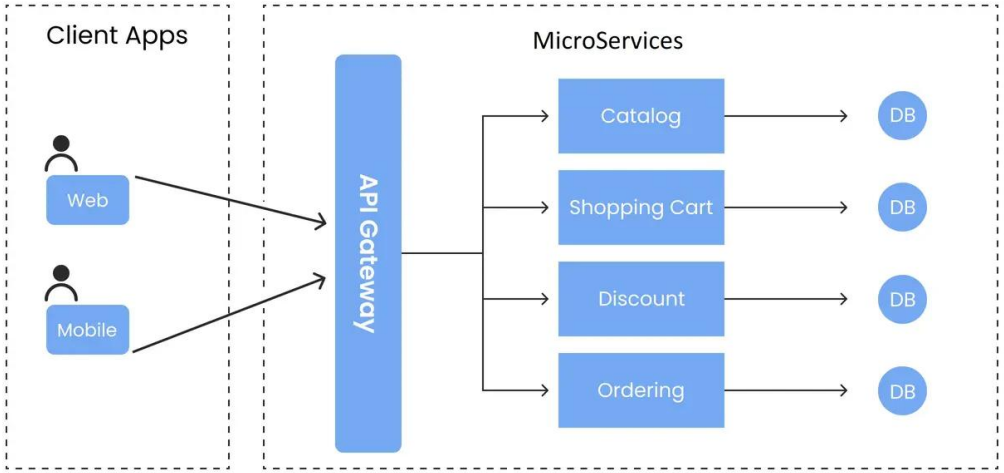
정의: 애플리케이션을 작은 독립적 서비스들의 집합으로 구성하는 아키텍처.
각 마이크로서비스는 특정 기능을 담당하며 독립적으로 배포, 관리 가능.
서비스들은 API 게이트웨이를 통해 통신한다.

장점

- 개발과 확장이 용이
- 특정 서비스만 수정할 수 있어, 빠른 업데이트가 가능
- 특정 기능에 대한 장애가 발생해도 다른 서비스에는 영향을 주지 않아 가용성이 높음

단점

- 서비스가 많아지면 각 서비스 간 통신과 데이터 관리가 복잡해짐
- 모니터링과 배포 관리에 추가적인 노력이 필요
- 서비스 간 데이터 일관성을 유지하기 위해 트랜잭션 관리가 복잡해질 수 있음



구성요소: 마이크로서비스[Microservice](독립적 서비스, 특정 기능 수행), API 게이트웨이[API Gateway](요청 라우팅, 서비스 간 통신 관리)

Layered Architecture

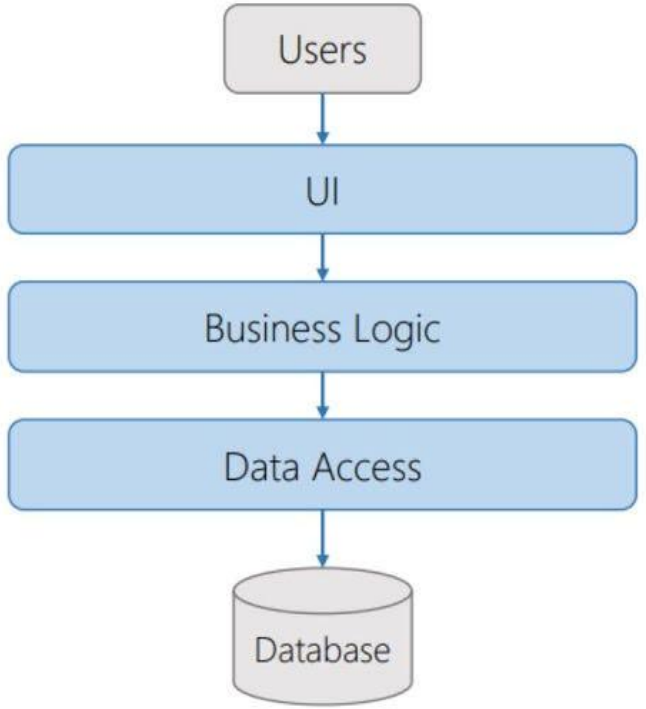
정의: 애플리케이션을 기능별로 계층화하여 각 계층이 특정 역할을 담당하도록 분리하는 아키텍처. 일반적으로 사용자 인터페이스, 비즈니스 로직, 데이터 접근 등의 계층으로 나뉘며, 각 계층은 하위 계층의 서비스를 호출해 작업을 수행한다.

장점

- 코드가 이해하기 쉽고 유지보수가 용이.
- 특정 계층의 수정이 다른 계층에 영향을 주지 않아 시스템 안정성이 높음.
- 계층별 테스트가 가능해 개발과 디버깅이 편리.

단점

- 각 계층을 거쳐야 하므로 요청 처리에 시간이 걸릴 수 있음.
- 각 계층 간의 의존성으로 인해 확장성에 제한이 생길 수 있음.



구성요소:

프레젠테이션 계층[Presentation Layer](사용자 인터페이스),

비즈니스 계층[Business Layer](로직 처리),

데이터 접근 계층[Data Access Layer](데이터베이스와의 상호작용)

Ports and Adapters

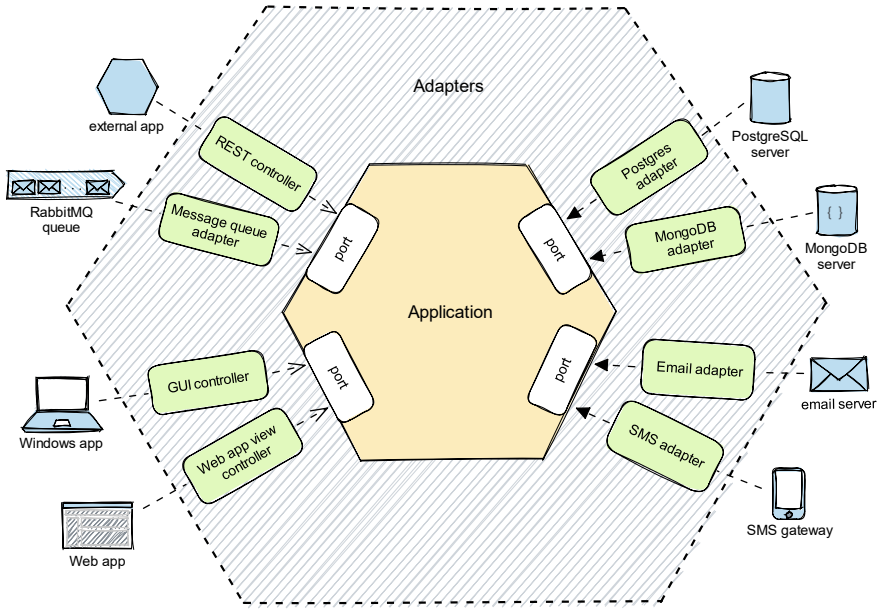
정의: 시스템의 핵심 로직을 외부 인터페이스로부터 분리하여 독립적으로 설계하는 아키텍처. 포트가 시스템과 외부 간의 인터페이스 역할을 하며, 어댑터가 포트를 통해 외부 요청을 처리하고 필요한 시스템 연결을 수행.

장점

- 핵심 비즈니스 로직이 외부 시스템으로부터 분리되어 변경에 유연함.
- 외부 의존성의 교체가 쉬움.
- 시스템을 다양한 인터페이스와 연결할 수 있어 확장성이 높음.

단점

- 포트와 어댑터를 설계하고 구현해야 하므로 복잡성이 증가할 수 있음.



구성요소: 포트[Ports](시스템과 외부 간 인터페이스 정의), 어댑터[Adapters](포트를 구현하여 외부 요청 처리, 시스템 연결)

Tactics

Performance Efficiency
(Time Behavior)

Cache Tactics

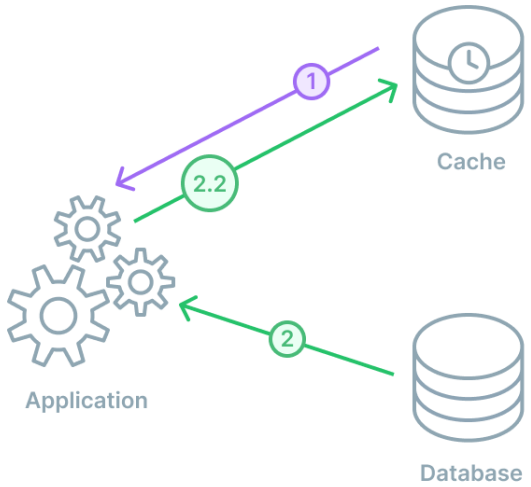
정의: 특정 데이터나 연산의 결과를 **임시 저장소**에 저장하여, 동일한 요청이 반복될 때 원본 소스로부터 데이터를 가져오는 대신 캐시된 데이터를 재사용하도록 함

장점

- 캐시에서 데이터를 가져오므로 요청 응답 시간 단축
- 원본 소스에 대한 요청의 빈도를 줄임으로써 서버 부하 감소
- 데이터베이스 접근 비용 절감

단점

- 데이터 일관성 문제
- 추가 자원 사용으로 인한 비용 발생
- 적절한 TTL (Time-to-Live) 설정의 어려움



Prioritization

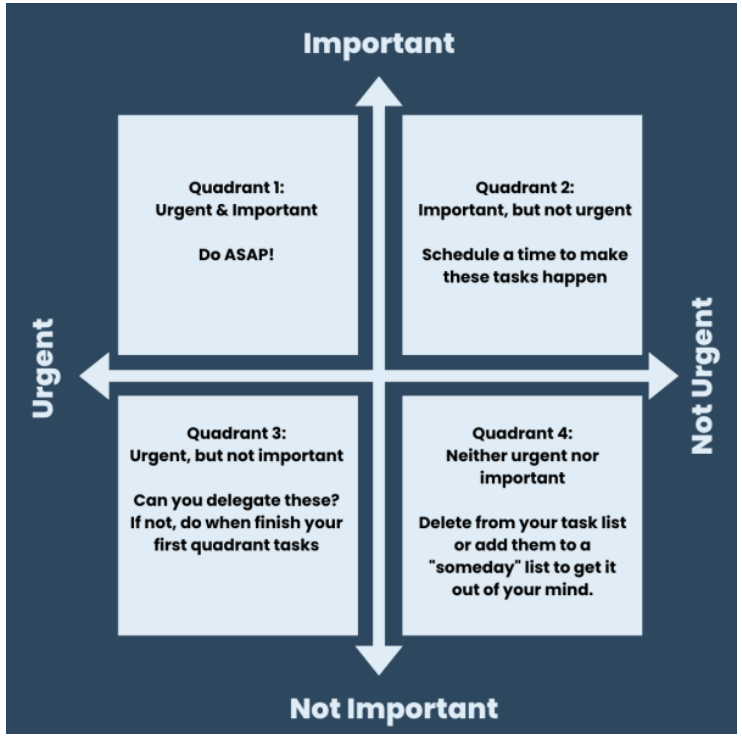
정의: 중요한 작업을 먼저 처리하도록 우선순위를 지정해 주요 작업이 빠르게 처리되도록 하는 전략
실시간 요청이 중요한 시스템에서는 실시간 작업에 높은 우선순위를 주고, 비실시간 작업은 후순위로 둠

장점

- 실시간성 확보 가능
- 자원 최적화

단점

- 비중요 작업이 계속 지연되어 성능 불균형이 발생할 수 있음
- 복잡한 자원의 관리
- 시스템 불안정성이 커질 수 있음



Batch Processing

정의: 일정량의 데이터를 모아서 한 번에 처리하는 방식

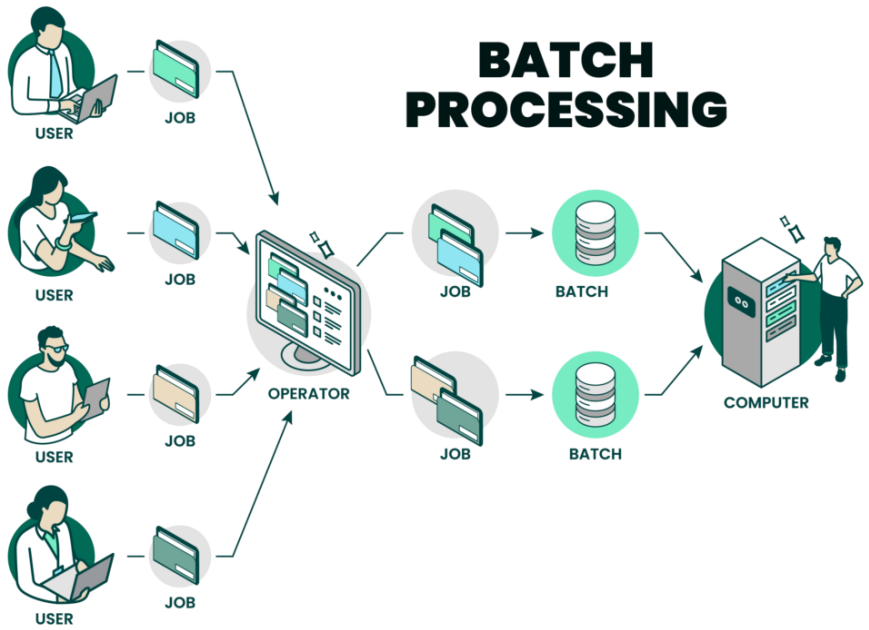
주로 시스템 자원을 효율적으로 사용하거나 대규모 데이터를 처리할 때 사용되며, 실시간이 아닌 특정 시간이나 조건에 맞춰 수행

장점

- 대규모 데이터 처리 시 시스템 자원을 효율적으로 사용할 수 있음.
- 실시간 응답이 필요하지 않은 작업들을 모아 한 번에 처리하여 성능을 최적화할 수 있음.

단점

- 실시간 처리가 필요한 상황에 부적합.
- 오류 발생 시 전체 작업을 재실행해야 할 수 있어 복구 시간이 길어질 수 있음.



Maintainability
(Modifiability)

Repository Pattern

정의: 데이터 접근을 추상화하여 비즈니스 로직과 데이터베이스 간의 결합을 줄이는 전술

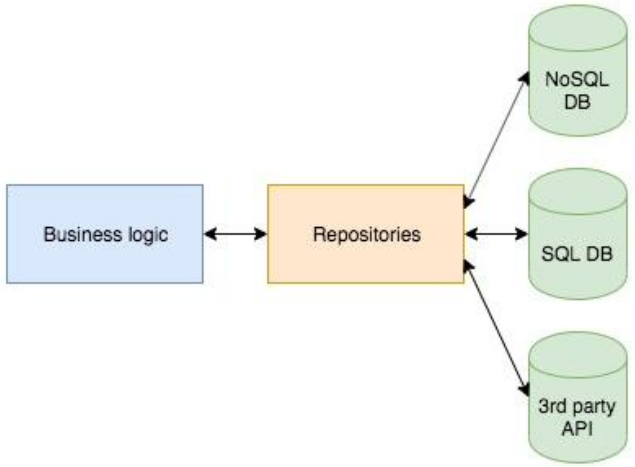
레포지토리는 데이터 소스에 대한 직접적인 호출 없이 데이터의 저장, 검색, 업데이트 등을 관리하며, 이를 통해 코드에서 데이터 접근을 쉽게 테스트하고 교체할 수 있음

장점

- 데이터 접근 로직이 응용 프로그램 코드와 분리되어 유지보수가 용이
- 데이터 소스 변경이 필요할 때 비즈니스 로직을 수정하지 않고 레포지토리만 수정하면 됨

단점

- 모든 데이터 접근을 추상화하면 초기 개발이 복잡하고 시간이 소요될 수 있음



Modularity

정의: 시스템을 기능별로 분리해 독립된 모듈로 나누는 전술

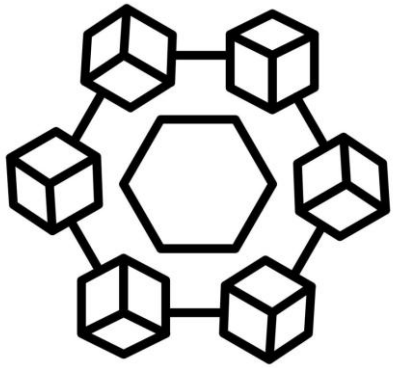
각 모듈은 특정 기능을 수행하며, 다른 모듈과 상호작용할 수 있도록 설계됨

장점

- 각 모듈이 독립적이므로 특정 기능을 변경할 때 다른 모듈에 영향을 미치지 않아 변경이 용이함
- 특정 기능을 수행하는 모듈은 다른 시스템이나 프로젝트에서 재사용이 가능함
- 문제 발생 시 해당 모듈만 수정하면 되므로 유지보수성이 향상됨
- 개발자들이 독립적인 모듈에서 작업할 수 있어 협업에 용이함

단점

- 모듈화가 오히려 시스템 복잡성을 증가시키므로 적절한 인터페이스의 관리가 필요함
- 각 모듈의 기능과 경계를 명확히 정의해야 하므로 초기 설계 단계에서 추가적인 시간이 소요될 수 있음
- 모듈의 유지보수와 변경 이력 관리를 위해 별도의 관리가 필요함
- 모듈 간의 의존성이 높아질 경우 모듈화의 이점이 줄어들고, 변경이 용이하지 않은 구조가 될 수 있음



Versioning

정의: 시스템의 구성 요소나 API, 데이터베이스 스키마 등에 버전을 부여하여, 변경 사항을 추적하고 관리하는 방식

장점

- 이전 버전을 기록하고 필요할 때 언제든지 롤백할 수 있어 수정 및 업데이트를 안전하게 수행할 수 있음
- 새로운 기능을 추가해도 기존 사용자나 클라이언트가 기존 버전을 사용할 수 있어 호환성을 유지할 수 있음

단점

- 여러 버전을 동시에 관리해야 하므로 시스템이 복잡해지고, 코드 베이스나 데이터베이스가 관리하기 어려워질 수 있음
- 새 버전과 기존 버전 간의 호환성 문제가 발생할 수 있음

Reliability
(Recoverability)

Retry on Failure

정의: 작업 실패 시 자동으로 일정 횟수 재시도하는 전술

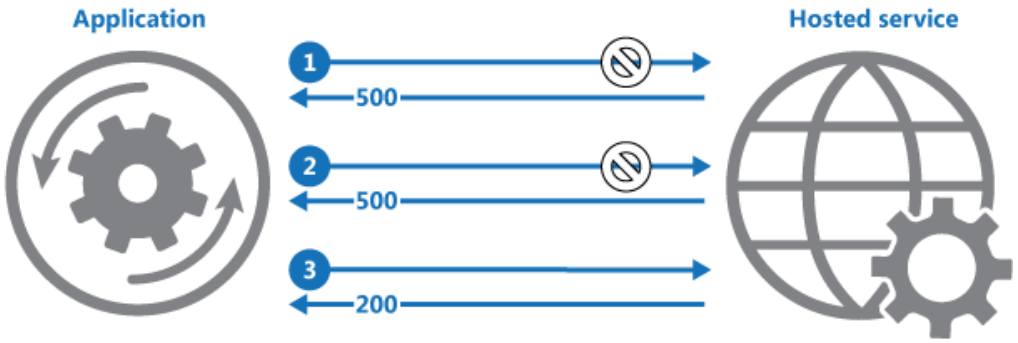
네트워크 요청이나 외부 API 호출 등 일시적 장애가 발생할 가능성이 있는 작업에 적합. 특히, 안정적인 응답을 보장해야 하는 시스템에서 유용

장점

- 일시적인 오류로 인한 실패를 최소화할 수 있음.
- 작업 안정성이 향상되어 사용자 경험을 개선.

단점

- 재시도가 계속 실패할 경우 시스템 자원이 낭비될 수 있음.
- 재시도 횟수가 많아지면 응답 지연이 발생할 수 있음.



Rollback

정의: 오류가 발생했을 때 시스템을 이전의 안정적인 상태로 되돌리는 전술

데이터 손상이나 트랜잭션 실패 시 마지막으로 성공한 상태로 돌아가 시스템의 안정성을 유지함

장점

- 잘못된 데이터나 실패한 트랜잭션으로부터 시스템을 안전하게 보호할 수 있음
- 마지막 성공 상태로 복구하기 때문에 빠른 복구 가능

단점

- 롤백 시 복구된 이후의 작업은 모두 무효화되므로 최근 작업이 모두 손실될 수 있음
- 빈번한 롤백은 시스템의 가용성과 신뢰성에 악영향을 줄 수 있음

Checkpointing

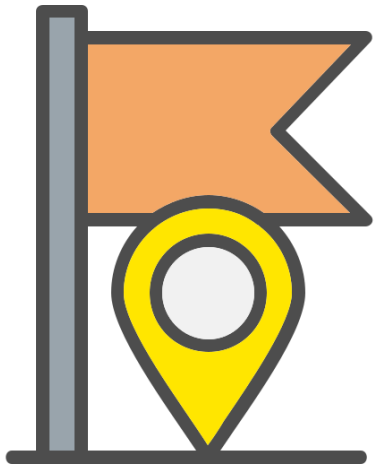
정의: 시스템 상태를 특정 지점 (checkpoint)에서 저장해 두는 전술

장점

- 장애 시 마지막 체크포인트로 빠르게 복구할 수 있어 시스템의 가용성이 향상됨
- 마지막 체크포인트 이후의 작업만 다시 수행하면 되므로 데이터 손실을 줄일 수 있음

단점

- 체크포인트 저장에 있어서 오버헤드가 발생함
- 체크포인트를 자주 저장하면 추가적인 연산과 저장공간을 필요로 함
- 장애 발생 시 최근 체크포인트 이후의 데이터는 손실될 수 있음
  (데이터 손실 가능성의 완전한 제거는 불가능)



Graceful Degradation

정의: 시스템에 문제가 발생하더라도 전체 시스템의 가동 중단을 방지하고 핵심 기능은 유지하면서 성능이나 기능을 제한적으로 제공하는 전략

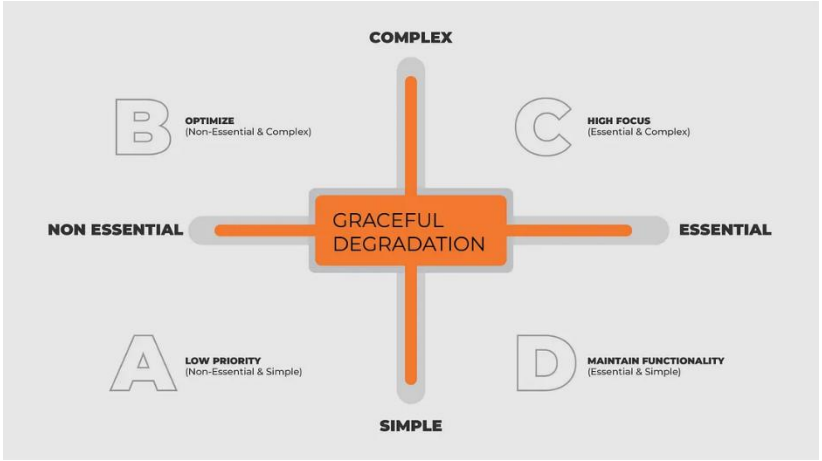
예기치 못한 상황에서도 최소한의 기능을 제공하여 사용자 경험에 미치는 영향을 최소화하고 시스템의 신뢰성을 높이려는 접근 방식

장점

- 장애가 발생하더라도 시스템이 완전히 중단되지 않고 핵심 기능을 유지하므로, 사용자는 서비스에 대한 신뢰성을 유지할 수 있음
- 전체 시스템 가동 중단을 방지하고 주요 기능을 지속적으로 제공함으로써 사용자 불편을 최소화할 수 있음
- 여러 기능 계층을 유동적으로 관리할 수 있기 때문에 트래픽이 증가하더라도 핵심 기능의 가동을 우선시할 수 있음

단점

- 각 기능을 중요도에 따라 설계하고 우선순위를 정하는 복잡한 설계 작업이 필요함
- 일부 기능이 제한되거나 저하된 상태에서 시스템을 운영하게 되면 사용자 경험이 저하될 수 있음
- 기능이 단계적으로 제한되는 시나리오가 다양하므로, 각 단계의 상황을 고려한 테스트와 유지 보수가 필요함



Usability
(Operability)

Configuration Management

정의: 시스템의 설정 및 구성 요소를 관리하고 변경이 발생했을 때 이를 체계적으로 반영하는 전술

운영자가 쉽게 설정을 변경하거나 시스템 환경에 맞게 조정할 수 있음

장점

- 시스템 설정을 쉽게 변경할 수 있어 운영 효율성이 높아짐
- 일관된 구성 관리가 가능하여 시스템의 안정성을 보장할 수 있음

단점

- 구성 관리에 필요한 추가적인 소프트웨어 및 인프라가 필요할 수 있음
- 설정 변경으로 인해 예상치 못한 부작용이 발생할 수 있음

Security
(Accountability)

Audit Trail

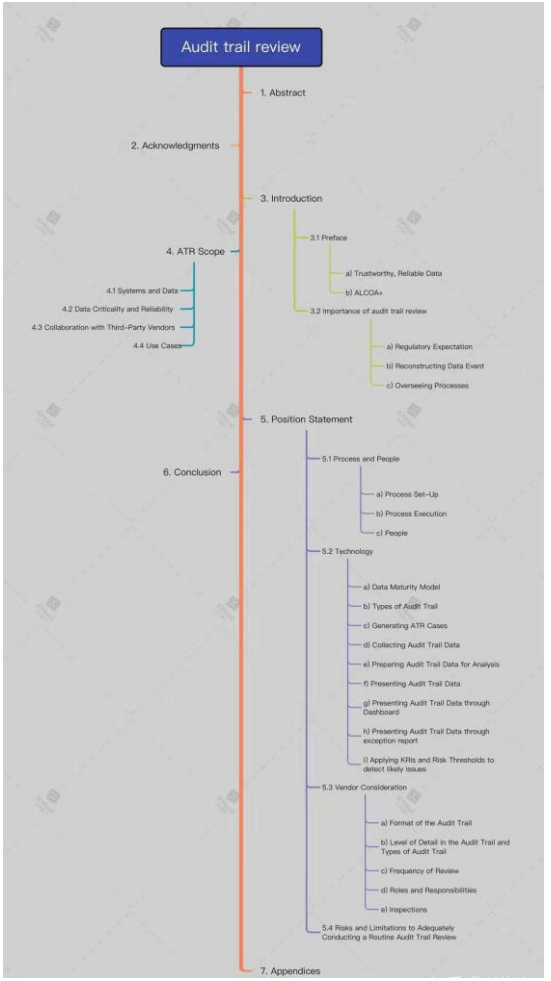
•**정의**: 사용자의 활동이나 시스템 이벤트를 기록하여, 보안 문제 발생 시 이를 추적하고 감사할 수 있도록 함

장점

- 사용자의 활동을 기록하여 불법적이거나 비정상적인 접근을 추적할 수 있음
- 누가 어떤 작업을 수행했는지 명확히 함으로써 책임 소재를 분명히 할 수 있음

단점

- 추가적인 메모리 및 스토리지 리소스 필요
- 과도한 로깅으로 인한 성능 저하 가능



감사합니다☺