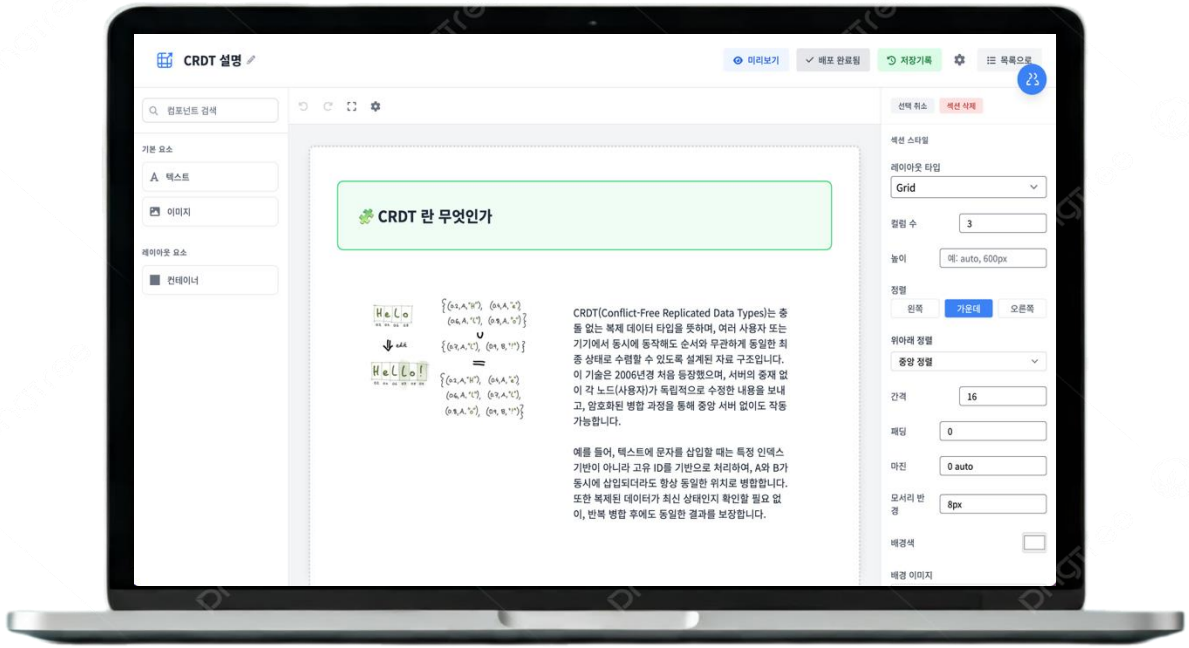
Easy to Web | 손쉽게 홈페이지를 게시하는 Web Builder 서비스

개발 기간 2025.01 ~2025.05

개발 인원 2명

기술 스택
- Java 21, SpringBoot 3.4.1
- TypeScript, React
- PostgreSQL, Redis
- CloudFlare, Oracle Cloud

기타 도구
- Notion
- Slack
- Intellij

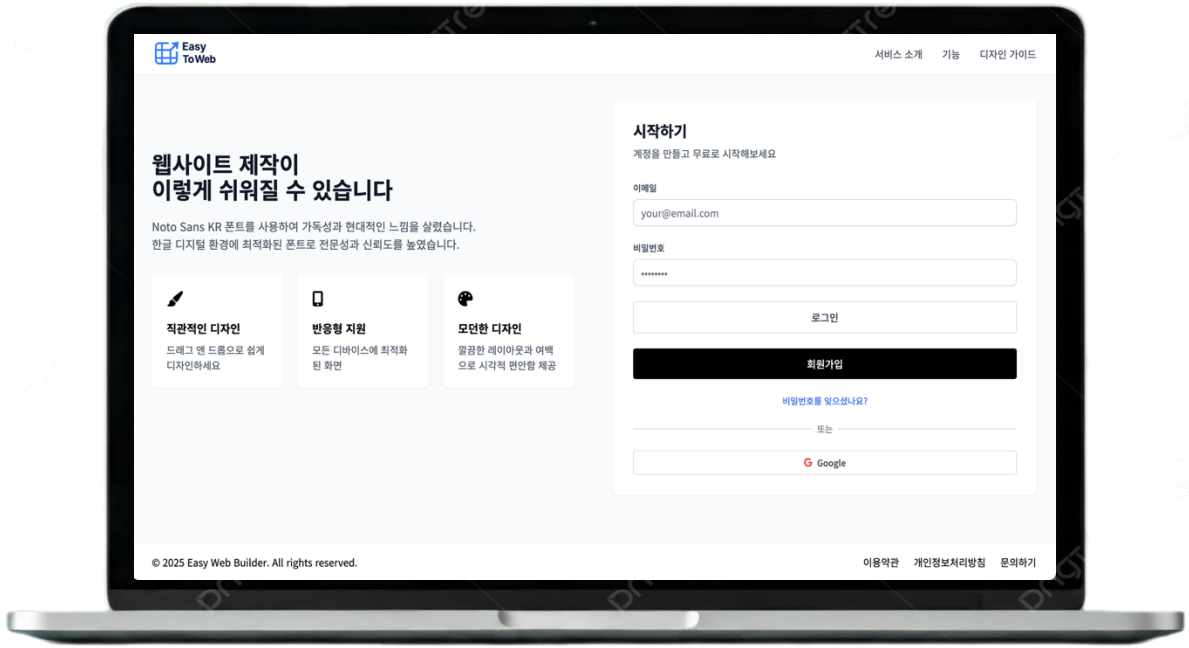


결과물

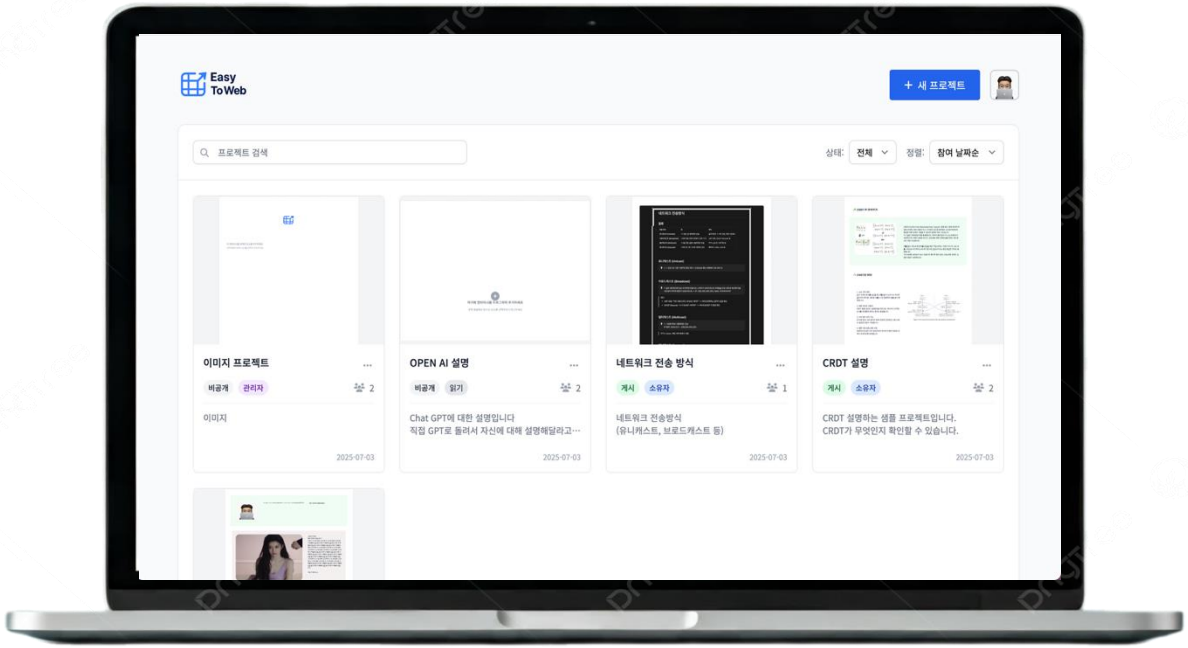
GitHub	https://github.com/rodminjo/Easy-To-Web
서비스 (운영)	https://easytoweb.store/
서비스 (개발)	https://dev.easytoweb.store/
개발 문서	Notion 개발 문서
API 문서 (Swagger)	https://dev-api.easytoweb.store/swagger/swagger-ui/index.html
ERD	ERD

목차

Easy to Web | 주요 기능 요약

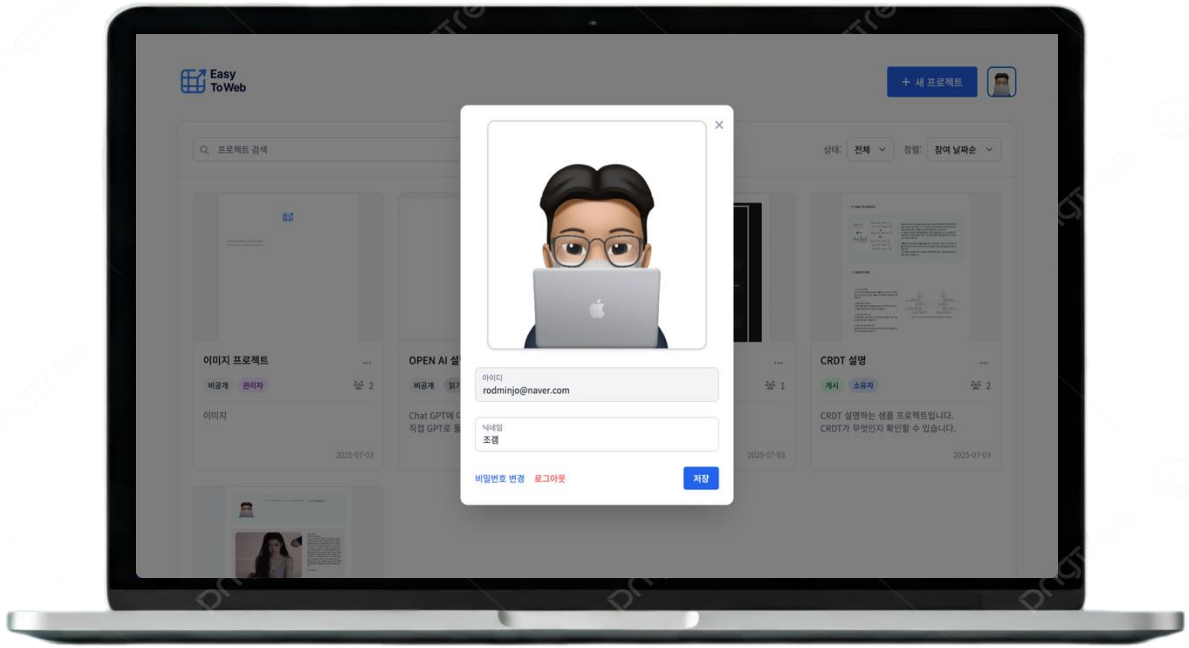


로그인

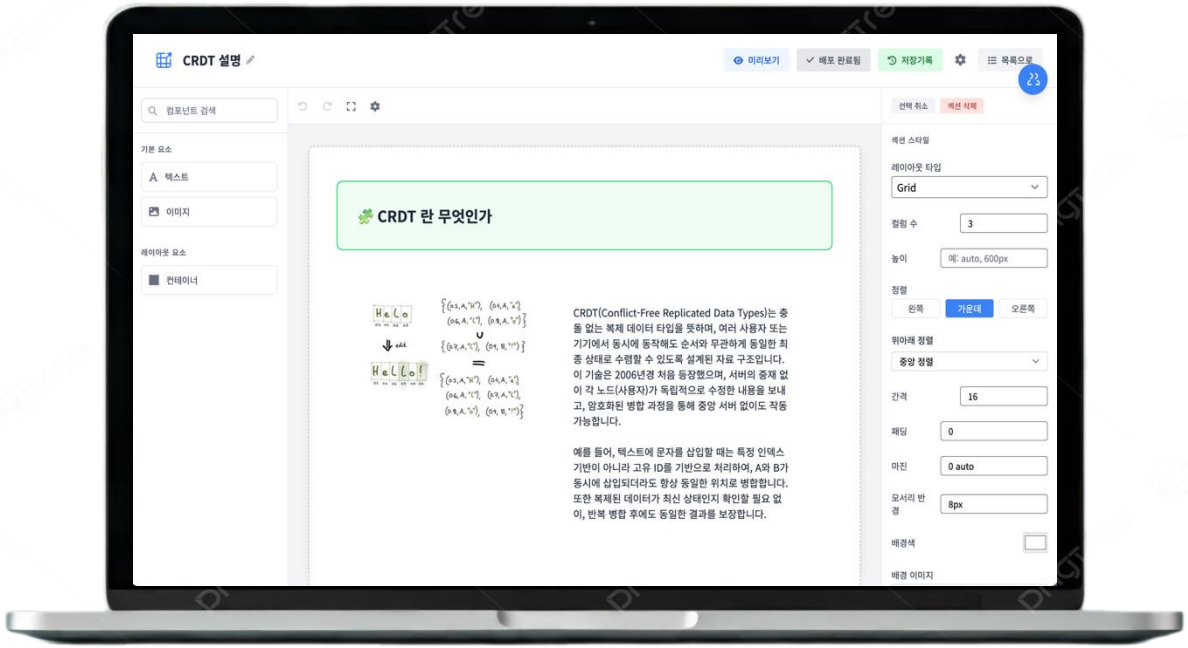


프로젝트 목록

Easy to Web | 주요 기능 요약

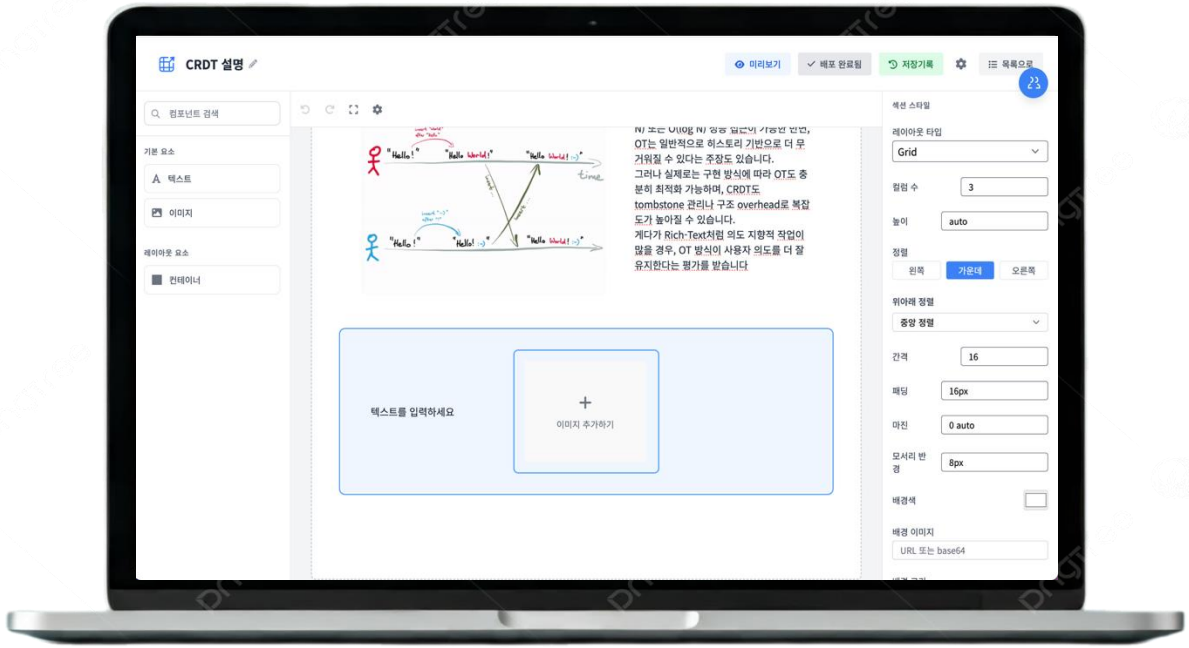


프로필 편집

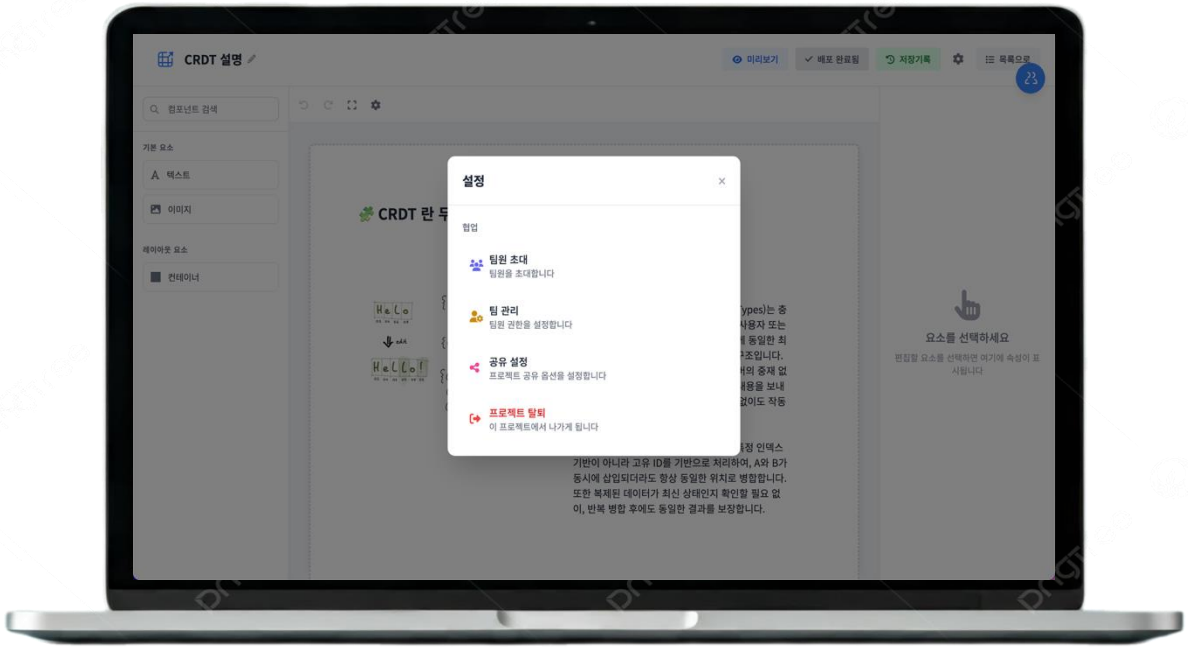


프로젝트 편집

Easy to Web | 주요 기능 요약

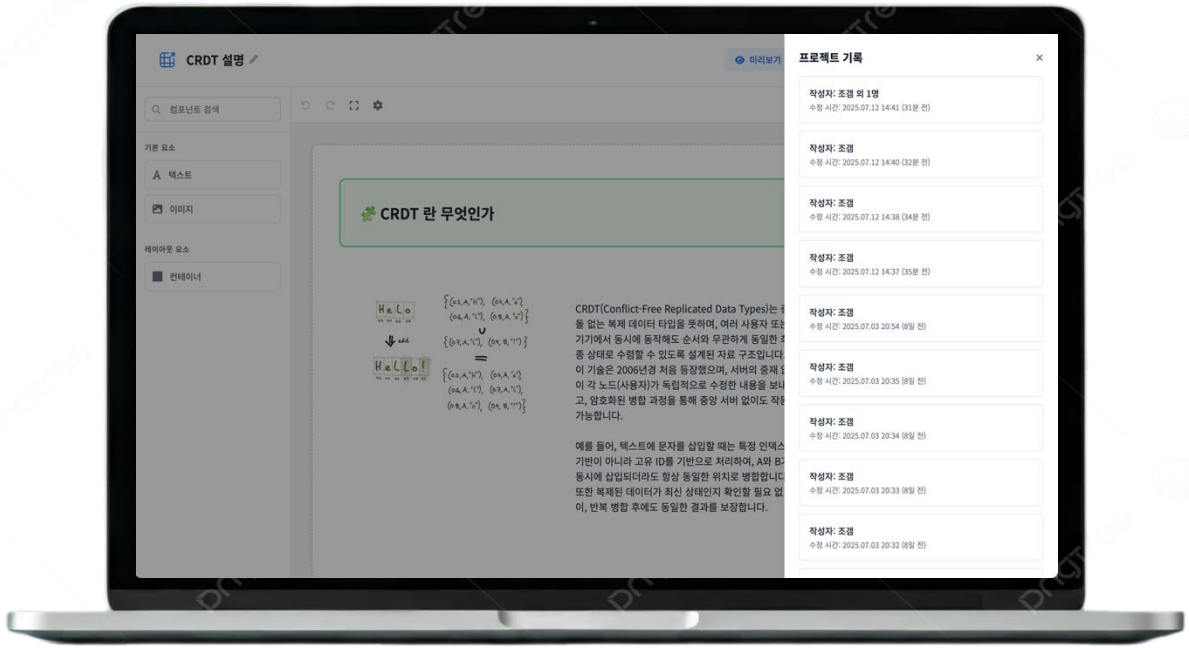


프로젝트 요소 편집

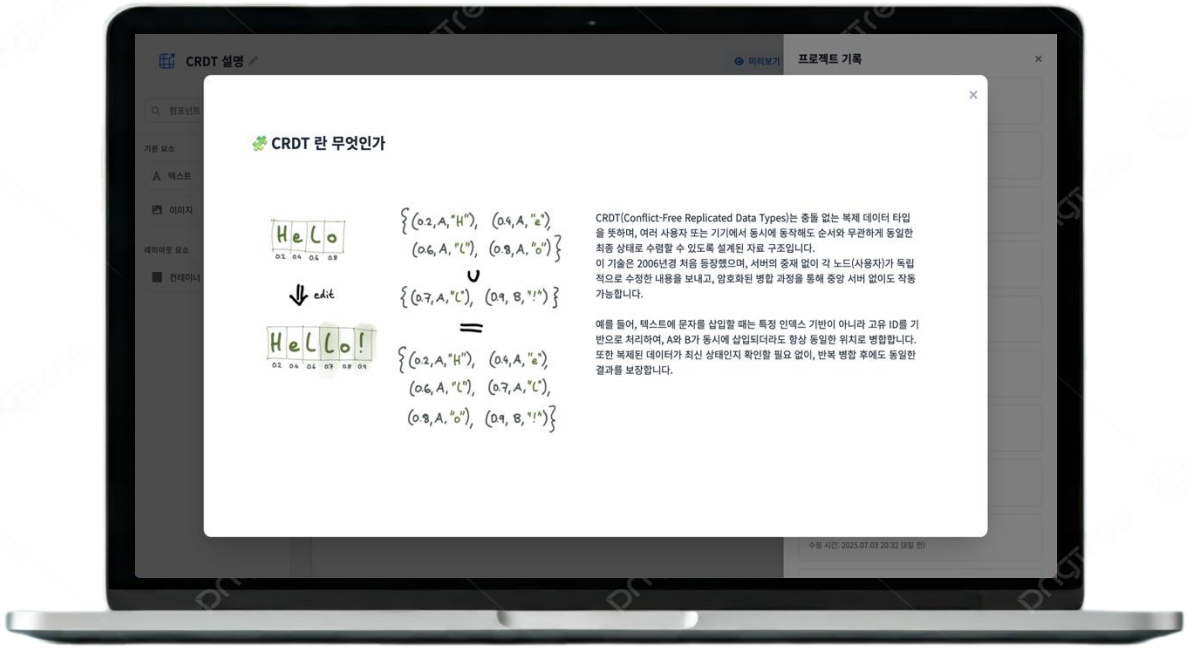


프로젝트 관리

Easy to Web | 주요 기능 요약

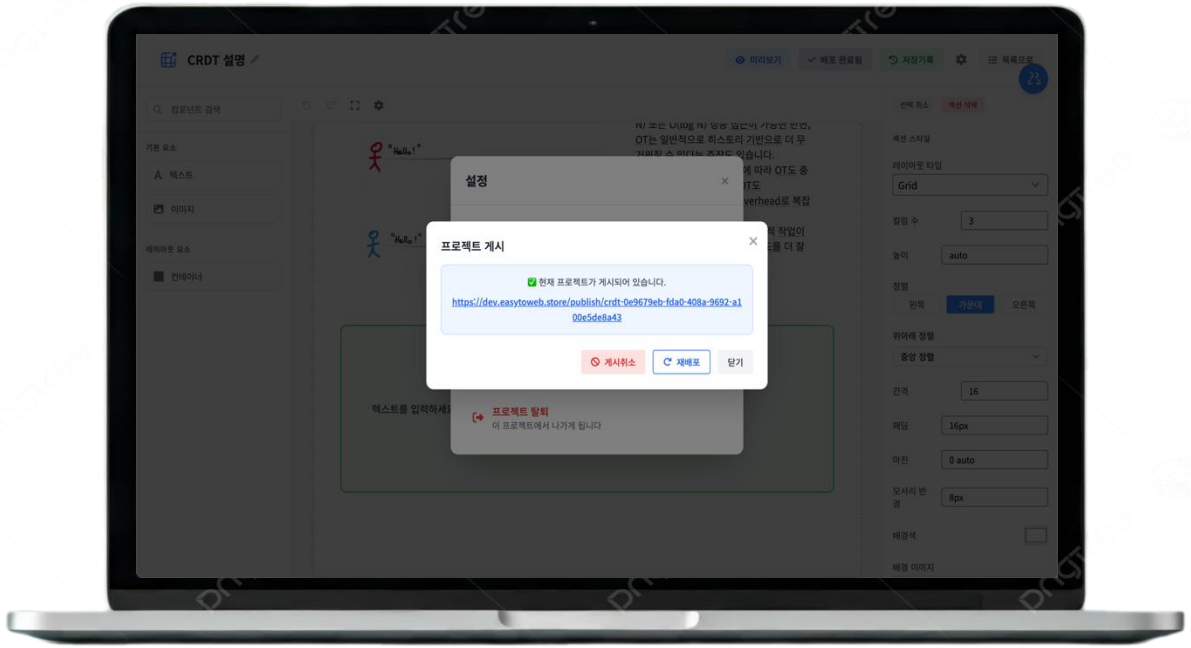


히스토리 조회

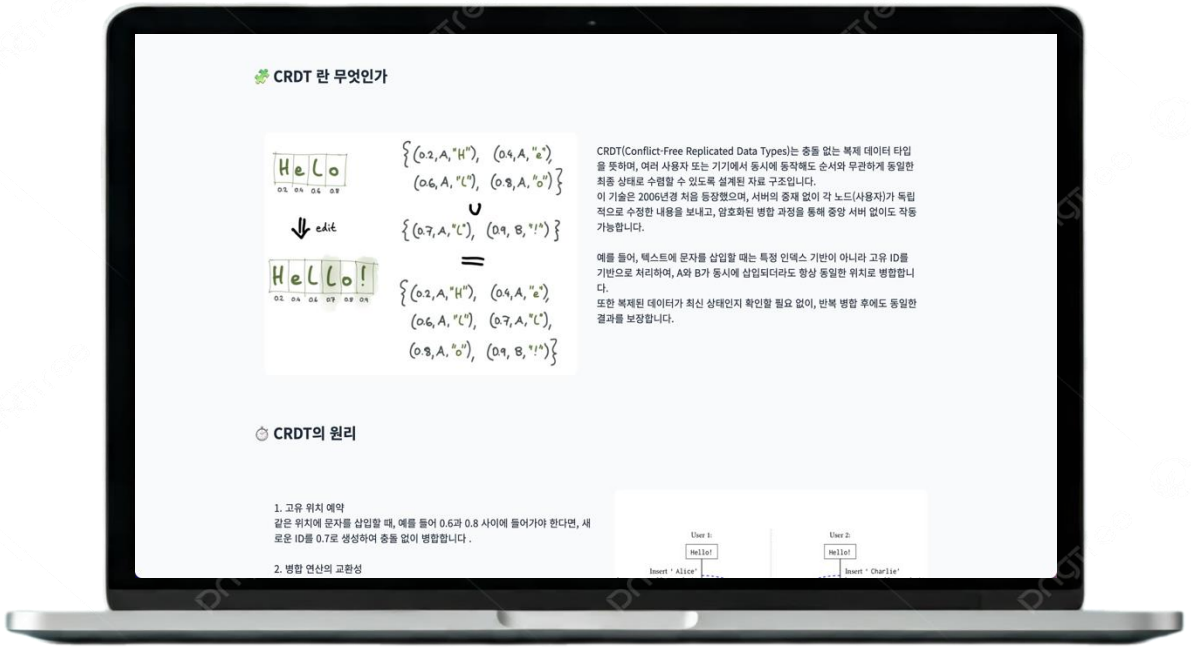


편집 화면

Easy to Web | 주요 기능 요약



프로젝트 게시 관리



프로젝트 게시

목차

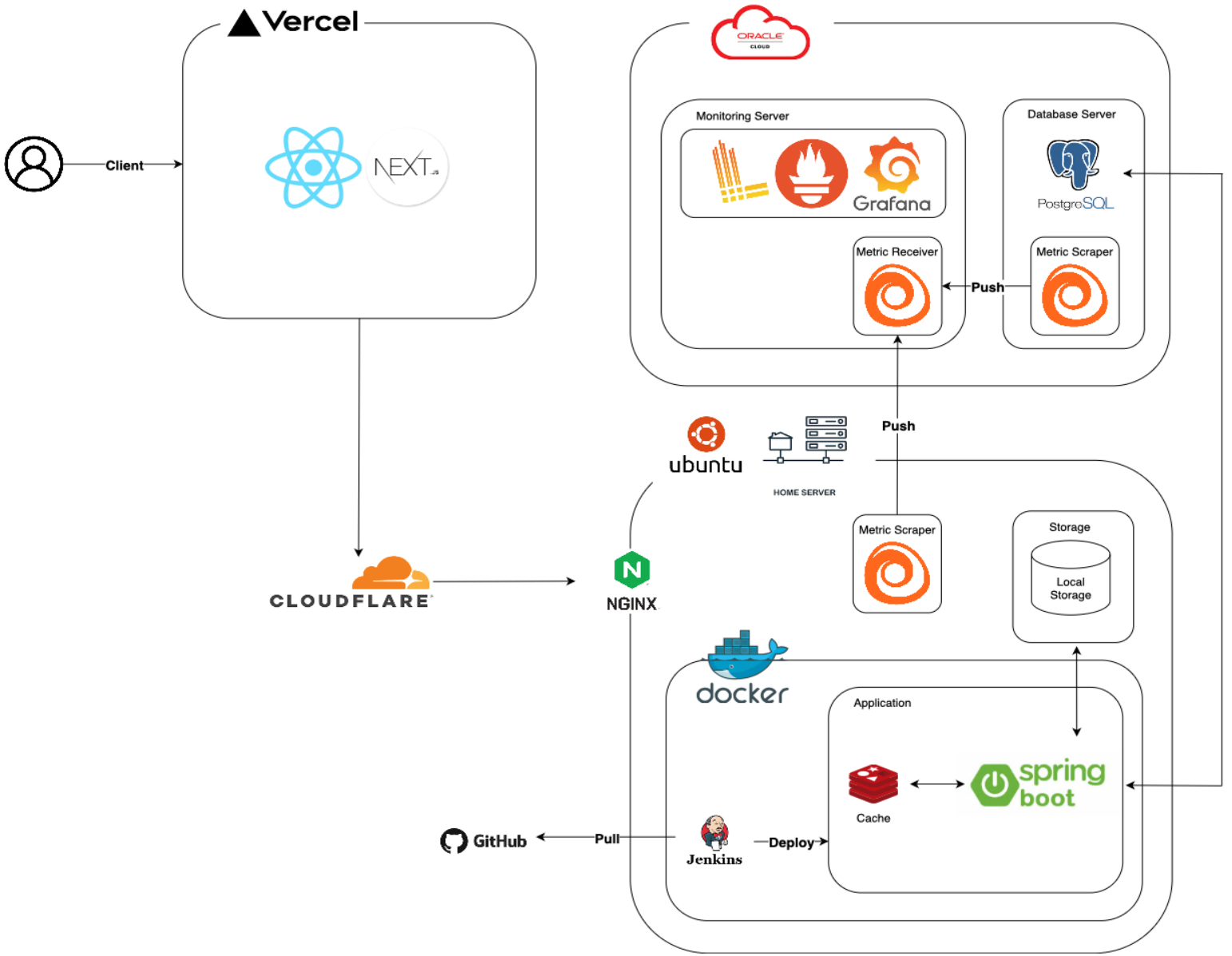
Easy to Web | 프로젝트 아키텍처

관련 문서

- 인프라 아키텍처

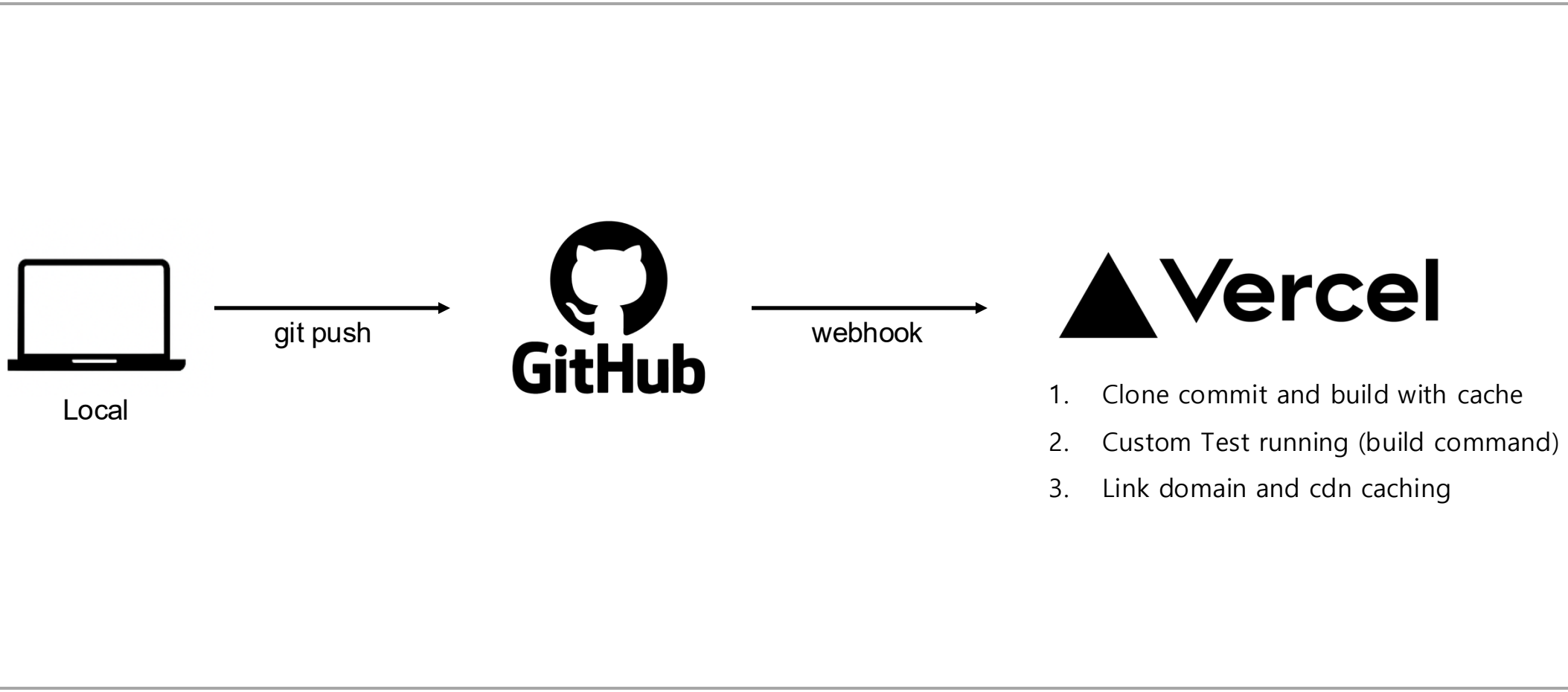
환경 구성

- React + Next.js 기반 FE application은 Vercel을 이용하여 배포
- Cloudflare Tunnel 사용하여 보안 강화
- Home Server에서 Docker로 application, Redis 구동
- Nginx로 리버스 프록시 구성하여 트래픽 라우팅 최적화
- PostgreSQL DB, 모니터링 서버 Oracle Cloud 별도 호스팅
- Prometheus + loki + alloy + Grafana로 모니터링 구현
- Jenkins CI/CD 파이프라인으로 GitHub에서 코드 자동 배포



Easy to Web | 프론트엔드 CI/CD

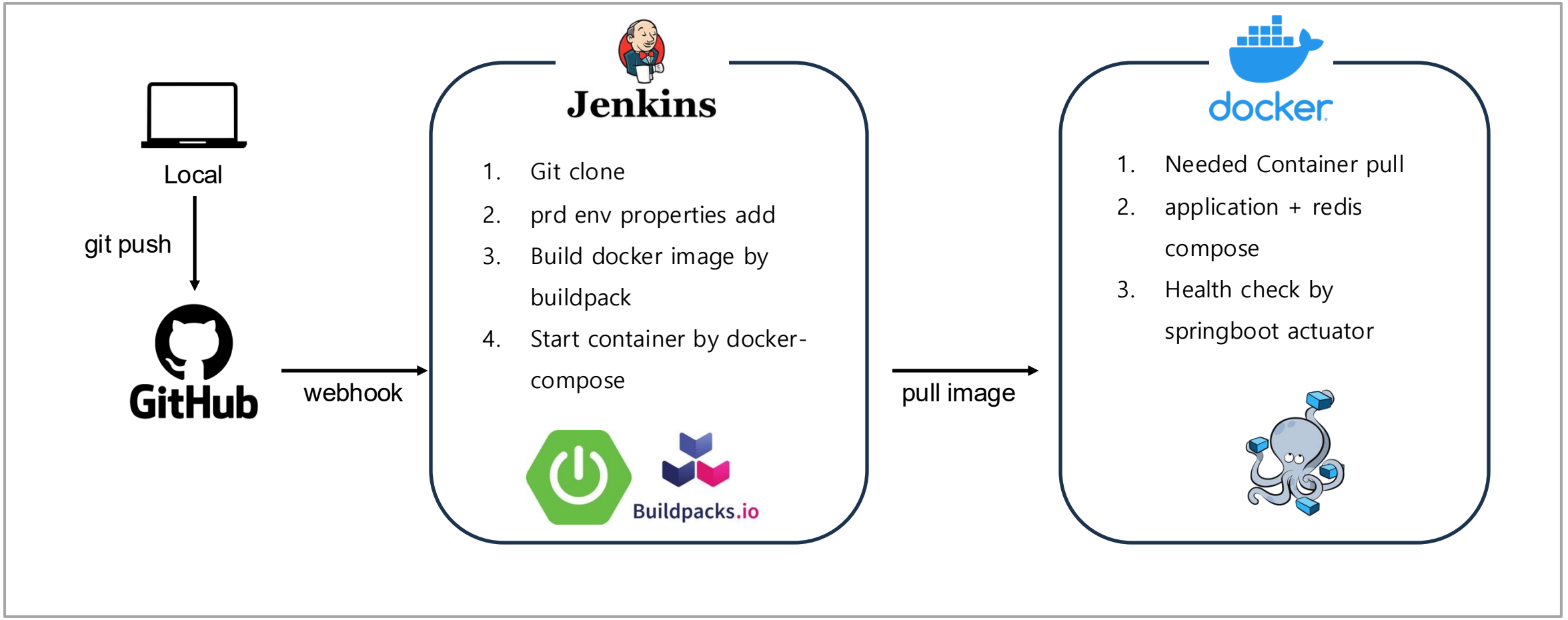
프론트엔드 코드 변경 후 main 브랜치 push 하면 GitHub Webhook 을 요청, Vercel이 이를 받아서 CI / CD 수행
빌드 중 문제 발생시 이전 배포로 롤백. 빌드 이후 도메인 연결 및 CDN Cache 갱신 수행



관련 문서 : 프론트엔드 CI/CD

Easy to Web | 백엔드 CI/CD

백엔드 코드 변경 후 main 브랜치 push 하면 GitHub Webhook 을 요청, Jenkins가 받아 CI/CD 수행.
docker compose로 앱+Redis 재배포 → 헬스체크 → 성공시 Nginx 라우팅/환경변수 반영



관련 문서 : 백엔드 CI/CD

목차

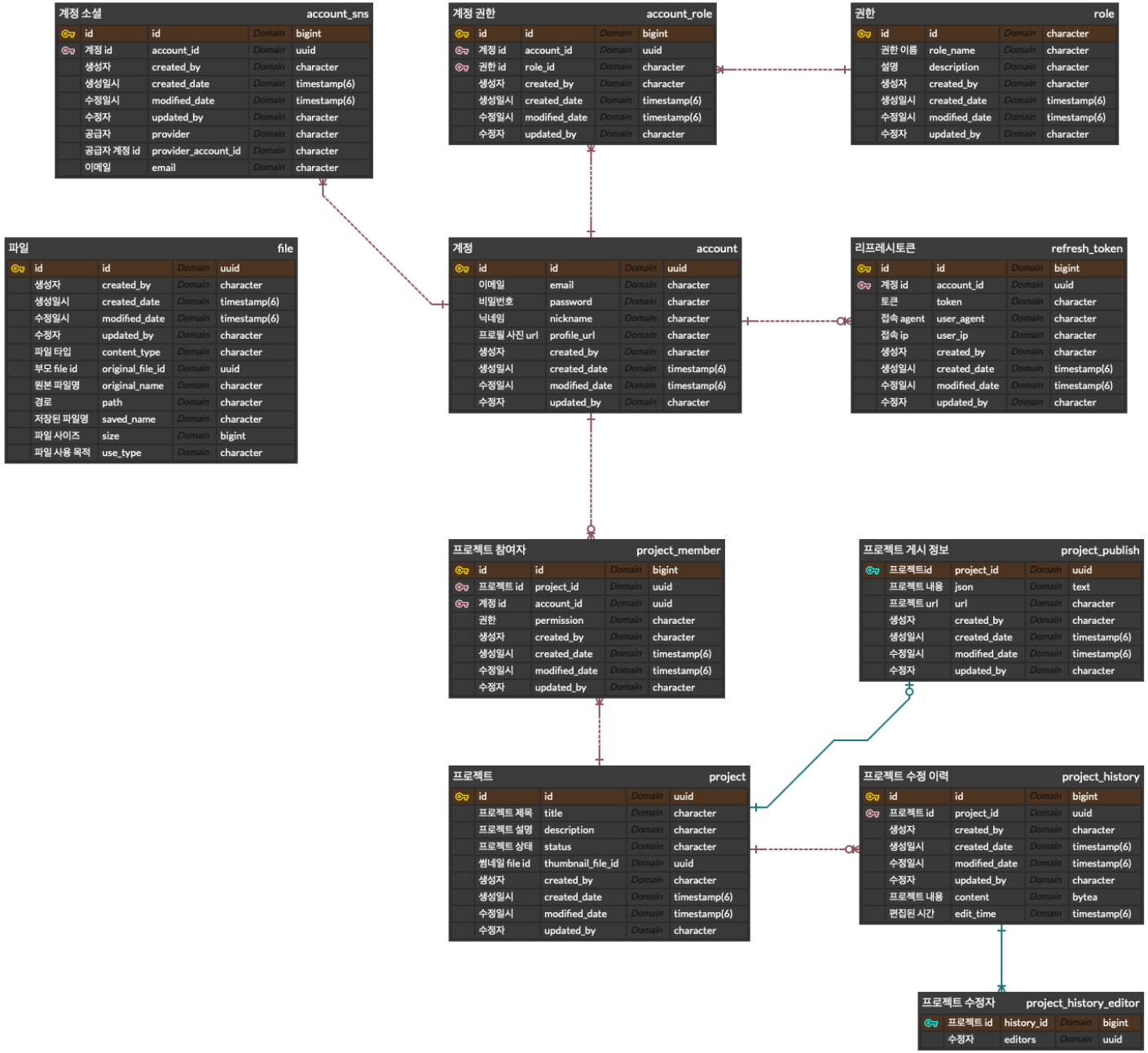
Easy to Web | Database

관련 문서

- DB 선택
- 외래키 사용방식

설계의도

- 외부에서 사용할 id는 UUID를 기본 키로 사용해 보안성 향상
- 엔티티 간 관계 분리
- 각 테이블에 Audit 필드 사용하여 메타데이터 컬럼을 통일, 추적 용이성 향상
- N:M 관계 테이블은 중간 테이블을 이용하여 관리



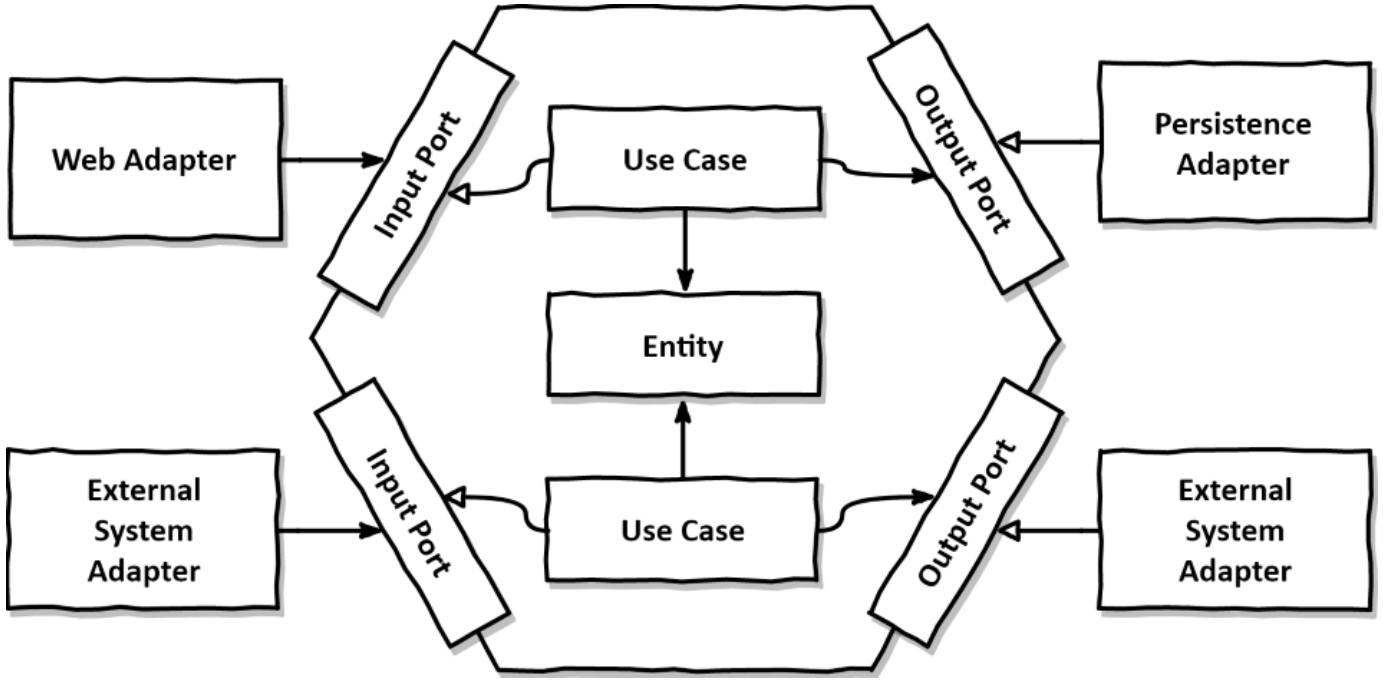
Easy to Web | 백엔드

관련 문서

- 아키텍처
- 구조

설계의도

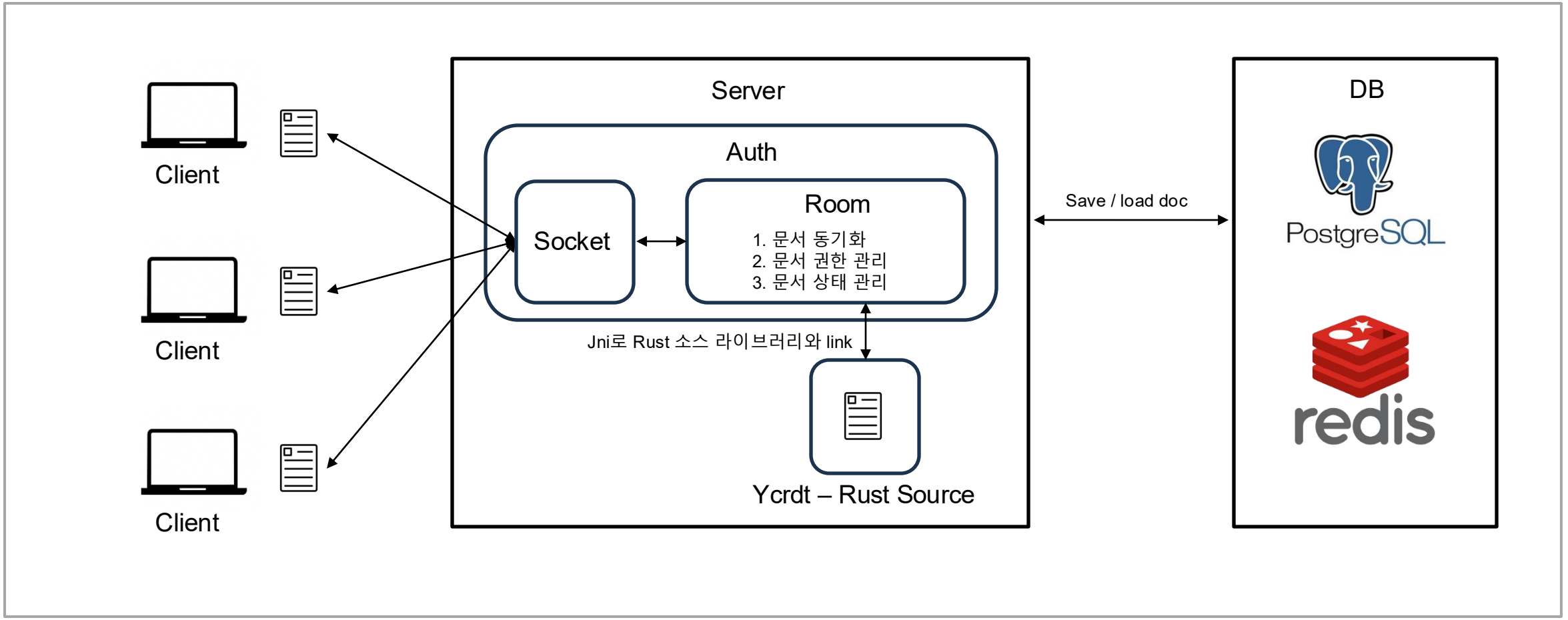
- 코드량 증가를 감수하고 유지보수성과 응집도를 위한 헥사고날 아키텍처 채택
- 도메인 중심 설계 - 핵심 로직은 순수 Java 객체로, 변경 영향 최소화
- 필요한 부분을 Mock 객체로 변경하여 테스트 용이
- 추후 인프라 확장, 교체 용이



목차

Easy to Web | 1.1 동시편집 구현 _흐름

WebSocket, Yjs 기반 CRDT를 사용하여 클라이언트 간 실시간 동기화를 지원하되, 서버에서도 문서를 활성화하여 모든 변경을 병합하며 서버 상태를 기준으로 일관성을 유지하도록 설계함.



관련 문서 :CRDT 구현 방법, 동시편집 통신 방법, 동시편집 구현 방식

Easy to Web | 1.2 동시편집 구현 _서버측 문서 병합을 위한 Custom Library

Rust 기반 y-crdt를 JNI로 연동하여 Java 환경에서도 활용 가능한 Custom Library를 구현, 서버에서 클라이언트 변경을 수집·병합해 문서 정합성을 유지.
별도의 JS 문서 서버가 필요 없고, 적합한 Java용 라이브러리가 부재했기에 서버 중심 아키텍처에 최적화된 자체 구현을 진행.

Jni 연결에 필요한 소스 형태 변경

```rust
rodminjo
#[no_mangle]
pub unsafe extern "C" fn ydoc_read_transaction(doc: *mut Doc) -> *mut Transaction {
    assert!(!doc.is_null());

    let doc : &mut Doc  = doc.as_mut().unwrap();
    if let Ok(txn : Transaction ) = doc.try_transact() {
        Box::into_raw(Box::new(Transaction::read_only(txn)))
    } else {
        null_mut()
    }
}
```

```rust
rodminjo
#[no_mangle]
pub unsafe extern "system" fn Java_com_yjava_bridge_NativeBridge_ydocReadTransaction(
    _env: JNIEnv,
    _class: JClass,
    doc_ptr: jlong,
) -> jlong {
    let doc : *mut Doc  = doc_ptr as *mut Doc;
    assert!(!doc.is_null());

    let doc : &mut Doc  = doc.as_mut().unwrap();
    let txn_ptr : *mut Transaction  = if let Ok(txn : Transaction ) = doc.try_transact() {
        Box::into_raw(Box::new(Transaction::read_only(txn)))
    } else {
        null_mut()
    };

    if txn_ptr == null_mut()  { 0 } else { txn_ptr as jlong }
}
```

native 메서드 구현

```java
Edit | Explain | Test | Document | Fix
public class NativeBridge {  19 usages  rodminjo

    private static final NativeBridge singleTon;  2 usages

    static {
        singleTon = new NativeBridge();
        System.load(NativeLibSupportSelector.support());
    }

    private NativeBridge() {}  1 usage  rodminjo

    public static NativeBridge getInstance() { return singleTon; }

    public native long ydocNew();  2 usages  rodminjo
    public native long ydocWriteTransaction(long doc_ptr, String origin);  3 usages  rodminjo
    public native long ydocReadTransaction(long doc_ptr);  3 usages  rodminjo
    public native void ytransactionCommit(long txn_ptr);  3 usages  rodminjo
    public native byte[] ytransactionStateVectorV1(long txn_ptr);  2 usages  rodminjo
    public native byte[] ytransactionStateDiffV1(long txn_ptr, byte[] diff_arr);  2 usages  rodminjo
    public native byte ytransactionApply(long txn_ptr, byte[] diff_arr);  2 usages  rodminjo
    public native void ydocDestroy(long doc_ptr);  2 usages  rodminjo
```

- Java 코드에서 Jni 를 이용하여 라이브러리 기능을 직접 호출 가능하도록 설계
- 정적블럭 내 운영체제 별 Library source 초기화 / 실행
- 애플리케이션 전체에서 단일 인스턴스를 유지하여 네이티브 리소스 관리 일관성을 확보.

운영체제별 Source 초기화

```java
Edit | Explain | Test | Document | Fix
public class MacLibSupporter implements NativeLibSupporter {  1 usage  rodminjo

    private static final String LIB_NAME = "libyrs.dylib";  1 usage

    Edit | Explain | Test | Document | Fix
    @Override  1 usage  rodminjo
    public boolean support() {
        return System.getProperty("os.name").toLowerCase().contains("mac");
    }

    Edit | Explain | Test | Document | Fix
    @Override  1 usage  rodminjo
    public String getLibPath() {
        return NativeLibSupporter.loadFromResources(LIB_NAME);
    }
}
```

```java
Edit | Explain | Test | Document | Fix
public class NativeLibSupportSelector {  2 usages  rodminjo
    private static final List<NativeLibSupporter> SUPPORTERS = List.of(  1 usage
            new MacLibSupporter(),
            new WindowsLibSupporter(),
            new LinuxLibSupporter()
    );

    Edit | Explain | Test | Document | Fix
    public static String support() {  1 usage  rodminjo
        return SUPPORTERS.stream()  Stream<NativeLibSupporter>
                .filter(NativeLibSupporter::support)
                .findFirst()  Optional<NativeLibSupporter>
                .orElseThrow(() -> new UnsupportedOperationException("Unsupported OS"))  NativeLibSupporter
                .getLibPath();
    }
}
```

관련 문서 : Custom library 구현

Easy to Web | 1.3 동시편집 구현 _문서 동기화 및 커스텀 기능을 위한 Protocol

JS 라이브러리인 y-protocol, byte 코드 사용을 위한 decoder, encoder를 Java 환경에 구현 / 적용하여 문서 상태 **업데이트와 동기화**를 지원
서버 중심 동시편집 구조에서 문서 일관성과 충돌 해결을 보장

y-protocol Java로 구현

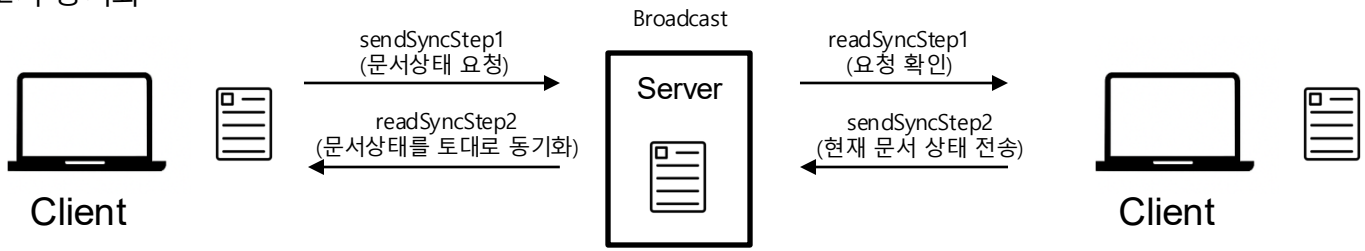
```
/**
 * @param {decoding.Decoder} decoder A message received from another client
 * @param {encoding.Encoder} encoder The reply message. Does not need to be sent if empty.
 * @param {Y.Doc} doc
 * @param {any} transactionOrigin
 */
export const readSyncMessage = (decoder, encoder, doc, transactionOrigin) => {  no usages
  const messageType = decoding.readVarUint(decoder)
  switch (messageType) {
    case messageYjsSyncStep1:
      readSyncStep1(decoder, encoder, doc)
      break
    case messageYjsSyncStep2:
      readSyncStep2(decoder, doc, transactionOrigin)
      break
    case messageYjsUpdate:
      readUpdate(decoder, doc, transactionOrigin)
      break
    default:
      throw new Error('Unknown message type')
  }
  return messageType
}
```

```
Edit | Explain | Test | Document | Fix
private void handleMessageSync(Decoder decoder, Encoder encoder, UUID accountId) {  1 usage   rodminjo
  log.debug("YjsWebSocketHandler.handleMessageSync - accountId: {}", accountId);
  switch (decoder.readVarUint()) {
    case MESSAGE_YJS_SYNC_STEP_1 -> handleMessageYjsSyncStep1(decoder, encoder);
    case MESSAGE_YJS_SYNC_STEP_2 -> handleMessageYjsSyncStep2(decoder, accountId);
    case MESSAGE_YJS_UPDATE -> handleMessageYjsUpdate(decoder, accountId);
  }
}
```

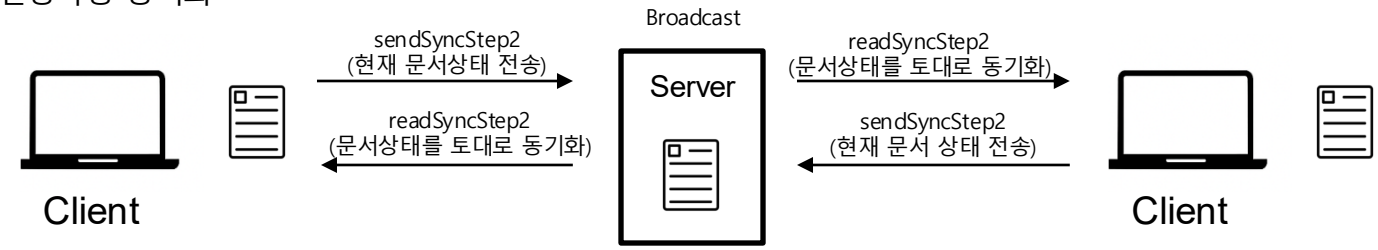
관련 문서 : [프로토콜 구현]

문서 동기화 흐름

초기 동기화



변동사항 동기화



- 각 byte 배열의 첫 byte를 사용하여 문서 타입 결정
- 서버는 권한을 확인 후 각 문서에 변동사항을 브로드캐스트
- 각 문서는 변동사항을 확인하고 반영 후 서버에 현재상태를 전송하여 전체 문서가 동기화 할 수 있도록 함
- 문서 타입을 커스텀으로 추가하여, 클라이언트가 수행할 명령을 서버에서 전송하도록 Custom 기능 구현

Easy to Web | 1.4 동시편집 구현 _ 문서 동시성 방지

Ydoc은 Transaction을 얻어 작동하게 구현됨
Socket 환경에서 여러 쓰레드가 접근할 경우 문제가 발생할 수 있어, Thread-Safe하게 문서에 접근 할 수 있도록 구현

Queue vs ReentrantLock

```
private final Thread workerThread;
private final BlockingQueue<Runnable> taskQueue = new LinkedBlockingQueue<>();

// Thread 설정
public TransactionalYDoc() {
  this.yDoc = new YDoc();
  this.lastEditTime = LocalDateTime.now();

  this.workerThread = new Thread(() -> {
    while (!Thread.currentThread().isInterrupted()) {
      try {
        Runnable task = taskQueue.take();
        task.run();
      } catch (InterruptedException e) {
        log.info("worker interrupted");
        break;
      } catch (Exception e) {
        log.info("worker exception: " + e.getMessage());
      }
    }
  }, "YDoc-Worker");

  this.workerThread.setDaemon(false);
  this.workerThread.start();
}
```

VS

```
private final ReentrantLock transactionLock = new ReentrantLock();

public TransactionalYDoc() {
  this.yDoc = new YDoc();
  this.lastEditTime = LocalDateTime.now();
}
```

관련 문서 : Transaction 구현

문서 동기화 흐름

항목	BlockingQueue + Worker Thread	ReentrantLock
구조	복잡	중간
처리 방식	비동기	동기
순서 보장	O (FIFO 큐)	O (fair mode 설정 필요)
재시도 로직	구현 쉬움	수동 구현 필요
가독성	낮음	중간
예외 처리 유연성	O (Future, 타임아웃 가능)	O
다중 조건 대기	수동 구현 필요	Condition 지원
성능 (낮은 경쟁 시)	느림	빠름
성능 (높은 경쟁 시)	안정적	경합 심함

- 두 선택지 중 현재 서버 요구조건 사용량이 크지 않아 **ReetrantLock** 를 사용하여 동시성 방지 구현
- TODO => 추후 서버 조건 향상이 요구 될 경우 **Queue + Thread로 전환 고려**

Easy to Web | 1.5 동시편집 구현 _편집 권한 관리, 효율 개선

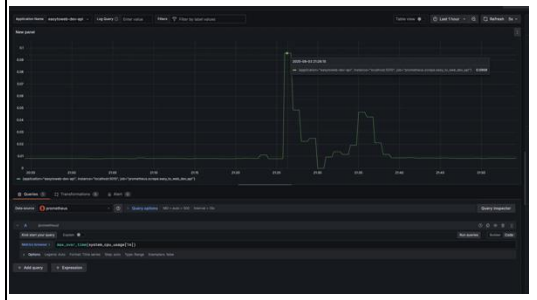
브로드캐스트 데이터를 확인하여 사용자 권한(편집, 조회 등)에 따라 전송 여부를 제어하고, 문서 편집자 혹은 뷰어 등 권한별 동작을 구현
또한 테스트를 수행하여 목표한 성능과 요구사항 충족 여부를 검증

편집 권한 관리 구현

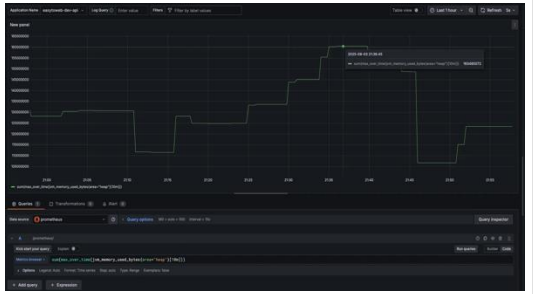
```
Edit | Explain | Test | Document | Fix
public void handle(WebSocketSession session, BinaryMessage message, UUID accountId) throws Exception {  rodminjo
  Decoder decoder = new Decoder(message.getPayload().array());
  Encoder encoder = new Encoder();

  if (!checkAuth(session, message)) {
    return;
  }

  handleByMessageType(decoder, encoder, accountId);
  send(session, message);

  if (!encoder.isEmpty()) {
    sendAll(encoder.toBinaryMessage());
  }
}
```

- 브로드캐스트 이전에 해당 문서 권한을 확인, 서버에 Editing 권한이 있을 경우에만 데이터 브로드캐스팅이 가능하도록 구현
- 데이터 타입을 확인하여 반응에 대한 데이터일 경우 브로드캐스트가 가능하도록 함

성능 테스트



CPU 사용량



메모리 사용량

성능 목표

- Client : 30, 각 client 별 초당 요청수 : 2

테스트 결과

- CPU use : 목표 환경에서 사용시 9.5% 사용
- Memory use : 목표 환경에서 사용시 153.13 MB 사용
- 평균 TPS : 44.12
- 평균 응답시간 : 0.24s

관련 문서 : 성능 테스트, 저장/불러오기/권한관리 구현

Easy to Web | 2.1 인증 구현 _요약

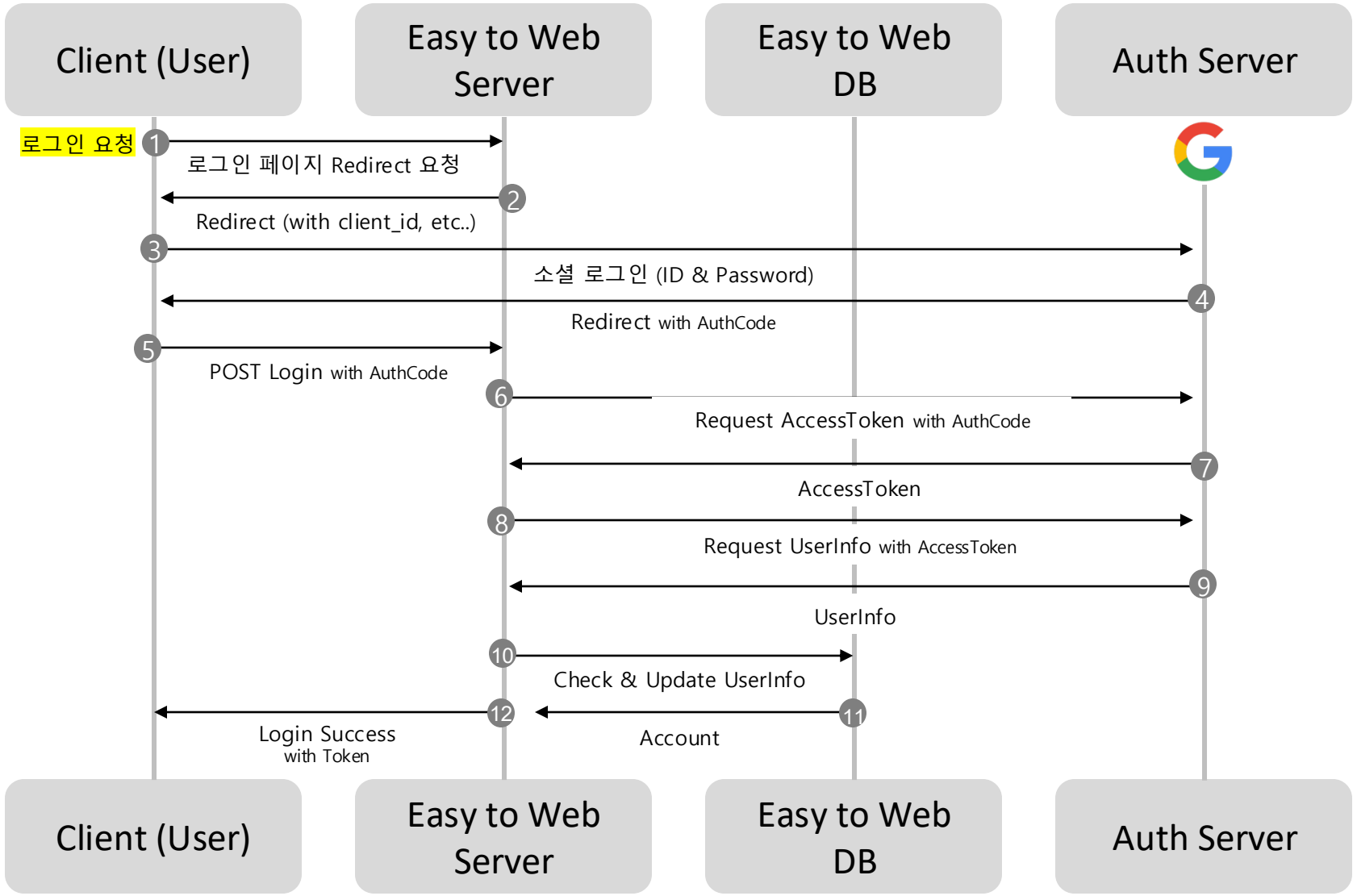
자체 로그인, 소셜 로그인 (Google) 을 구현했으며, JWT 를 사용하여 유저의 로그인 상태를 검증하고 유지하도록 구현함.
자체 로그인은 회원가입시 이메일 인증을 통해 사용자를 검증함

관련 문서 : JWT기반 인증 구현 , 이메일 인증

Easy to Web | 2.2 인증 구현 _소셜 로그인 성공 흐름

로그인 성공 흐름 설명

- **1~2** : 서버로 로그인 요청시 Auth Server로 Client_id 등 정보와 함께 Redirect
- **3~5** : Auth Server 로그인 시도 후 AuthCode와 함께 서버로 Redirect
- **6~9** : 서버에서 AuthCode로 토큰을 발급받아 유저 정보 조회
- **10~11** : 해당 정보를 받아 DB 조회 수행 유저 정보가 없다면 save, 있다면 update
- **12** : 유저정보를 담은 서버 AccessToken, RefreshToken 을 return



Easy to Web | 2.3 인증 구현 _토큰 재발급 흐름

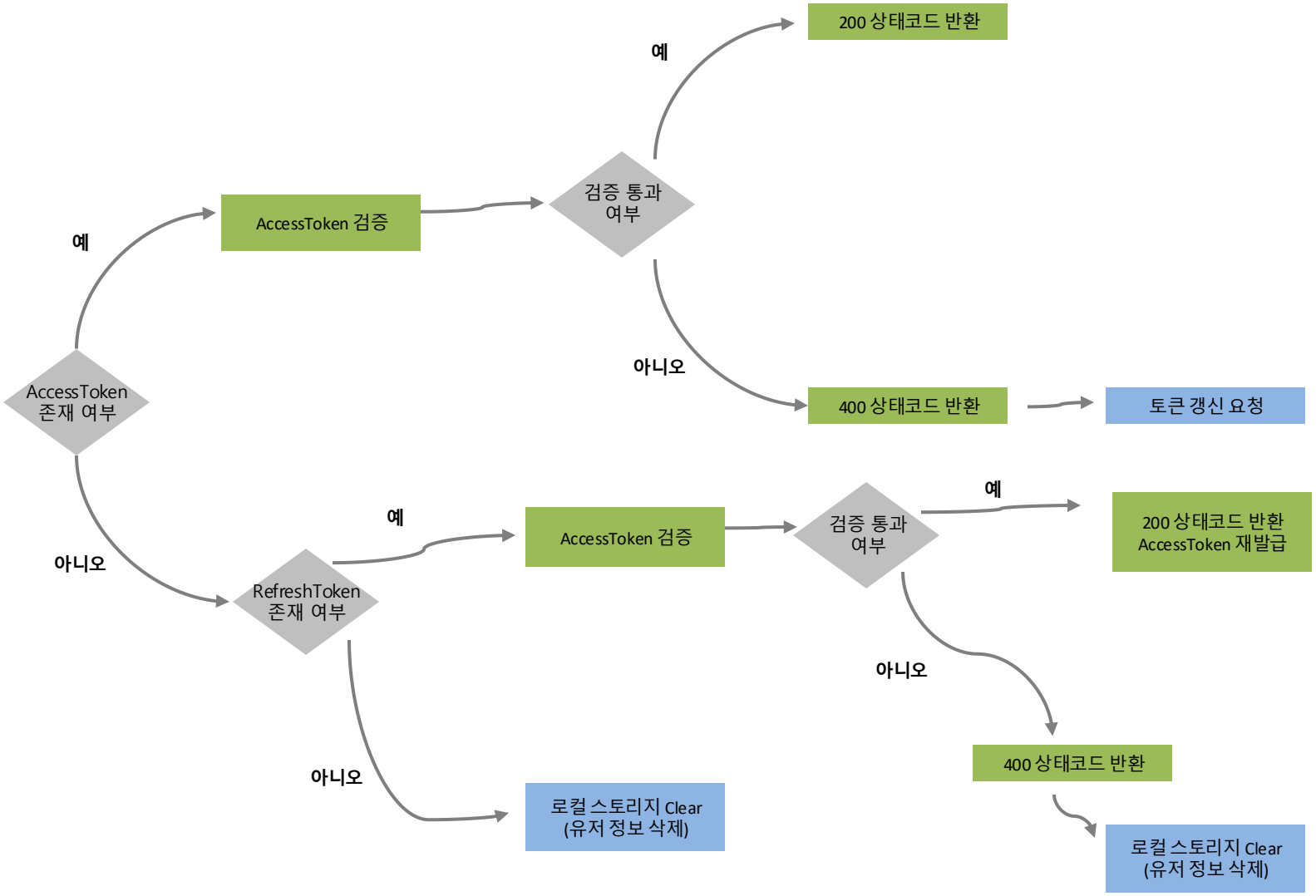
로그인 성공 흐름 설명

- 엑세스 토큰과 리프레시 토큰을 사용하여 유저가 로그인 상태를 일정 기간 유지할 수 있도록 구현함

- 이 로직은 스프링 MVC의 핸들러 인터셉터에 구현되어 모든 HTTP 요청 처리에 앞서 호출됨

- 프론트 Axios Intercepter에서 Authorization 헤더에 AccessToken을 넣고 요청 전송.
  만일 401 상태코드를 응답받을 경우,
  RefreshToken으로 토큰 갱신.
  해당 API에서도 401일 경우 로컬스토리지에 저장된 유저 정보 삭제 처리.

- 토큰 만료시간
  AccessToken : 1 시간
  RefreshToken : 30 일



Easy to Web | 2.4 인증 구현 _인증정보 사용 및 데이터 저장

Async Thread 환경에서도 인증정보를 사용할 수 있도록 static Map과 Thread Custom 구현을 이용한 인증 정보 저장 로직 구현
데이터 정합성을 보장하기 위해 사용자 인증 정보를 두 개의 테이블이 분리하여 저장함

Async Thread 환경 인증정보 사용

```java
Edit | Explain | Test | Document | Fix
public class SecurityScopeUtils {  rodminjo

  Edit | Explain | Test | Document | Fix
  // requestContextHolder.getRequestAttribute를 이용해 현재 요청 정보 받아옴
  public static Authentication getAuthentication() {  rodminjo
    try{
      Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
      if (authentication == null){throw new NullPointerException("anotherThread");}
      return authentication;

    }catch(Exception ex){
      return AsyncScopeUtils.getAuthentication();

    }
  }
```

- SecurityScopeUtils는 기본으로 SecurityContextHolder에서 인증정보 가져옴
- 만일 인증정보가 없는 비동기 환경이라면 아래 로직을 따라 인증정보를 저장, 조회할 수 있게 구현
  1. Thread 시작 전 AsyncScopeUtils에 존재하는 static Map에 Thread Name 으로 Authentication 저장
  2. Thread 종료 시 Authentication 삭제

데이터 정합성을 위해 인증정보 분리 저장

- 서버의 Account 를 기준으로 설정, 소셜 로그인시 계정 정보를 기반으로 Account를 만들고, 소셜로그인 정보를 1: N으로 저장
- Account 와 소셜로그인의 매핑 기준은 변하지 않는 값인 이메일로 설정, 자체로그인 혹은 타 소셜로그인으로 로그인시, 연동된 Account가 없다면, 이메일을 기준으로 찾고 해당 Account와 연동
- 회원은 자신의 닉네임, 프로필 사진 등을 변경할 수 있음

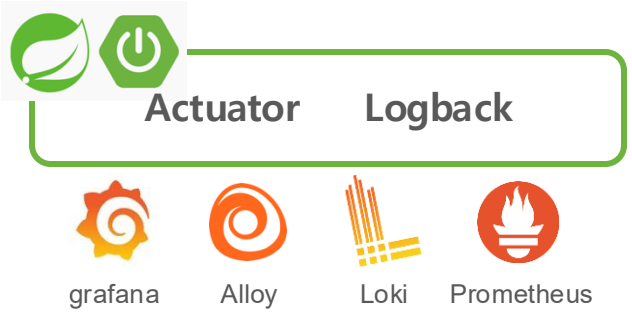
Easy to Web | 3.1 모니터링 시스템 구축_요약

효과적인 디버깅, 위기 상황 판단을 위해 모니터링 시스템을 구축하였으며, 데이터 수집 및 시각화에 대한 흥미도 로깅 및 모니터링 시스템 구축의 동기가 됨

모니터링 시스템 구축 동기

- 다양한 환경에서 에러를 마주하며 효과적인 디버깅을 위해 각 상황에 맞는 에러 감지 및 분석 시스템의 필요성 인지함
  (ex. 배포 환경 스프링 부트 앱에서 에러 발생시
  서버에 접속하여 로그 확인 vs. Grafana 대시보드에서 로그 확인)
- 데이터 기반 의사결정에 관심을 가지며, 데이터 수집과 시각화의 중요성을 인지하게 됨. 그 과정에서 로깅, 모니터링에 대한 흥미를 가지게 됨

다양한 로깅/모니터링 도구 사용



사용도구	목적 및 활용방식
Spring Actuator	1. 헬스 체크 엔드포인트 제공 2. JVM 메트릭 엔드포인트 제공 (⟹ Prometheus & Grafana 시각화)
Grafana	시각화 대시보드 제공
Alloy	메트릭, 로그 수집기
Loki	로그 저장에 필요한 기능 제공
Prometheus	메트릭 시계열 저장 기능 제공

Easy to Web | 3.2 모니터링 시스템 구축_수집 및 대시보드화

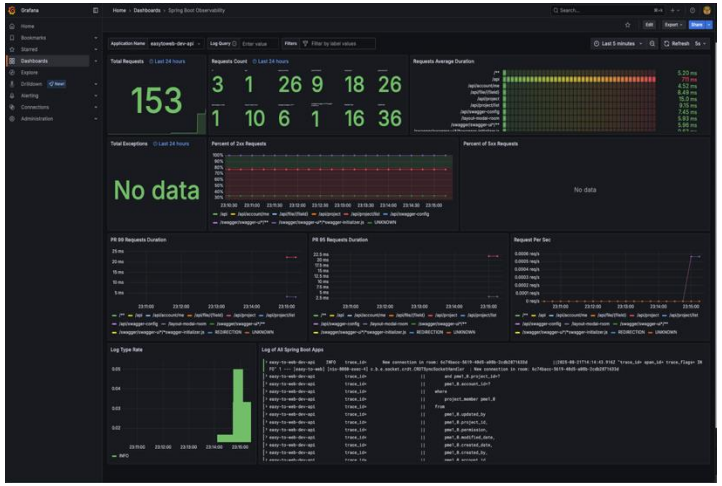
Alloy 수집기를 통해 여러서버에서 로그, 메트릭을 수집하여 모니터링 서버에 Push
Grafana를 이용하여 대시보드화 시키고, 서버 경고 수준을 설정하여 위기 발생시 Slack 알림이 오도록 연동

로그, 메트릭 수집

```
// ========================
// 메트릭, 로그 전송
// ========================

otelcol.receiver.prometheus "metrics_receiver" {
  output {
    metrics = [otelcol.exporter.otlphttp.to_admin.input]
  }
}

otelcol.receiver.loki "logs_receiver" {
  output {
    logs = [otelcol.processor.batch.to_admin.input]
  }
}

otelcol.processor.batch "to_admin" {
  timeout = "10s"
  send_batch_size = 10000

  output {
    metrics = [otelcol.exporter.otlphttp.to_admin.input]
    logs    = [otelcol.exporter.otlphttp.to_admin.input]
    traces  = [otelcol.exporter.otlphttp.to_admin.input]
  }
}


otelcol.exporter.otlphttp "to_admin" {
  client {
    endpoint = "http://localhost:30000"
  }
  sending_queue {
    enabled = true
    num_consumers = 4
    queue_size    = 8192
  }
}
```

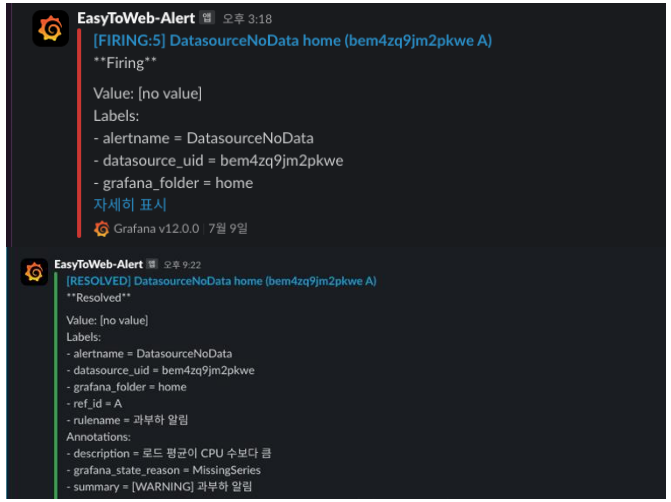
- Alloy 수집기가 여러 경로에서 들어오는 메트릭을 가공하여 모니터링 서버의 수집기로 전송

Grafana 대시보드화



- 여러 수집기에서 들어오는 메트릭을 Grafana 대시보드를 이용하여 시각화 구현
- 서버의 상태를 쉽게 파악 가능

모니터링 알람 연동

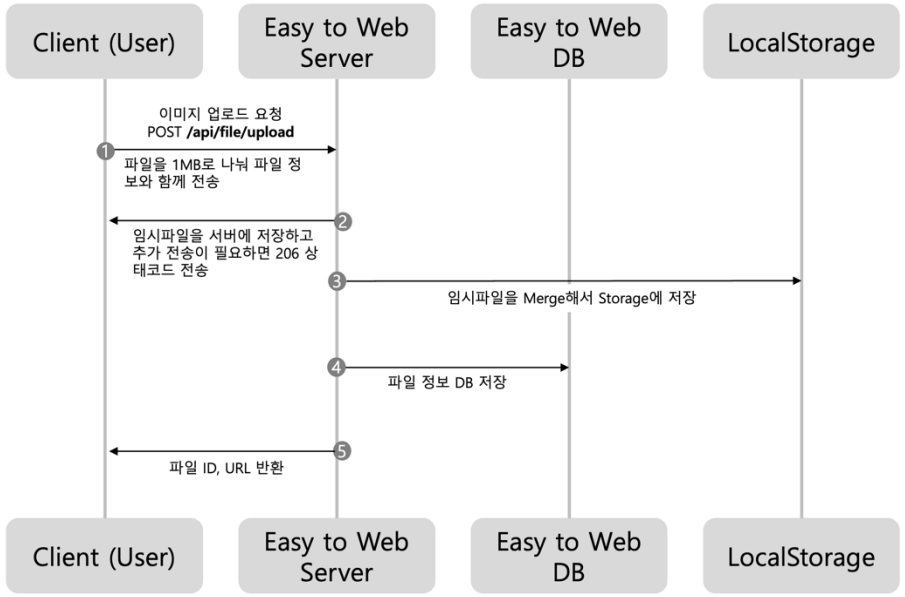


- 서버 경고수준을 설정하고 Slack과 연동하여 위기 발생시 알람이 오도록 연동
- 위기 해제시에도 알림 발송

Easy to Web | 4. 파일 첨부, 이미지 인코딩 구현_처리 과정

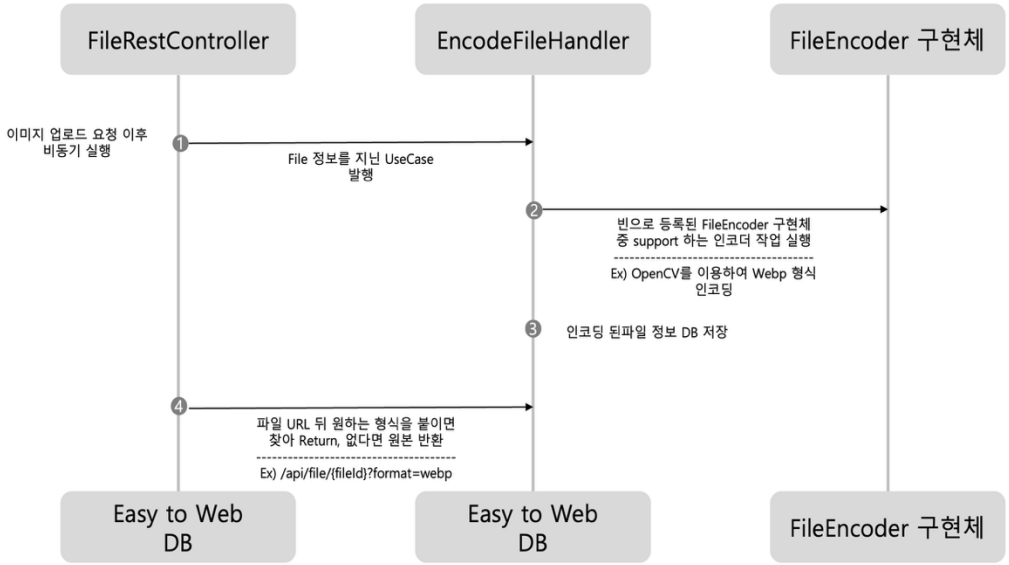
웹 문서로 이미지 등 파일을 사용할 수 있게 구현함. 또한 이미지 파일의 경우, 여러 형태로 인코딩 될 수 있도록 구현함
에러처리와 파일 권한 문제, 추후 S3로 전환 등을 고려하여 서버를 통해 업로드 하는 방식으로 구현함

파일 업로드시 처리 과정



- 추후 대용량 파일에 적용할 수 있도록 확장구현
- Interface를 활용하여 추후 S3 등 다양한 저장소로 이전할 수 있도록 구현

이미지 인코딩 처리 과정



- Strategy 패턴을 이용하여 다양한 형태의 Encoder를 구현, 사용할 수 있게 함
- 인코딩처리는 파일 저장 이후, 비동기로 작동될 수 있게 구현

관련 문서 : 대용량 청크 업로드 , 이미지 인코딩

APPENDIX

1. 문서화
2. API 문서

Easy to Web | 문서화

체계적이고 일관된 문서화를 통해 분절된 작업의 한계를 보완하고 효율성을 향상시키고자 함

전체 개발 문서 (링크)

Aa 문서	설명	문서 종류
개발 원칙	반복적인 결정 사항을 원칙으로 설정	DB
Decisions	작은 단위 의사결정	DB
기능 구현	기능 구현 과정 기록	DB
Troubleshooting	개발 중 에러 해결	DB
성능 테스트	API 성능 테스트	DB
Diagram	다이어그램 재사용 위해 관리	DB
참고	타 서비스 or 정보 참고	Single Page

주요 문서 설명

문서 이름	설명	예시
개발 원칙	• Decisions 중 비슷한 유형의 Decision 가 여럿 존재할 경우 큰 틀에서 원칙을 정해 빠르게 의사결정을 할 수 있도록 작성	• API설계원칙
Decisions	• 작은 단위 결정 사항의 결정 과정을 간단히 기록	• 인프라 아키텍처 • 인증방식
기능구현	• 기능 구현 관련 기록	• 대용량 청크 업로드 • Rust 코드 jni 연동
Troubleshooting	• 프로젝트 중 문제상황 해결 과정 정리	• Webp 인코딩 에러 • @Transactional 문제
성능테스트	• 성능 테스트 관련 문서	• Yjs 성능 테스트

Easy to Web | API 문서

Controller에 개발자가 기재한 어노테이션을 기반으로 OpenAPI 문서를 생성함.
해당 문서를 Swagger UI에서 조회 가능하도록 구현

어노테이션 기반으로 API 문서 자동 생성

```
Edit | Explain | Test | Document | Fix
@Tag(name = "Account API", description = "계정 API")  2 usages  1 implementation  rodminjo
public interface AccountRestControllerDoc {

    Edit | Explain | Test | Document | Fix
    @Operation(summary = "인증 이메일 전송", description = "인증 이메일 전송 API")  no usages  1 implementation  rodminjo
    @ApiResponse(responseCode = "200", description = "전송 성공", useReturnTypeSchema = true)
    @ApiResponseExplanations(errors = {
        @ApiExceptionExplanation(value = ExceptionMessage.INPUT_VALUE_INVALID),
        @ApiExceptionExplanation(value = ExceptionMessage.EMAIL_ALREADY_EXISTS),
        @ApiExceptionExplanation(value = ExceptionMessage.REDIS_FAILED),
        @ApiExceptionExplanation(value = ExceptionMessage.MAIL_WRITE_FAILED),
        @ApiExceptionExplanation(value = ExceptionMessage.MAIL_SEND_FAILED),
        @ApiExceptionExplanation(value = ExceptionMessage.MAIL_FILE_ERROR)
    })
    Response<JoinMailSendOutput> sendCertificationMailForJoin(JoinMailSendInput input);
```

- Controller interface에서 어노테이션을 구현하여 API문서 관련 코드를 실제 프로덕션 코드와 최대한 분리
- Custom 어노테이션을 구현하여 프로젝트 ErrorCode와 연동, 에러상태를 자동으로 작성할 수 있게 구현
- Swagger UI를 사용하여 직관적인 인터페이스를 가지고, API 테스트를 간편하게 실행 가능

관련 문서 : Swagger Error Handling 을 위한 Custom Annotation

Swagger API 문서