Presented by Group K6

GATHER LOVE

ADVANCED PROGRAMMING - 2024

TEAM MEMBERS

GROUP K6

ANDRIYO AVERILL

2306172325

Fundraising - CD - Database

HAFIZH ZEN

2306256343

Authentication

KEVIN ADRIANO

2306172552

Report - CI

KUSUMA RATIH

2306256406

Article - Admin Dashboard

ADIENA NIMEESHA

2306170912

Donation

GEORDIE VANNESE

2306170414

Wallet

GatherLove

GROUP K6

This application is a dedicated online fundraising platform where Fundraisers launch and manage donation campaigns, Donors contribute funds and track progress, and Admins ensure campaign legitimacy and oversee all activity. It features a digital wallet for secure transactions and provides transparent reporting on fund usage, acting as a secure and accountable intermediary in the fundraising process.

ADVANCED PROGRAMMING - 2024

GROUP K6

Introduction

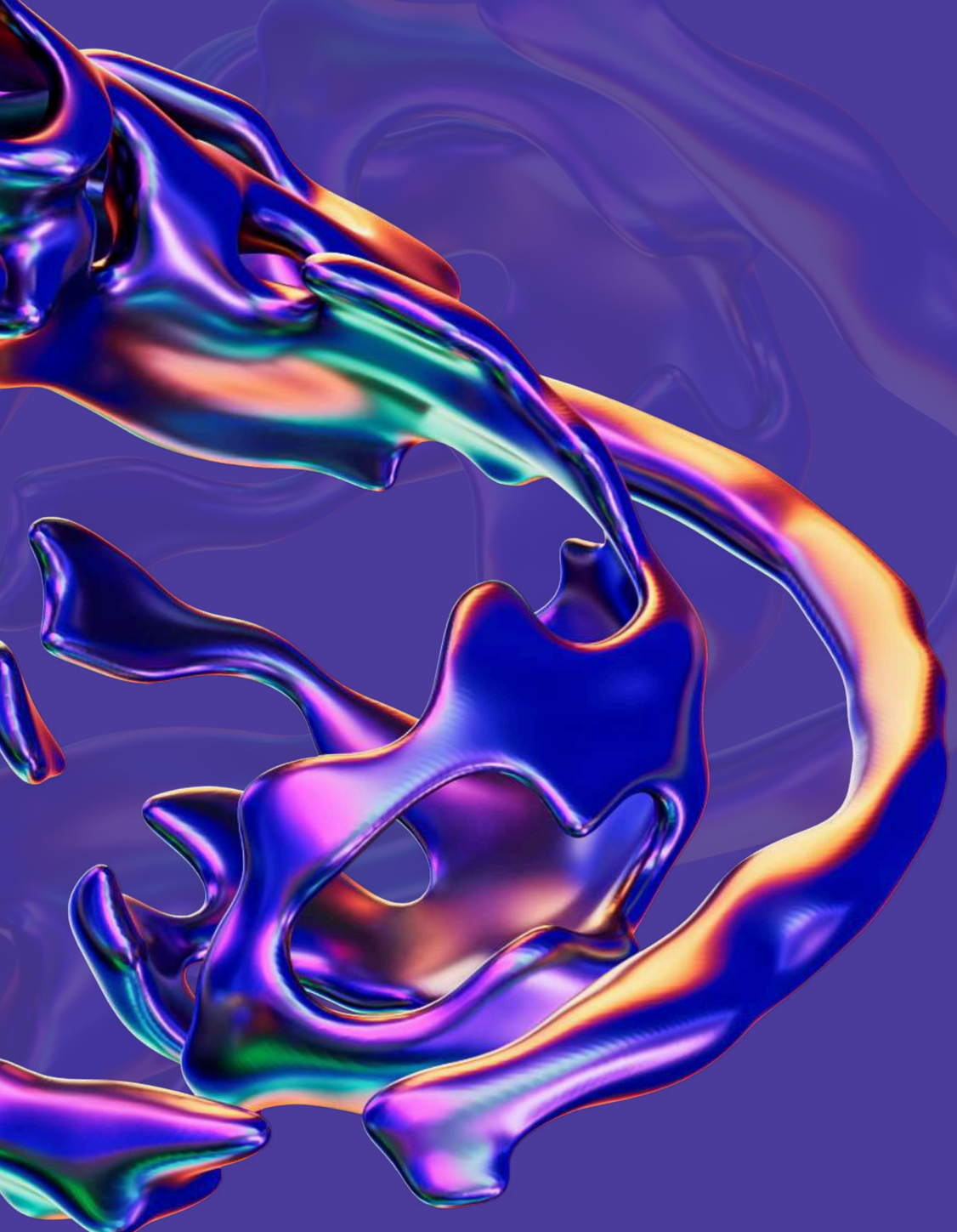
Our application is built using a Monolithic architecture, which is our GatherLove Application. This approach allows us to efficiently manage multiple critical use cases within a single, integrated system.
It consists of multiple main feature branches such as Fundraising, Wallet, Donation, Articles, Reports, Authentication.

ADVANCED PROGRAMMING - 2024

GROUP K6

Introduction

Fundraising, enabling fundraisers to create, manage, and track their campaigns. Donations, this module allows donors to easily contribute to campaigns. Our integrated digital wallet facilitates seamless transactions, including top-ups, withdrawals, and provides users with a complete transaction history. Users can report campaigns that may violate rules, and administrators on the Report module. Beyond fundraising, we offer an article module where administrators can share important updates and users can engage by reading, liking, and commenting on content.



GROUP K6

Introduction

Deployment Link: 34.206.4.43:8080/

Monitoring Link : Not Available in AWS, available in Localhost (docker-compose)

Link Sonarcloud

https://sonarcloud.io/project/overview?id=GatherLoveK6_gatherlove

ADVANCED PROGRAMMING - 2024

GROUP K6

Components

Software Design

- Design Pattern
- SOLID Principle
- Maintainability

Software Quality

- Clean Code
- Secure Coding
- Testing
- Profiling

Software Architecture

- Concurrency
- Asynchronous
- High-Level Networking

Software Deployment

- CI/CD
- Automated Deployment
- Containerization
- Infrastructure as Code
- Monitoring

GROUP K6

DESIGN PATTERN

ADVANCED PROGRAMMING - 2024

FACTORY : FUNDRAISING

GROUP K6

Our team decided to use the Factory Design Pattern in the creation and update process of the Campaign model on Fundraising.

(+) The advantage of using this design pattern is that it separates the object construction logic from the rest of the business logic.

This makes the creation of a Campaign object more organized, reusable, and easier to manage. Instead of manually setting each field repeatedly throughout the codebase, we encapsulate the creation logic in a single method (createCampaign) inside the factory. This approach increases readability, maintainability, and consistency when instantiating or updating a campaign.

```java
package k6.gatherlove.fundraising.factory;

import k6.gatherlove.fundraising.dto.CampaignCreationRequest;
import k6.gatherlove.fundraising.dto.CampaignUpdateRequest;
import k6.gatherlove.fundraising.model.Campaign;
import k6.gatherlove.fundraising.model.CampaignStatus;

import java.math.BigDecimal;
import java.time.LocalDateTime;

public class CampaignFactory {

    public static Campaign createCampaign(CampaignCreationRequest request, Long userId) {
        Campaign campaign = Campaign.builder()
                .title(request.getTitle())
                .description(request.getDescription())
                .goalAmount(request.getGoalAmount())
                .deadline(request.getDeadline())
                .status(CampaignStatus.PENDING_VERIFICATION)
                .userId(userId)
                .createdAt(LocalDateTime.now())
                .updatedAt(LocalDateTime.now())
                .currentAmount(BigDecimal.ZERO)
                .build();

        return campaign;
    }

    public static void updateCampaign(Campaign campaign, CampaignUpdateRequest request) {
        campaign.setTitle(request.getTitle());
        campaign.setDescription(request.getDescription());
        campaign.setGoalAmount(request.getGoalAmount());
        campaign.setDeadline(request.getDeadline());
        campaign.setUpdatedAt(LocalDateTime.now());
    }
}
```

ADVANCED PROGRAMMING - 2024

FACTORY: REPORT

Our team decided to use the Factory Design Pattern in determining which report service implementation to use based on the user's role.

(+) The advantage of using this design pattern is that it centralizes the logic of selecting the correct service implementation in one place (the factory class).

This makes the controller and other parts of the application cleaner and easier to maintain. If a new role or service is added in the future, we only need to update the factory class without touching other parts of the code. This follows the Open/Closed Principle of SOLID design principles, promoting better modularity and extensibility.

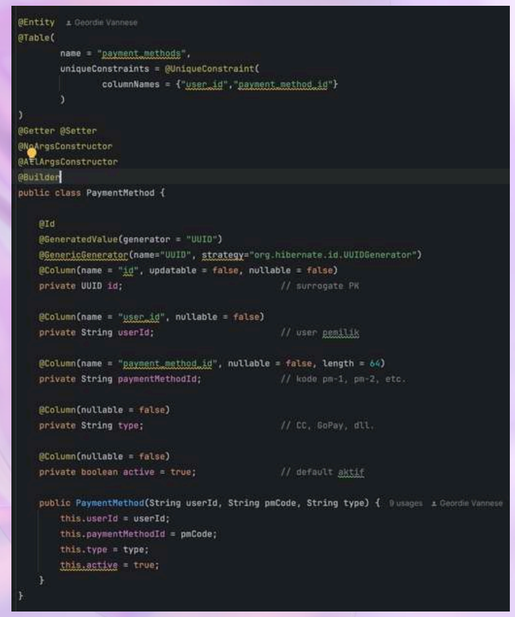
```java
package k6.gatherlove.report.factory;

import k6.gatherlove.report.enums.Role;
import k6.gatherlove.report.service.AdminReportServiceImpl;
import k6.gatherlove.report.service.ReportService;
import k6.gatherlove.report.service.UserReportServiceImpl;
import org.springframework.stereotype.Component;

@Component
public class ReportServiceFactory {

    private final UserReportServiceImpl userReportServiceImpl;
    private final AdminReportServiceImpl adminReportServiceImpl;

    public ReportServiceFactory(UserReportServiceImpl userReportServiceImpl, AdminReportServiceImpl adminReportServiceImpl) {
        this.userReportServiceImpl = userReportServiceImpl;
        this.adminReportServiceImpl = adminReportServiceImpl;
    }

    public ReportService getReportAction(Role role) {
        if (role == null) {
            throw new IllegalArgumentException("Role cannot be null");
        }

        switch (role) {
            case ADMIN:
                return adminReportServiceImpl;
            case USER:
                return userReportServiceImpl;
            default:
                throw new IllegalArgumentException("Unsupported role: " + role);
        }
    }
}
```

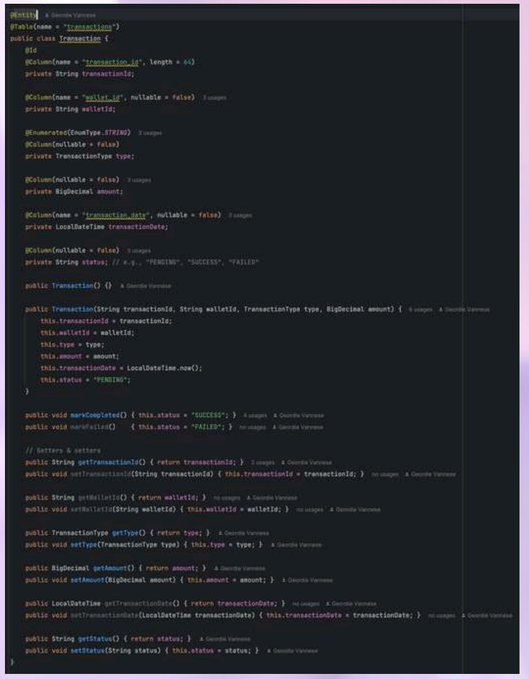
Domain-Driven Design : WALLET

Our team decided to model all of our wallet-related objects (Wallet, Transaction, and PaymentMethod) as Domain Models backed by JPA/Hibernate's Data Mapper Pattern, so that persistence logic stays neatly separate from our core business behaviors. In the Transaction entity we went further by applying the Type-Safe Enum Pattern(TransactionType) to make each operation TOP_UP or WITHDRAW explicit and checked at compile time. Meanwhile, for PaymentMethod we leveraged Lombok's Builder Pattern (@Builder) to give us a fluent, self-documenting way to create payment-method instances, and declared a unique constraint on (user_id, payment_method_id) right in the table definition to enforce our business rule at the database level.

(+) The combined advantage is that each entity fully encapsulates its own behavior like topUp(), withdraw(), or markCompleted() making the code highly cohesive and self-validating. JPA/Hibernate handles all the tedious SQL mapping, our enum ensures valid transaction types, the builder gives clear and concise instantiation for payment methods, and the database constraint guarantees data integrity. This yields a module that's structured, readable, and robust against both coding and data errors.

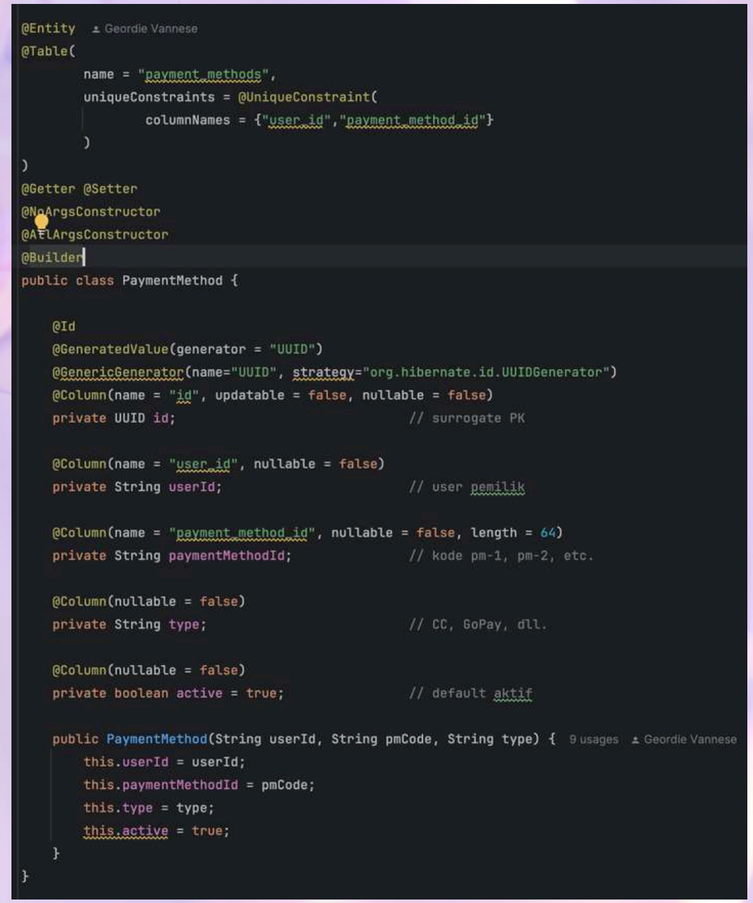
ADVANCED PROGRAMMING - 2024





Data Mapper Pattern & Builder Pattern : WALLET

For PaymentMethod, we again combined the Domain Model and Data Mapper patterns so persistence stays separate from logic. On top of that, we used Lombok’s Builder Pattern (@Builder) to give us a clear, fluent API when creating new payment methods. Finally, we declared a unique constraint on (user_id, payment_method_id) at the table level to enforce our business rule directly in the database.



```java
@Entity  Geordie Vannese
@Table(
        name = "payment_methods",
        uniqueConstraints = @UniqueConstraint(
                columnNames = {"user_id","payment_method_id"}
        )
)
@Getter @Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class PaymentMethod {

    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(name="UUID", strategy="org.hibernate.id.UUIDGenerator")
    @Column(name = "id", updatable = false, nullable = false)
    private UUID id;                                // surrogate PK

    @Column(name = "user_id", nullable = false)
    private String userId;                          // user pemilik

    @Column(name = "payment_method_id", nullable = false, length = 64)
    private String paymentMethodId;                 // kode pm-1, pm-2, etc.

    @Column(nullable = false)
    private String type;                            // CC, GoPay, dll.

    @Column(nullable = false)
    private boolean active = true;                  // default aktif

    public PaymentMethod(String userId, String pmCode, String type) {  9 usages  Geordie Vannese
        this.userId = userId;
        this.paymentMethodId = pmCode;
        this.type = type;
        this.active = true;
    }
}
```

ADVANCED PROGRAMMING - 2024

Data Mapper Pattern & Type-Safe Enum Pattern : WALLET

The Transaction entity is also a Domain Model backed by JPA/Hibernate’s Data Mapper Pattern, but with an extra layer of safety: we applied the Type-Safe Enum Pattern via our TransactionType enum. By doing so, every Transaction can only be TOP_UP or WITHDRAW, and the compiler enforces it. Business behaviors like markCompleted() or markFailed() live inside the entity, keeping our code self-validating and expressive.

```java
@Entity  Geordie Vannese
@Table(name = "transactions")
public class Transaction {
    @Id
    @Column(name = "transaction_id", length = 64)
    private String transactionId;

    @Column(name = "wallet_id", nullable = false)  3 usages
    private String walletId;

    @Enumerated(EnumType.STRING)  3 usages
    @Column(nullable = false)
    private TransactionType type;

    @Column(nullable = false)  3 usages
    private BigDecimal amount;

    @Column(name = "transaction_date", nullable = false)  3 usages
    private LocalDateTime transactionDate;

    @Column(nullable = false)  5 usages
    private String status; // e.g., "PENDING", "SUCCESS", "FAILED"

    public Transaction() {}  Geordie Vannese

    public Transaction(String transactionId, String walletId, TransactionType type, BigDecimal amount) {  6 usages  Geordie Vannese
        this.transactionId = transactionId;
        this.walletId = walletId;
        this.type = type;
        this.amount = amount;
        this.transactionDate = LocalDateTime.now();
        this.status = "PENDING";
    }

    public void markCompleted() { this.status = "SUCCESS"; }  4 usages  Geordie Vannese
    public void markFailed()    { this.status = "FAILED"; }  no usages  Geordie Vannese

    // Getters & setters
    public String getTransactionId() { return transactionId; }  2 usages  Geordie Vannese
    public void setTransactionId(String transactionId) { this.transactionId = transactionId; }  no usages  Geordie Vannese

    public String getWalletId() { return walletId; }  no usages  Geordie Vannese
    public void setWalletId(String walletId) { this.walletId = walletId; }  no usages  Geordie Vannese

    public TransactionType getType() { return type; }  Geordie Vannese
    public void setType(TransactionType type) { this.type = type; }  Geordie Vannese

    public BigDecimal getAmount() { return amount; }  Geordie Vannese
    public void setAmount(BigDecimal amount) { this.amount = amount; }  Geordie Vannese

    public LocalDateTime getTransactionDate() { return transactionDate; }  no usages  Geordie Vannese
    public void setTransactionDate(LocalDateTime transactionDate) { this.transactionDate = transactionDate; }  no usages  Geordie Vannese

    public String getStatus() { return status; }  Geordie Vannese
    public void setStatus(String status) { this.status = status; }  Geordie Vannese
}
```

```java
package k6.gatherlove.wallet.domain;

public enum TransactionType {  14 usages
    TOP_UP,  5 usages
    WITHDRAW;  2 usages
}
```

Data Mapper Pattern : WALLET

We modeled the Wallet class as a Domain Model and used JPA/Hibernate’s Data Mapper Pattern to handle persistence. This lets us keep methods like topUp() and withdraw()—which update balance and timestamps—right inside the Wallet entity, while Hibernate automatically maps those changes to the wallets table without cluttering our business logic with SQL.

```java
@Entity  Geordie Vannese
@Table(name = "wallets")
public class Wallet {
    @Id
    @Column(name = "wallet_id", length = 36, nullable = false)
    private String walletId;

    @Column(name = "user_id", nullable = false)  3 usages
    private String userId;

    @Column(nullable = false)  8 usages
    private BigDecimal balance;

    @Column(name = "created_at", nullable = false)  3 usages
    private LocalDateTime createdAt;

    @Column(name = "updated_at", nullable = false)  5 usages
    private LocalDateTime updatedAt;

    public Wallet() {}  Geordie Vannese

    public Wallet(String walletId, String userId) {  9 usages  Geordie Vannese
        this.walletId = walletId;
        this.userId = userId;
        this.balance = BigDecimal.ZERO;
        this.createdAt = LocalDateTime.now();
        this.updatedAt = LocalDateTime.now();
    }

    public boolean canWithdraw(BigDecimal amount) { return balance.compareTo(amount) >= 0; }

    public void topUp(BigDecimal amount) {  6 usages  Geordie Vannese
        balance = balance.add(amount);
        updatedAt = LocalDateTime.now();
    }

    public void withdraw(BigDecimal amount) {  3 usages  Geordie Vannese
        if (!canWithdraw(amount)) {
            throw new IllegalArgumentException("Insufficient funds");
        }
        balance = balance.subtract(amount);
        updatedAt = LocalDateTime.now();
    }

    // Getters & setters
    public String getWalletId() { return walletId; }  3 usages  Geordie Vannese
    public void setWalletId(String walletId) { this.walletId = walletId; }  no usages  Geordie Vannese

    public String getUserId() { return userId; }  Geordie Vannese
    public void setUserId(String userId) { this.userId = userId; }  Geordie Vannese

    public BigDecimal getBalance() { return balance; }  6 usages  Geordie Vannese
    public void setBalance(BigDecimal balance) { this.balance = balance; }  no usages  Geordie Vannese

    public LocalDateTime getCreatedAt() { return createdAt; }  Geordie Vannese
    public void setCreatedAt(LocalDateTime createdAt) { this.createdAt = createdAt; }  Geordie Vannese

    public LocalDateTime getUpdatedAt() { return updatedAt; }  no usages  Geordie Vannese
    public void setUpdatedAt(LocalDateTime updatedAt) { this.updatedAt = updatedAt; }  no usages  Geordie Vannese
}
```

Data Transfer Object: Donation

We use DTOs like CommentRequest and DonationRequest to cleanly decouple our API layer from the core domain model: by mapping incoming JSON into a focused object, we control exactly which fields can be set (and validate them with annotations) before they ever reach our entities or business logic. This approach keeps our controllers thin, shields internal details such as database IDs or audit timestamps from public exposure, and makes it far easier to evolve the API contract, if we need to rename, add, or deprecate a property, we only touch the DTO rather than risk breaking service code or unintended side effects in our persistence layer.

```
@Autowired  adienanimeesha
public DonationController(DonationService donationService) { this.donationService = donationService; }

@PostMapping  adienanimeesha
public Donation create(@RequestBody DonationRequest req) {
    return donationService.createDonation(
            req.getUserId(),
            req.getAmount(),
            req.getCampaignId()
    );
}
```

```
@PostMapping  adienanimeesha
@ResponseStatus(HttpStatus.CREATED)   // returns 201 Created instead of 200 OK
public Comment addComment(
        @PathVariable String campaignId,
        @RequestBody CommentRequest req
) {
    return commentService.addComment(
            campaignId,
            req.getUserId(),
            req.getText()
    );
}
```

Layered Architecture Pattern: ARTICLE

We applied the Layered Architecture Pattern in the Article Management module. This pattern separates the application into clear layers: Controller (handles requests), Service (business logic), Repository (data access), and Model (data structure). As shown in ArticleManagementServiceImpl.java, the service layer ensures that business rules are isolated, making the system modular, testable, and easy to maintain.

```java
@Service  ksmratih
@RequiredArgsConstructor
public class ArticleManagementServiceImpl implements ArticleManagementService {

    private final ArticleManagementRepository articleRepository;
    private final RestTemplate restTemplate = new RestTemplate(); // REST client  1 usage

    @Override  2 usages  ksmratih
    public ArticleManagementModel createArticle(ArticleManagementModel article) {
        ArticleManagementModel saved = articleRepository.save(article);

        // High-Level Networking: Notify another service
        try {
            String message = "A new article titled '" + saved.getTitle() + "' has been created.";
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            String body = "{\"message\": \"" + message + "\"}";
            HttpEntity<String> request = new HttpEntity<>(body, headers);

            // Replace with actual target service URL
            String url = "http://localhost:8081/notify";
            restTemplate.postForEntity(url, request, String.class);

        } catch (Exception e) {
            System.err.println("Failed to notify external service: " + e.getMessage());
        }

        return saved;
    }
    @Override  3 usages  ksmratih
    public ArticleManagementModel getArticleById(Long id) {
        return articleRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Article not found"));
    }

    @Override  2 usages  ksmratih
    public List<ArticleManagementModel> getAllArticles() {
        return articleRepository.findAll();
    }

    @Override  3 usages  ksmratih
    public ArticleManagementModel updateArticle(Long id, ArticleManagementModel updatedArticle) {
        ArticleManagementModel existing = getArticleById(id);
        existing.setTitle(updatedArticle.getTitle());
        existing.setContent(updatedArticle.getContent());
        return articleRepository.save(existing);
    }

    @Override  2 usages  ksmratih
    public void deleteArticle(Long id) {
        articleRepository.deleteById(id);
    }
}
```

Strategy Pattern & Chain-of-Responsibility Pattern: AUTHENTICATION

Our authentication module employs the Chain-of-Responsibility Pattern, which involves adding a custom JwtAuthenticationFilter into Spring Security's filter chain so that each request goes through a clear sequence of responsibility for token extraction, validation, and security context setup, and the Strategy Pattern, which defines an AuthStrategy interface with concrete implementations like UserAuthStrategy (and easily addable OAuth or LDAP strategies) to encapsulate all login and registration behavior. With controllers handling HTTP traffic, strategies handling credential logic, and filters handling security plumbing, this combination creates a highly extensible, cleanly divided architecture that is easy to unit-test and expand with new authentication techniques.

```java
package kó.gatherlove.auth.filter;

import ...

@Component 1 inheritor Hafizh-Zen
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;

    @Override 3 usages 1 override Hafizh-Zen
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain)
            throws IOException, ServletException {

        // 1) extract JWT from cookie
        String token = null;
        if (req.getCookies() != null) {
            for (Cookie c : req.getCookies()) {
                if ("JWT".equals(c.getName())) {
                    token = c.getValue();
                    break;
                }
            }
        }

        // 2) validate & set Authentication
        if (token != null && jwtUtil.validateToken(token)) {
            var auth = jwtUtil.getAuthentication(token);
            SecurityContextHolder.getContext().setAuthentication(auth);
        }

        chain.doFilter(req, res);
    }
}
```

```java
package kó.gatherlove.auth.strategy;

import kó.gatherlove.auth.dto.RegisterUserRequest;
import kó.gatherlove.auth.model.User;

public interface AuthStrategy { 5 usages 1 implementation Ha
    User login(String email, String password); 6 usages 1 in
    User register(RegisterUserRequest request); 3 usages 1
}
```

```java
package kó.gatherlove.auth.strategy;

import ...

@Service Hafizh-Zen
@RequiredArgsConstructor
public class UserAuthStrategy implements AuthStrategy {

    private final UserRepository userRepository;
    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder(); 2 us

    @Override 6 usages Hafizh-Zen
    public User login(String email, String password) {
        User user = userRepository.findByEmail(email)
                .orElseThrow(() -> new RuntimeException("User not found"));

        if (!passwordEncoder.matches(password, user.getPassword())) {
            throw new RuntimeException("Invalid credentials");
        }

        return user;
    }

    @Override 3 usages Hafizh-Zen
    public User register(RegisterUserRequest request) {
        // build a new User entity
        User user = User.builder()
                .fullName(request.getFullName())
                .email(request.getEmail())
                .username(request.getUsername())
                .password(passwordEncoder.encode(request.getPassword()))
                .phone(request.getPhone())
                .address(request.getAddress())
                // always give new users the USER role:
                .roles(Set.of(Role.ROLE_USER))
                .build();

        return userRepository.save(user);
    }
}
```

GROUP K6

SOLID Principle Implementations

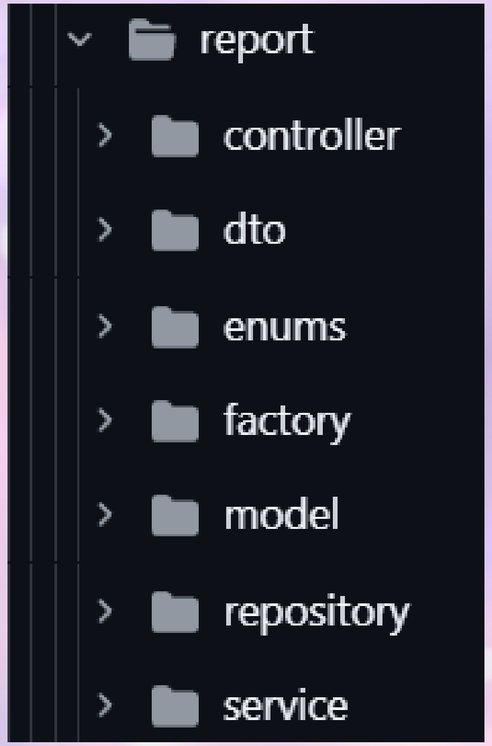
Proyek Grup 2024

Single Responsibility Principle

GROUP K6

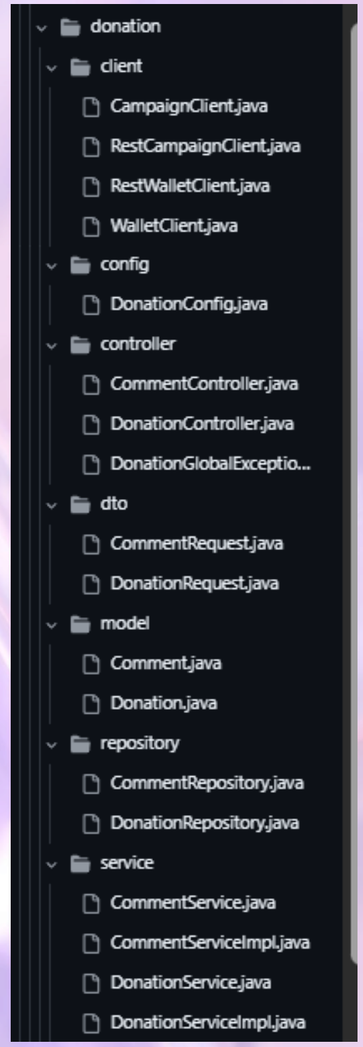
Our team applied the Single Responsibility Principle by ensuring that each file is focused on a single functionality. We implemented this by separating files based on their responsibilities, such as Controller, Service, Repository, etc.

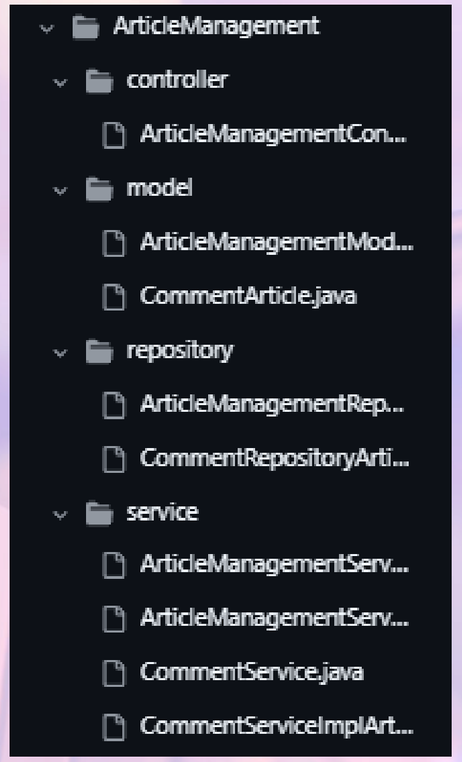
Each Controller, Service, and Repository is also further divided into multiple files to distinguish their specific functionalities. For example, ReportController is used specifically for handling endpoints related to "/report".



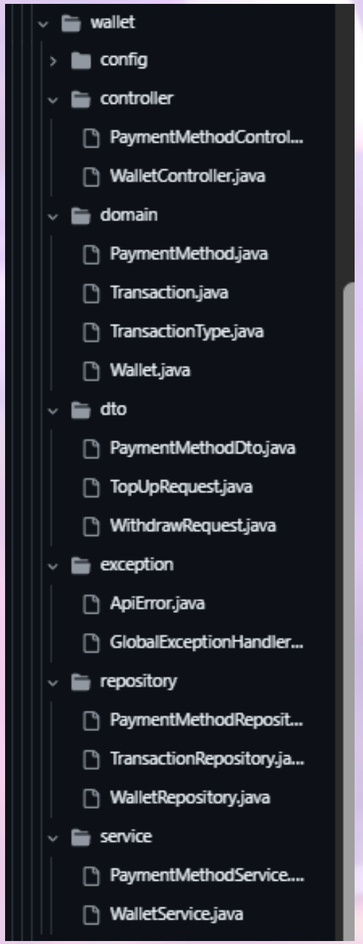
Single Responsibility Principle

GROUP K6





examples from other modules



ADVANCED PROGRAMMING - 2024

Open Closed Principle

GROUP K6

We applied the Open/Closed Principle by designing our system in a way that allows new features or behaviors to be added through extension rather than modification of existing code. To achieve this, we created abstract classes or interfaces that define common behaviors, and then implemented them in separate concrete classes based on specific needs.

Each concrete class provides its own implementation of the defined methods, allowing flexibility and customization without altering the original structure. When a new functionality is needed, we can simply create a new class that extends the abstract class or implements the interface—without touching the existing codebase. This approach ensures that our system is open for extension but closed for modification, promoting maintainability and scalability.

```java
package k6.gatherlove.report.service;

import k6.gatherlove.report.model.Report;
import java.util.List;
import java.util.UUID;

public interface ReportService {
    Report createReport(String campaignId, String userId, String title, String description, String violationType);
    List<Report> viewReports(String userId);
    void deleteReport(UUID reportId);
    void verifyCampaign(String campaignId);
}
```

Open Closed Principle

GROUP K6

```java
package k6.gatherlove.fundraising.service;

import k6.gatherlove.fundraising.dto.CampaignCreationRequest;
import k6.gatherlove.fundraising.dto.CampaignUpdateRequest;
import k6.gatherlove.fundraising.model.Campaign;

import java.util.List;
import java.util.Optional;

import org.springframework.web.multipart.MultipartFile;

public interface CampaignService {
    Campaign createCampaign(CampaignCreationRequest request, Long userId);
    List<Campaign> getCampaignsByUserId(Long userId);
    Optional<Campaign> getCampaignById(Long id);
    List<Campaign> getAllCampaigns();

    Campaign updateCampaign(Long id, CampaignUpdateRequest request, Long userId);
    void deleteCampaign(Long id, Long userId);
    void uploadProofOfFundUsage(Long id, Long userId, MultipartFile file);
    void verifyCampaign(Long id, boolean approved);
    List<Campaign> getActiveCampaigns();

    /**
     * Get all campaigns with PENDING_VERIFICATION status
     * @return List of pending campaigns
     */
    List<Campaign> getPendingCampaigns();
}
```

```java
package k6.gatherlove.ArticleManagement.service;

import k6.gatherlove.ArticleManagement.model.CommentArticle;

import java.util.List;

public interface CommentService {
    CommentArticle addComment(Long articleId, String username, String content);
    List<CommentArticle> getCommentsForArticle(Long articleId);
}
```

```java
package k6.gatherlove.donation.service;


import k6.gatherlove.donation.model.Comment;
import java.util.List;

public interface CommentService {
    Comment addComment(String campaignId, String userId, String text);
    List<Comment> listComments(String campaignId);
}
```

ADVANCED PROGRAMMING - 2024

Liskov Substitution Principle

GROUP K6

We applied the Liskov Substitution Principle by creating an abstract class that defines common methods such as create, getAll, and getCurrentData. Subclasses can use these existing methods as-is or override them to provide more specific behavior—without breaking the expected functionality of the system.

This means that any subclass can be used in place of its parent class without affecting the correctness of the program. By following this principle, we ensure that our code remains reliable, extendable, and that inheritance hierarchies are used in a safe and consistent manner.

ADVANCED PROGRAMMING - 2024

Liskov Substitution Principle

GROUP K6

```java
package k6.gatherlove.donation.service;


import k6.gatherlove.donation.model.Comment;
import java.util.List;

public interface CommentService {
    Comment addComment(String campaignId, String userId, String text);
    List<Comment> listComments(String campaignId);
}
```

```java
@Override
@Transactional
public Comment addComment(String campaignId, String userId, String text) {
    if (text == null || text.isBlank()) {
        throw new IllegalArgumentException("Comment text cannot be empty");
    }

    Comment saved = repo.save(new Comment(campaignId, userId, text));

    // notify fundraiser via REST call, only if we have a URL
    if (!notificationBaseUrl.isBlank()) {
        try {
            rest.postForEntity(
                    notificationBaseUrl + "/notifications/comments",
                    Map.of(
                            "campaignId", campaignId,
                            "commentId",   saved.getId(),
                            "userId",      userId,
                            "text",        text
                    ),
                    Void.class
            );
        } catch (RestClientException ex) {
            System.err.println("Warning: failed to notify on new comment: " + ex.getMessage());
        }
    }

    return saved;
}
```

ADVANCED PROGRAMMING - 2024

Interface Segregation Principle

GROUP K6

Our team applied the Interface Segregation Principle by creating a separate interface for each service. Each service class then implements only the methods defined in its corresponding interface, according to its specific responsibilities.

These service interfaces are then used in the controllers, ensuring that the controllers depend only on the functionalities they actually use. This keeps our code clean, modular, and easy to maintain, while also avoiding bloated interfaces that force classes to implement unnecessary methods.

ADVANCED PROGRAMMING - 2024

Interface Segregation Principle

GROUP K6

```java
package k6.gatherlove.ArticleManagement.service;

import k6.gatherlove.ArticleManagement.model.CommentArticle;

import java.util.List;

public interface CommentService {
    CommentArticle addComment(Long articleId, String username, String content);
    List<CommentArticle> getCommentsForArticle(Long articleId);
}
```

```java
@PostMapping("/{id}/comments")
public ResponseEntity<CommentArticle> addComment(
        @PathVariable Long id,
        @RequestParam String username,
        @RequestParam String content) {
    return ResponseEntity.ok(commentService.addComment(id, username, content)); // ✅ fix
}

@GetMapping("/{id}/comments")
public ResponseEntity<List<CommentArticle>> getComments(@PathVariable Long id) {
    return ResponseEntity.ok(commentService.getCommentsForArticle(id)); // ✅ fix
}
```

```java
package k6.gatherlove.ArticleManagement.service;

import k6.gatherlove.ArticleManagement.model.ArticleManagementModel;
import k6.gatherlove.ArticleManagement.model.CommentArticle;
import k6.gatherlove.ArticleManagement.repository.ArticleManagementRepository;
import k6.gatherlove.ArticleManagement.repository.CommentRepositoryArticle;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class CommentServiceImplArticle implements CommentService {

    private final CommentRepositoryArticle commentRepositoryArticle;
    private final ArticleManagementRepository articleRepository;

    @Override
    public CommentArticle addComment(Long articleId, String username, String content) {
        ArticleManagementModel article = articleRepository.findById(articleId)
                .orElseThrow(() -> new RuntimeException("Article not found"));
        CommentArticle commentArticle = CommentArticle.builder()
                .username(username)
                .content(content)
                .article(article)
                .build();
        return commentRepositoryArticle.save(commentArticle);
    }

    @Override
    public List<CommentArticle> getCommentsForArticle(Long articleId) {
        return commentRepositoryArticle.findByArticleId(articleId);
    }
}
```

ADVANCED PROGRAMMING - 2024

Dependency Inversion Principle

GROUP K6

We applied the Dependency Inversion Principle by separating the business logic into dedicated service classes. These services are then called by the controllers.

The main advantage of this approach is that if there are any changes to the business logic, we only need to update the service class—without modifying the controller. This promotes better modularity, maintainability, and decoupling between the components.

ADVANCED PROGRAMMING - 2024

Dependency Inversion Principle

GROUP K6

```java
@Override
@Transactional
public Donation createDonation(String userId, double amount, String campaignId) {
    walletClient.deduct(userId, amount);

    if (amount > userBalance) {
        throw new IllegalStateException("Insufficient balance to create donation!");
    }
    userBalance -= amount;

    Donation donation = new Donation(userId, amount, campaignId);
    donation.setConfirmed(true);
    Donation saved = repo.save(donation);

    campaignClient.recordDonation(campaignId, saved.getId(), amount);

    metricsService.incrementDonationCreated();

    return saved;
}
```

```java
@PostMapping
public Donation create(@RequestBody DonationRequest req) {
    return donationService.createDonation(
            req.getUserId(),
            req.getAmount(),
            req.getCampaignId()
    );
}
```

ADVANCED PROGRAMMING - 2024

GROUP K6

Components

Software Design

- Design Pattern
- SOLID Principle
- Maintainability

Software Quality

- Clean Code
- Secure Coding
- Testing
- Profiling

Software Architecture

- Concurrency
- Asynchronous
- High-Level Networking

Software Deployment

- CI/CD
- Automated Deployment
- Containerization
- Infrastructure as Code
- Monitoring

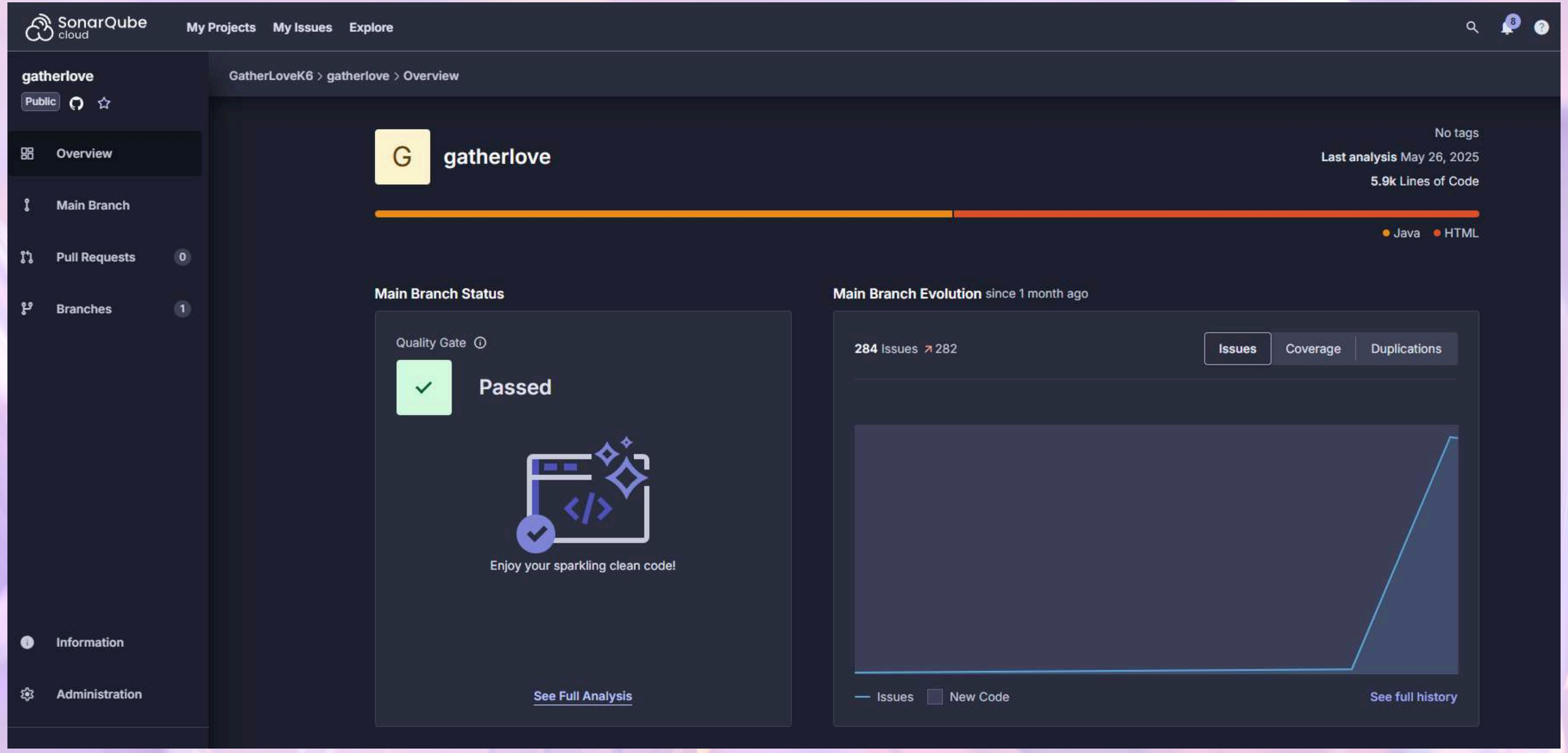
GROUP K6

CLEAN CODE & SECURE CODING

Proyek Grup 2024

SonarCloud

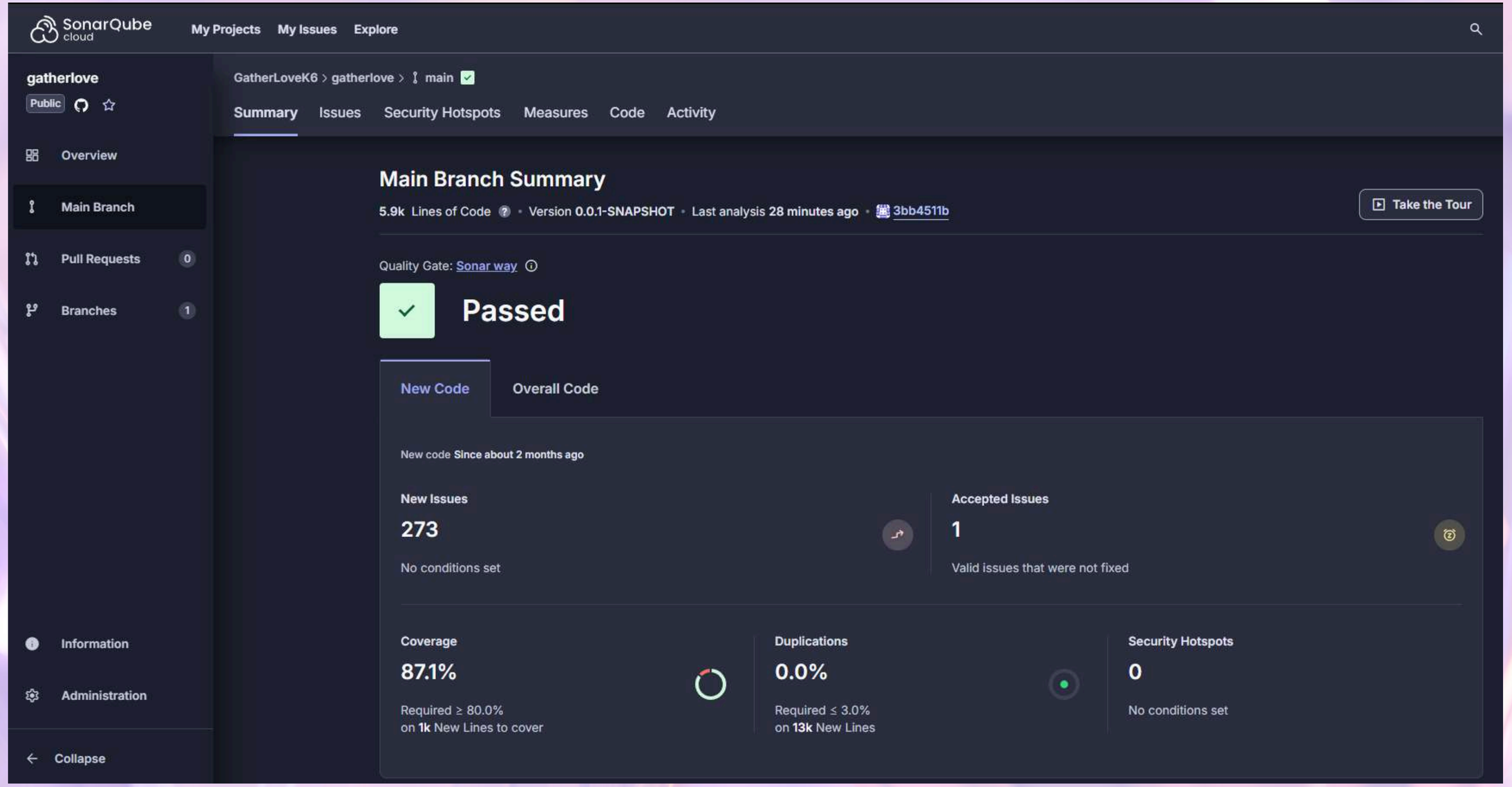
GROUP K6



ADVANCED PROGRAMMING - 2024

SonarCloud

GROUP K6



ADVANCED PROGRAMMING - 2024

GROUP K6

TESTING

Proyek Grup 2024

JacocoTestReport

GROUP K6

gatherlove

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
k6.gatherlove.donation.client		16%		n/a	4	6	14	22	4	6	0	2
k6.gatherlove.wallet.domain		76%		100%	12	35	12	60	12	33	0	4
k6.gatherlove.wallet.service		83%		83%	5	18	7	56	4	15	0	2
k6.gatherlove.donation.service		78%		60%	6	16	8	44	2	11	0	2
k6.gatherlove.controller		20%		0%	3	6	9	12	1	4	0	2
k6.gatherlove.wallet.exception		9%		n/a	4	5	8	9	4	5	1	2
k6.gatherlove.fundraising.service		91%		76%	5	36	5	63	0	21	0	2
k6.gatherlove.AdminDashboard.service		81%		50%	4	19	8	37	3	18	0	6
k6.gatherlove.auth.controller		78%		75%	2	7	9	34	1	5	0	1
k6.gatherlove.donation.controller		68%		100%	5	13	10	29	5	12	0	3
k6.gatherlove.donation.config		34%		0%	3	5	5	9	1	3	0	1
k6.gatherlove.ArticleManagement.service		83%		n/a	2	9	2	30	2	9	0	2
k6.gatherlove.auth.service		94%		100%	0	9	2	26	0	5	0	1
k6.gatherlove.report.factory		79%		80%	1	5	1	10	0	2	0	1
k6.gatherlove.fundraising.exception		44%		n/a	1	2	2	4	1	2	0	1
k6.gatherlove.fundraising.validator		95%		100%	1	11	1	12	1	2	0	1
k6.gatherlove.fundraising.factory		94%		n/a	1	3	1	19	1	3	0	1
k6.gatherlove.auth.filter		96%		80%	2	6	0	11	0	1	0	1
k6.gatherlove.fundraising.controller		99%		85%	8	50	0	142	0	22	0	2
k6.gatherlove.report.controller		100%		100%	0	14	0	76	0	10	0	1
k6.gatherlove.wallet.controller		100%		n/a	0	14	0	39	0	14	0	2
k6.gatherlove.auth.util		100%		n/a	0	6	0	33	0	6	0	1
k6.gatherlove.report.service		100%		n/a	0	12	0	31	0	12	0	2
k6.gatherlove		100%		100%	0	6	0	31	0	4	0	2
k6.gatherlove.auth.config		100%		n/a	0	7	0	15	0	7	0	2
k6.gatherlove.auth.strategy		100%		100%	0	4	0	15	0	3	0	1
k6.gatherlove.ArticleManagement.controller		100%		n/a	0	7	0	10	0	7	0	1
k6.gatherlove.fundraising.model		100%		50%	1	4	0	14	0	3	0	2
k6.gatherlove.AdminDashboard.controller		100%		n/a	0	7	0	11	0	7	0	6
k6.gatherlove.donation.model		100%		n/a	0	4	0	17	0	4	0	2
k6.gatherlove.wallet.dto		100%		n/a	0	13	0	13	0	13	0	3
k6.gatherlove.auth.model		100%		n/a	0	9	0	12	0	9	0	2
k6.gatherlove.wallet.config		100%		n/a	0	2	0	8	0	2	0	1
k6.gatherlove.report.enums		100%		n/a	0	1	0	3	0	1	0	1
k6.gatherlove.user		100%		n/a	0	1	0	3	0	1	0	1
k6.gatherlove.donation.dto		100%		n/a	0	2	0	2	0	2	0	2
k6.gatherlove.report.dto		100%		n/a	0	1	0	1	0	1	0	1
k6.gatherlove.AdminDashboard.dto		100%		n/a	0	1	0	1	0	1	0	1
Total	456 of 3,763	87%	34 of 179	81%	70	376	104	964	42	286	1	71

ADVANCED PROGRAMMING - 2024

GROUP K6

Components

Software Design

- Design Pattern
- SOLID Principle
- Maintainability

Software Quality

- Clean Code
- Secure Coding
- Testing
- Profiling

Software Architecture

- Concurrency
- Asynchronous
- High-Level Networking

Software Deployment

- CI/CD
- Automated Deployment
- Containerization
- Infrastructure as Code
- Monitoring

MONOLITHIC

GROUP K6

We implemented a monolithic architecture, where all features and functionalities are developed and deployed as a single, unified application.
Our monolith includes multiple modules:

- Admin Dashboard
- Article Management
- auth
- donation
- fundraising
- report
- wallet

Unlike microservices, all modules in our application share the same codebase and are deployed together. Additionally, we use a single shared database, allowing direct communication between different parts of the application without the need for inter-service APIs.
This approach simplifies development and deployment, especially in early-stage projects, while still allowing us to maintain modular code organization within the monolith.

ADVANCED PROGRAMMING - 2024

High-Level Networking

GROUP K6

Our team implemented REST API in designing the Controllers to apply the concept of High-Level Networking.

```java
@RestController
@RequestMapping("/api/fundraising")
@RequiredArgsConstructor
@Slf4j
public class CampaignController {

    private final CampaignService campaignService;
    private final FileStorageService fileStorageService;

    // Updated to handle authentication more robustly
    private Long getAuthenticatedUserId() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null || !authentication.isAuthenticated() ||
            "anonymousUser".equals(authentication.getPrincipal())) {
            throw new ValidationException("User not authenticated");
        }

        String userId = authentication.getName();
        log.debug("Raw authenticated user ID: {}", userId);

        try {
            // Try to parse as Long directly
            return Long.parseLong(userId);
        } catch (NumberFormatException e) {
            // If not a direct number, it might be a UUID or complex ID
            // Generate a consistent numeric hash for this session
            // This is a workaround - in production you'd want proper ID mapping
            long hashedId = (long) Math.abs(userId.hashCode());
            log.debug("Converted user ID {} to numeric ID {}", userId, hashedId);
            return hashedId;
        }
    }

    // Helper method to check if user is authenticated without throwing exception
    private boolean isUserAuthenticated() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        return authentication != null && authentication.isAuthenticated() &&
               !"anonymousUser".equals(authentication.getPrincipal());
    }
```

ADVANCED PROGRAMMING - 2024

Asynchronous

GROUP K6

We utilized asynchronous programming in both the front-end and back-end components of our project to improve execution speed and responsiveness.

On the front-end, asynchronous behavior was implemented using async-await.

```
async function deleteArticle() {  Show usages  ksmratih
    const id = document.getElementById("deleteId").value;
    const response = await fetch(`/articles/${id}`, {
        method: "DELETE"
    });
    document.getElementById("deleteStatus").innerText = await response.text();
}

async function likeArticle(id) {  Show usages  ksmratih
    const response = await fetch(`/articles/${id}/like`, { method: "POST" });
    if (response.ok) {
        alert("Article liked!");
        getAllArticles();
    } else {
        alert("Failed to like article.");
    }
}
```

ADVANCED PROGRAMMING - 2024

Asynchronous

GROUP K6

On the back-end, we used Spring Boot's built-in @Async annotation to handle asynchronous tasks efficiently. This helps ensure the main thread is not blocked during execution, improving system responsiveness.

```java
@Async("taskExecutor") 3 usages  andriyoaverill +1
@Transactional
public CompletableFuture<Transaction> topUpAsync(String userId, BigDecimal amount, String pmCode) {
    Timer.Sample sample = metricsService.startWalletTransactionTimer();

    try {
        pmService.validatePaymentMethod(userId, pmCode);

        Wallet wallet = getOrCreateWallet(userId);
        wallet.topUp(amount);
        walletRepo.save(wallet);

        Transaction tx = new Transaction(
            UUID.randomUUID().toString(),
            wallet.getWalletId(),
            TransactionType.TOP_UP,
            amount
        );
        tx.markCompleted();
        Transaction saved = txRepo.save(tx);

        return CompletableFuture.completedFuture(saved);
    } finally {
        metricsService.recordWalletTransaction(sample);
    }
}
```

ADVANCED PROGRAMMING - 2024

GROUP K6

Components

Software Design

- Design Pattern
- SOLID Principle
- Maintainability

Software Quality

- Clean Code
- Secure Coding
- Testing
- Profiling

Software Architecture

- Concurrency
- Asynchronous
- High-Level Networking

Software Deployment

- CI/CD
- Automated Deployment
- Containerization
- Infrastructure as Code
- Monitoring

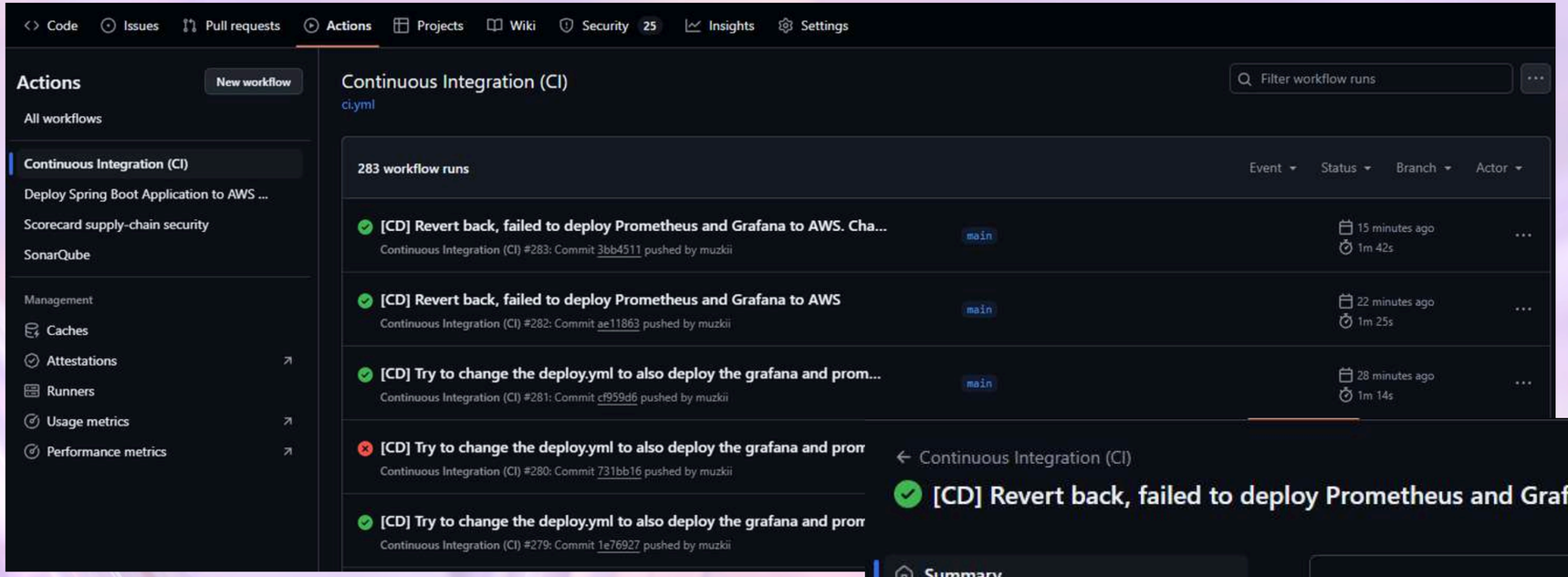
GROUP K6

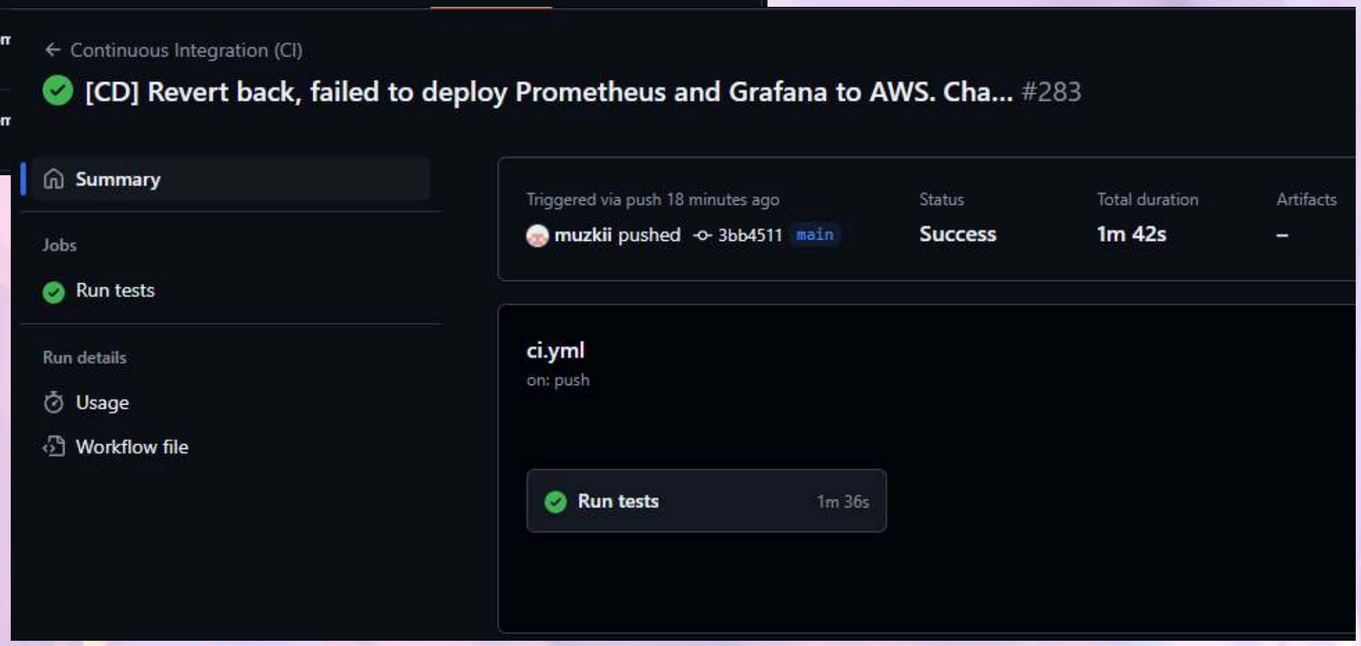
CI/CD

Proyek Grup 2024

CI-CD : ci.yml

GROUP K6

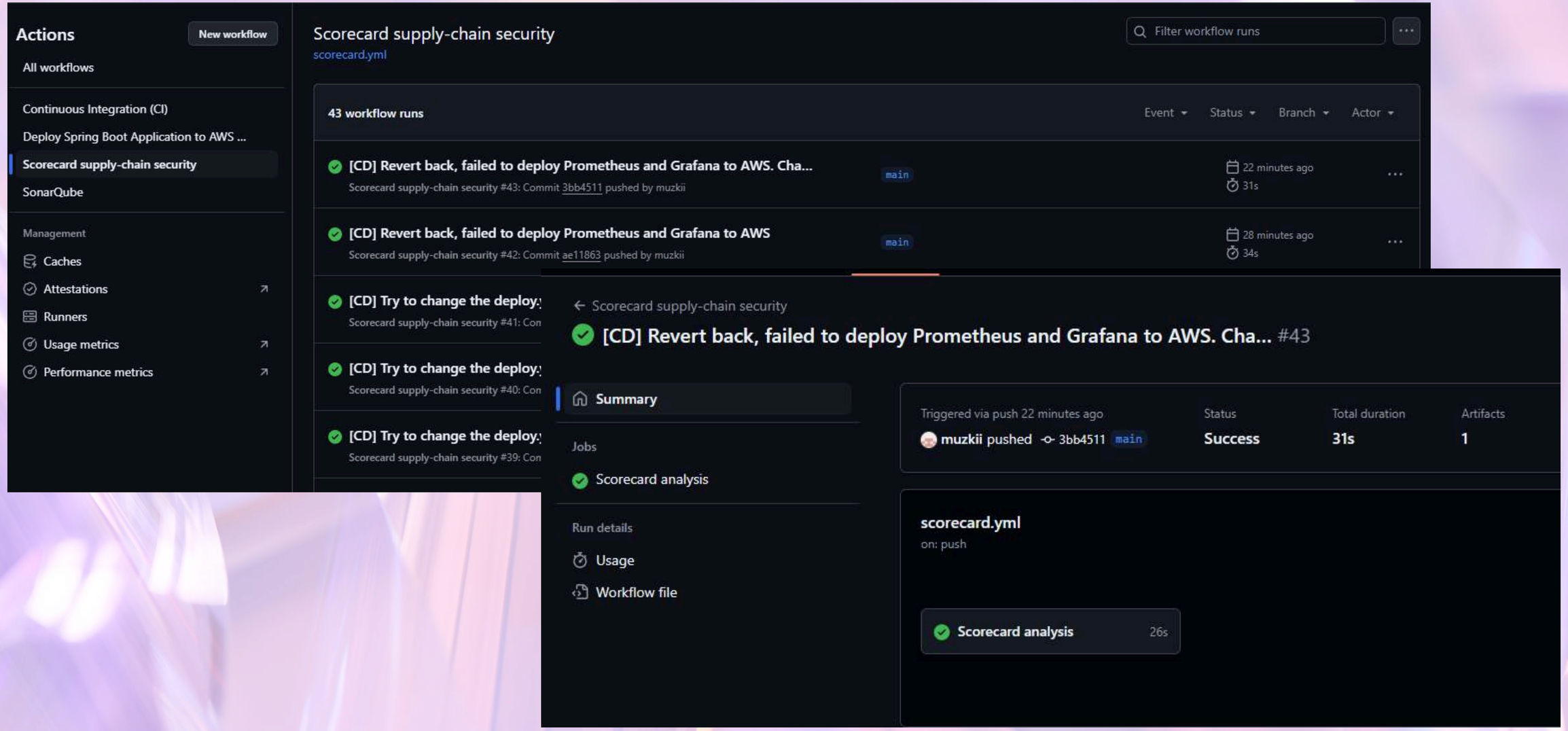




ADVANCED PROGRAMMING - 2024

CI-CD : scorecard.yml

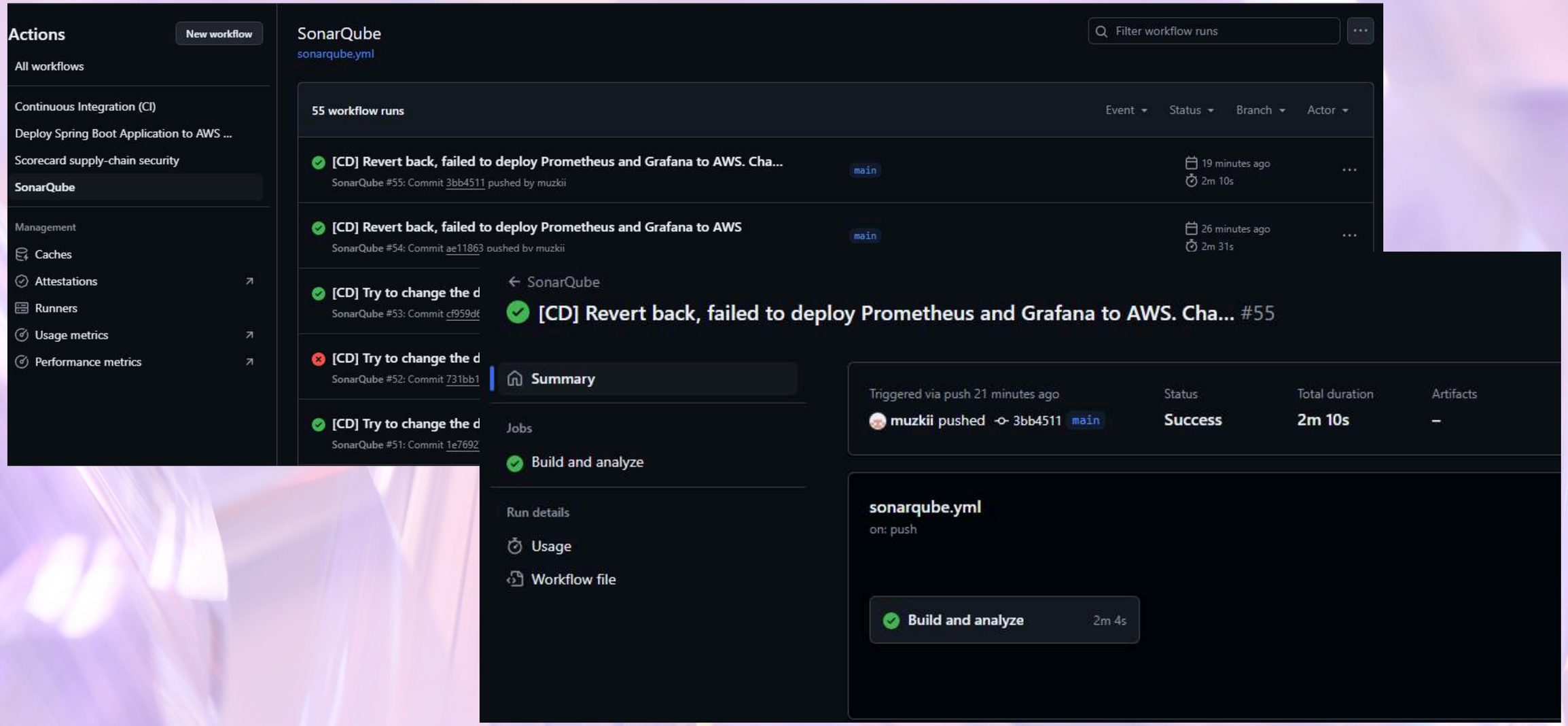
GROUP K6



ADVANCED PROGRAMMING - 2024

CI-CD : sonarqube.yml

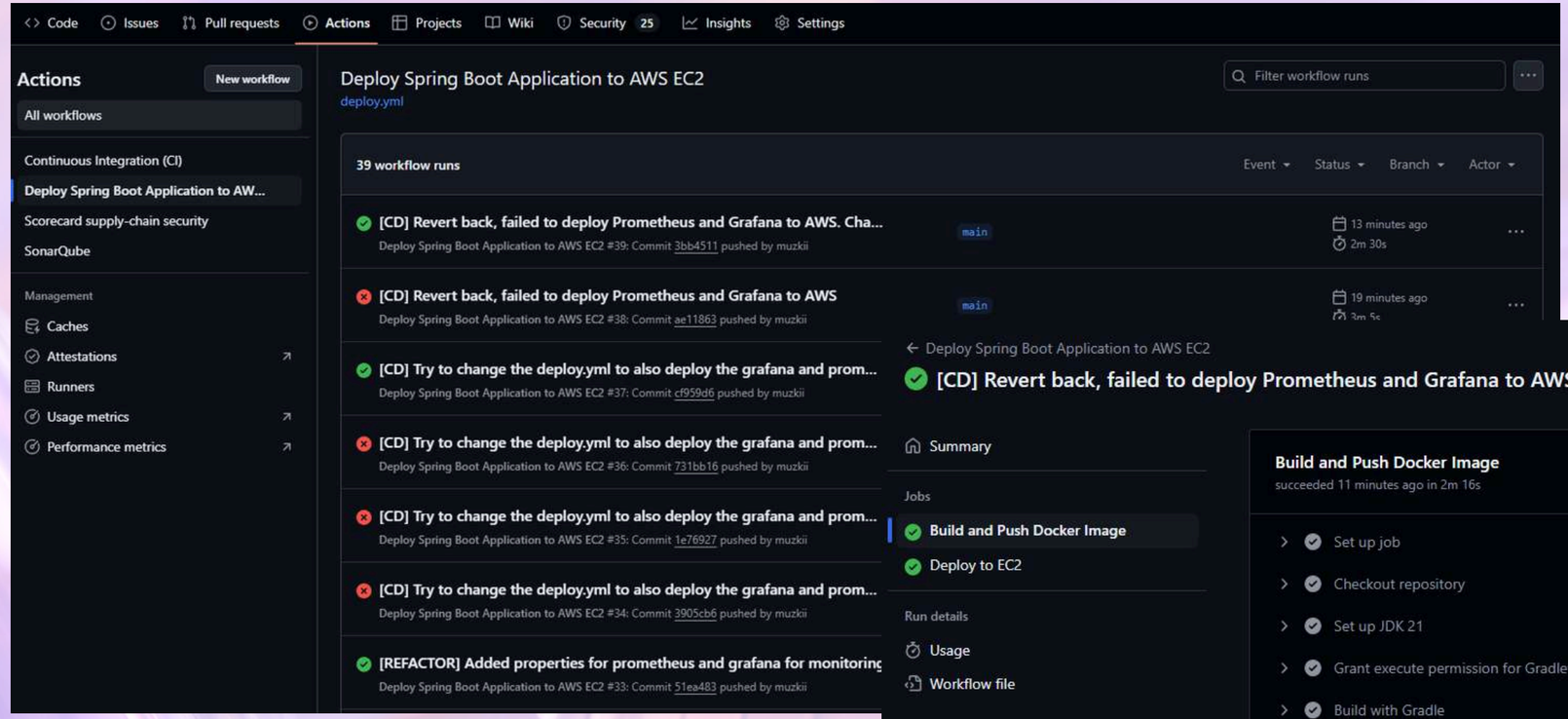
GROUP K6

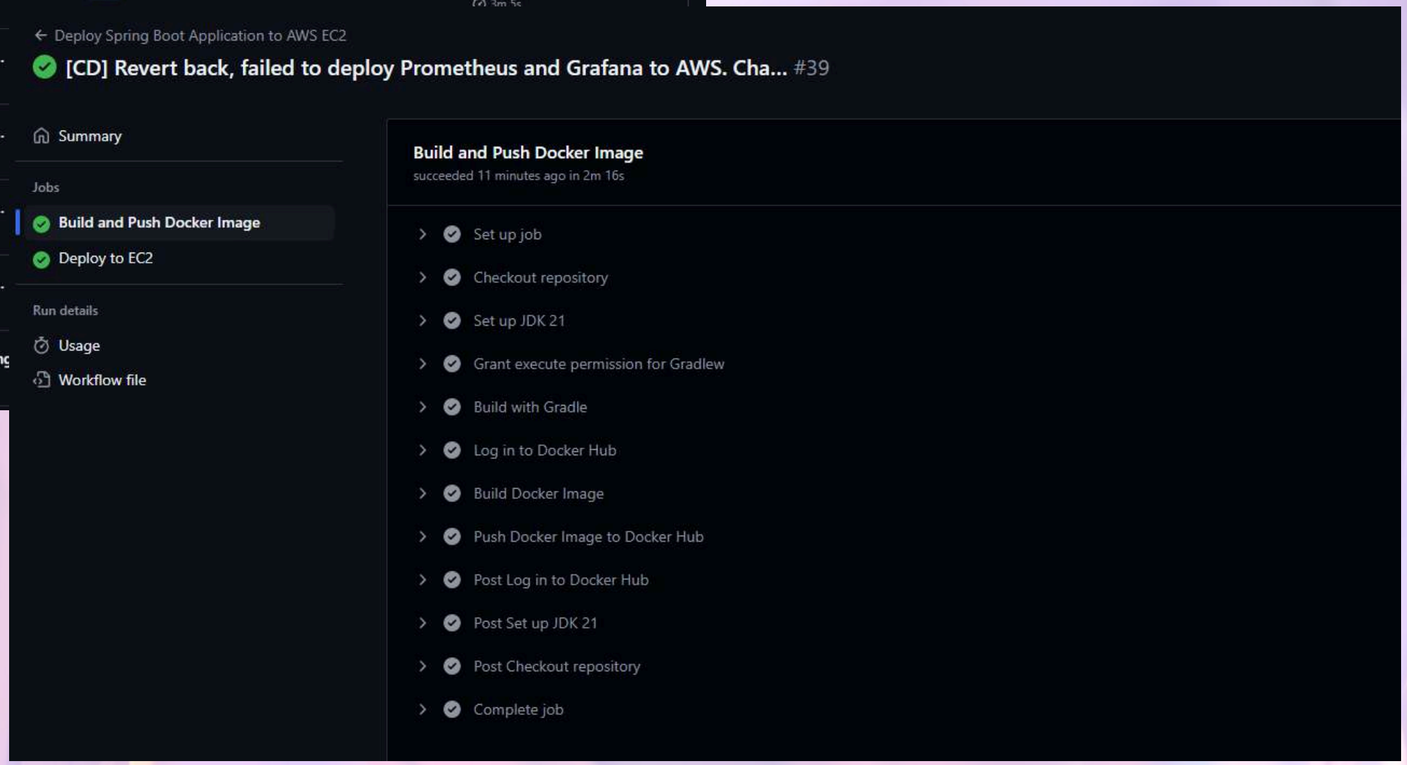


ADVANCED PROGRAMMING - 2024

CI-CD : deploy.yml

GROUP K6





ADVANCED PROGRAMMING - 2024

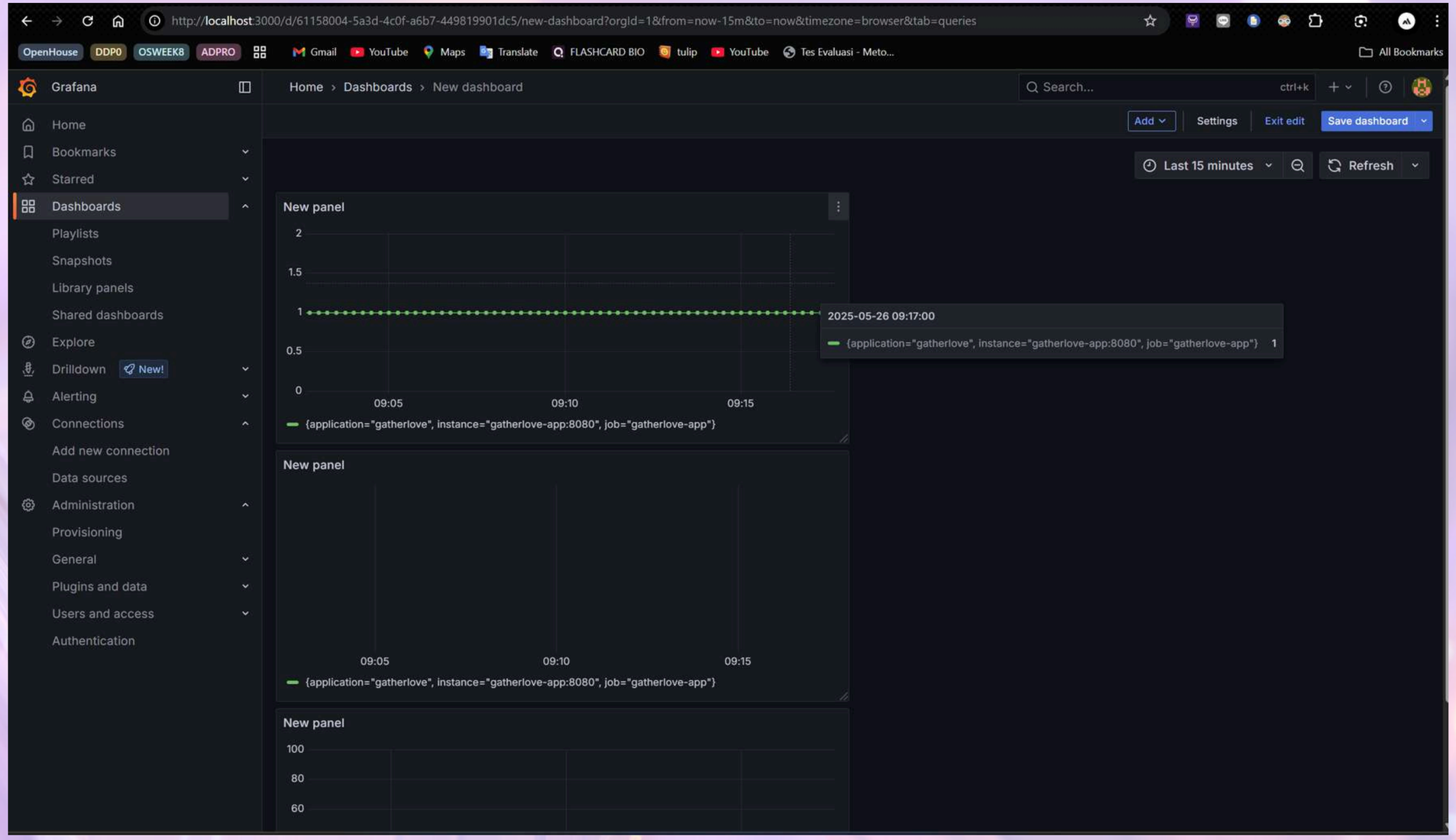
GROUP K6

MONITORING

Proyek Grup 2024

Promotheus & Grafana

GROUP K6



ADVANCED PROGRAMMING - 2024

GROUP K6

GITHUB REPOSITORY

https://github.com/GatherLoveK6/gatherlove

Proyek Grup 2024

ADVANCED PROGRAMMING - 2024

THANK YOU

Thank you for your attention. We hope that our presentation is clear, concise, and beneficial to all.

GROUP K6

GROUP K6

Q&A SESSION

ADVANCED PROGRAMMING - 2024