Software Architecture and Design Report

Group:
2025-2026-ADS-Grupo _International-A

Members:
Ilan Federico Piczenik Acher
Alan Skorka Shuster
Kyan Philippe Jamshid Zeynali

Professor:
José Pereira dos Reis

Repository link:
https://github.com/kyan17/gitstats

Table of Contents **2**
3. INTRODUCTION **3**
3.1 Project Context 3
3.2 Objectives 4
3.3 Scope 4
4. SCRUM PROJECT MANAGEMENT **5**
4.1 SCRUM Framework Adoption 5
4.2 Team Roles and Responsibilities 5
4.3 SCRUM Artifacts 5
4.4 SCRUM Events 6
4.5 Sprint Organization 6
4.6 Project Management Tools 7
4.7 Definition of Ready (DoR) and Definition of Done (DoD) 7
5. SYSTEM REQUIREMENTS **8**
5.1 Functional Requirements 8
5.2 Non-Functional Requirements 9
5.3 Use Cases 9
6. SYSTEM ARCHITECTURE **9**
6.1 Architecture Overview 9
6.2 Technology Stack 10
6.3 High-Level Architecture 10
6.4 Backend Architecture 10
6.5 Frontend Architecture 10
6.6 Authentication Flow 11
6.7 GitHub API Integration 11
7. DETAILED DESIGN AND IMPLEMENTATION **11**
7.1 Class Diagram 11
7.2 Design Patterns 12
7.3 Project Structure 13
7.4 Security Implementation 13
7.5 Key Implementation Details 14
8. CONCLUSIONS **14**
8.1 Objectives Achievement 14
8.2 Lessons Learned 15
8.3 Challenges Encountered 15

3. INTRODUCTION

3.1 Project Context

Modern software development relies heavily on collaborative platforms like GitHub for version control and team coordination. While GitHub provides basic repository statistics, it lacks comprehensive analytical tools for detailed individual contributor metrics. This project, **GitStats**, addresses this gap by developing a web application that integrates with the GitHub API to generate personalized dashboards for repository contributors. The application enables authenticated users to visualize metrics such as commits, code changes, issues, pull requests, and activity timelines through interactive charts and graphs. This project demonstrates practical API integration and represents a real-world case of systems integration through REST APIs.

3.2 Objectives

The main objective is to develop a complete web application integrated with the GitHub API that generates personalized dashboards for authenticated users to analyze contributor activity in GitHub repositories.

Specific objectives include: implementing secure GitHub OAuth authentication; enabling repository selection and exploration; automatically identifying repository contributors; generating individual metrics (commits, lines added/removed, issues, pull requests); implementing interactive graphical visualizations (bar charts, pie charts, timelines, network graphs); designing a responsive and intuitive user interface; applying SCRUM methodology for project management; and documenting the system using standard notations (UML, BPMN).

3.3 Scope

The project implements authentication via GitHub OAuth, repository listing and selection, contributor identification, individual metrics per contributor (commits by period, lines added/removed, activity tracking), temporal timelines (commits, issues, pull requests), graphical visualizations (bar charts, pie charts, line graphs, network graphs), language distribution analysis, and a fully responsive user interface.

The system is limited to public repositories with OAuth scopes restricted to `read:user` and `public_repo`. Metrics are calculated at the commit level without code quality analysis. The system is subject to GitHub API rate limits (5,000 requests/hour) and does not implement persistent data caching. Timelines are limited to 30 days, 12 weeks, or 12 months depending on granularity. Out of scope features include predictive analytics, integration with other platforms, offline mode, real-time notifications, multi-platform support, and report exporting.

4. SCRUM PROJECT MANAGEMENT

4.1 SCRUM Framework Adoption

The team adopted the SCRUM methodology to manage project development in an iterative and incremental manner. Work was organized into two-week sprints with continuous collaboration and regular feedback loops. SCRUM was chosen for its flexibility, transparency, and ability to deliver working software incrementally, which aligned well with the project's API integration nature and evolving requirements.

4.2 Team Roles and Responsibilities

👥 Team Roles

Member	SCRUM Role	Responsibilities
Ilan Piczenik	Product Owner	Defines product vision, prioritizes backlog items, ensures value delivery.
Kyan Jamshid	Scrum Master	Facilitates SCRUM processes, resolves blockers, ensures coordination.
Alan Skorka	Developer	Implements functionality, maintains code quality, supports testing and documentation.

4.3 SCRUM Artifacts

Product Backlog
Maintained a prioritized list of all features and requirements including authentication, repository management, metrics calculation, visualizations, and UI improvements. User stories were written with clear acceptance criteria and estimated by the team.

Sprint Backlog
For each sprint, the team selected high-priority items from the Product Backlog and committed to delivering them. Tasks were broken down into manageable work items and tracked through Trello.

4.4 SCRUM Events

Sprint Planning
Held at the beginning of each sprint to select backlog items, define the sprint goal, and estimate effort. The team collectively decided on the sprint scope based on capacity and priority.

Sprint Review
Conducted at sprint end to demonstrate completed work to stakeholders, gather feedback, and validate that acceptance criteria were met.

Sprint Retrospective
Team reflection session to identify what went well, what could improve, and action items for the next sprint.

4.5 Sprint Organization

Sprint 1 – Authentication & Repository Listing
Duration: 1 month
Goal: Implement secure authentication and enable repository exploration

User Stories Completed:
- [US-01] GitHub Authentication (OAuth login)
- [US-02] Repository List for Authenticated User
- [US-03] Repository Details View (basic info)
- [US-04] Logout from App and GitHub

Key Deliverables:
- OAuth 2.0 integration with GitHub
- Secure token management with Spring Security
- Repository listing with sorting by last update
- Basic repository information display

Sprint 2 – Metrics & Visualizations
Duration: 1 month
Goal: Implement contributor metrics and interactive dashboards

User Stories Completed:
- [US-05] Show contributor metrics (commits, PRs, issues)
- [US-06] Display graphical charts (bar, line, pie)
- [US-07] Improve frontend UI/UX
- [DOC-02] Project README documentation

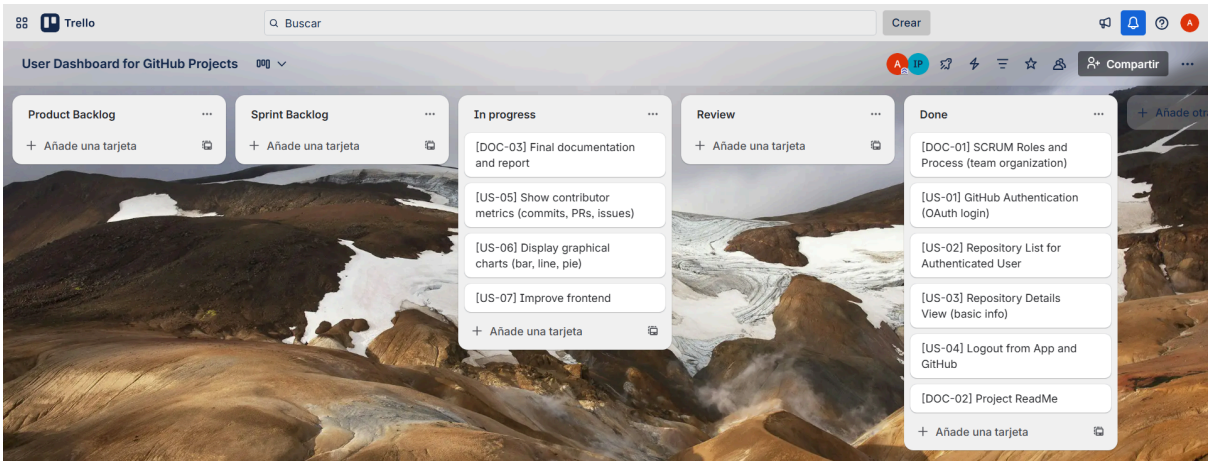
Key Deliverables:
- Individual contributor statistics (all-time, monthly, weekly)
- Commit, issues, and pull requests timelines
- Bar chart for commit comparison
- Pie chart for contribution percentage
- Network graph visualization
- Language distribution chart
- Responsive design implementation

4.6 Project Management Tools

Trello Board: https://trello.com/b/dCj0BVcA/user-dashboard-for-github-projects

The team used Trello with the following columns to track work:
- Product Backlog: All identified features and requirements
- Sprint Backlog: Items committed for the current sprint
- In Progress: Work currently being developed
- Review: Completed work awaiting code review or testing
- Done: Fully completed and integrated features

Labels were used to categorize tasks: frontend, `backend`, `docs`, `bug`, `enhancement`.



4.7 Definition of Ready (DoR) and Definition of Done (DoD)

Definition of Ready:
A user story is ready for sprint planning when it has a clear description, defined acceptance criteria, is properly estimated, dependencies are identified, and the team understands the requirements.

Definition of Done:
A user story is considered done when it meets all acceptance criteria, functionality is working on the main branch, code is reviewed and tested, documentation is updated, and the Product Owner has accepted the work.

5. SYSTEM REQUIREMENTS

5.1 Functional Requirements

RF01: Implement GitHub OAuth 2.0 authentication with secure token management and session handling.

RF02: Enable repository selection by retrieving the authenticated user's public repositories from GitHub API and displaying them in a sortable list.

RF03: Automatically identify all contributors in a selected repository using the GitHub contributors endpoint.

RF04: Generate individual contributor metrics including total commits (all-time, monthly, weekly), lines added/removed, issues opened/closed, pull requests created/merged, and recent activity timeline.

RF05: Provide interactive visualizations through bar charts comparing commits per contributor, pie charts showing contribution percentages, line charts displaying temporal evolution of commits/issues/PRs, network graphs for branch and commit visualization, and language distribution charts.

RF06: Design a responsive user interface that adapts to desktop, tablet, and mobile devices with intuitive navigation and clear visual feedback.

5.2 Non-Functional Requirements

RNF01: Security - OAuth 2.0 implementation, secure token storage, session management with Spring Security, HTTPS support.

RNF02: Usability - Clean interface, intuitive navigation, loading indicators, error messages, consistent design patterns.

RNF03: Performance - Efficient API calls with pagination, responsive UI with asynchronous data loading, optimized rendering of charts.

RNF04: Scalability - Modular architecture supporting feature additions, stateless backend design, configurable API limits.

RNF05: Maintainability - Clean code structure, separation of concerns, comprehensive DTO layer, documented API endpoints.

5.3 Use Cases

The main use cases include: User authenticates via GitHub OAuth, User browses personal repositories, User selects a specific repository, System displays repository overview with general metrics, User views contributor list with commit statistics, User explores temporal timelines for commits/issues/pull requests, User analyzes language distribution, User examines network graph of branches and commits, and User logs out from application.

6. SYSTEM ARCHITECTURE

6.1 Architecture Overview

The system follows a client-server architecture pattern with clear separation between frontend and backend layers. The frontend is a single-page application built with React that communicates with the backend through REST APIs. The backend, built with Spring Boot, handles authentication, GitHub API integration, and business logic. The architecture supports stateless operations with OAuth token-based authentication.

6.2 Technology Stack

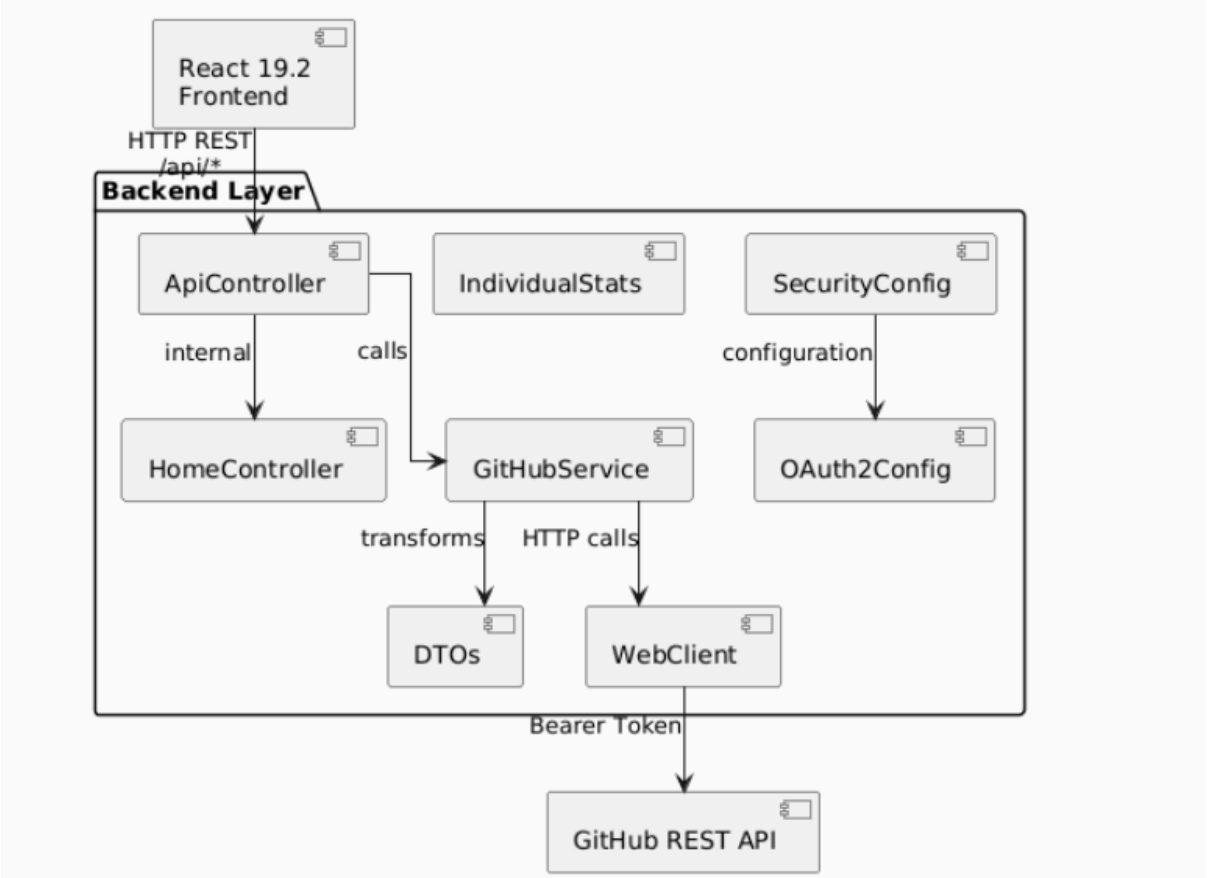
Backend: Spring Boot 3.5.6, Java 25, Spring Security with OAuth2 Client, Spring WebFlux with reactive WebClient, Jetty HTTP Client, Maven for dependency management.

Frontend: React 19.2, TypeScript, Vite build tool, Chart.js with react-chartjs-2 for visualizations, CSS for styling.

Integration: GitHub REST API v3, GitHub OAuth 2.0.

6.3 High-Level Architecture

The system consists of three main layers: Presentation Layer (React frontend components), Application Layer (Spring Boot controllers and services), and Integration Layer (GitHub API client). Data flows from user interactions through the frontend to backend REST endpoints, which authenticate requests and communicate with GitHub API, then transform responses into DTOs and return them to the frontend for visualization.



6.4 Backend Architecture

The backend follows a layered architecture with Controllers (ApiController, HomeController) handling HTTP requests, Services (GitHubService, IndividualStats) implementing business logic and API integration, Security layer (SecurityConfig, OAuth2AuthorizedClientService) managing authentication, and DTOs (Repository, Contributor, CommitStats, CommitTimeline, IssuesTimeline, PullRequestsTimeline, NetworkGraph, LanguageStats) transferring data between layers.

6.5 Frontend Architecture

The frontend uses component-based architecture with routing managed by a custom Routes module. Main components include App (root component with routing logic), RepoListView (displays user repositories), RepoDetailsView (shows repository metrics and visualizations), CommitTimelineView, IssuesTimelineView, PullRequestsTimelineView (temporal charts), NetworkView (branch visualization), LanguagesView (language distribution), and PostLogoutView (post-logout state).

6.6 Authentication Flow

The OAuth flow sequence: User clicks login, application redirects to GitHub authorization page, user grants permissions, GitHub redirects back with authorization code, Spring Security exchanges code for access token, token is stored via OAuth2AuthorizedClientService, session is established, user can access protected endpoints, frontend receives session cookie and makes authenticated API calls.

6.7 GitHub API Integration

The system integrates with GitHub REST API using Spring WebClient configured with a 16MB memory buffer for large responses. Key endpoints used include /user/repos for repository listing, /repos/{owner}/{repo}/contributors for contributor data, /repos/{owner}/{repo}/commits for commit history with pagination, /repos/{owner}/{repo}/issues for issue tracking, /repos/{owner}/{repo}/pulls for pull request data, /repos/{owner}/{repo}/languages for language statistics, and /repos/{owner}/{repo}/branches for branch information. The system handles rate limiting (5000 requests/hour) and implements pagination for large datasets.

7. DETAILED DESIGN AND IMPLEMENTATION

7.1 Class Diagram

DIAGRAM - UML Class Diagram showing main backend classes and their relationships:

Key Classes:

GitHubService
- authorizedClientService: OAuth2AuthorizedClientService
- webClient: WebClient
+ getUserRepositories(authentication): List<Repository>
+ getContributors(authentication, owner, repo): List<Contributor>
+ getCommitTimeline(authentication, owner, repo, period): CommitTimeline
+ getIssuesTimeline(authentication, owner, repo, period): IssuesTimeline
+ getPullRequestsTimeline(authentication, owner, repo, period): PullRequestsTimeline
+ getNetworkGraph(authentication, owner, repo, maxCommits): NetworkGraph
+ getLanguages(authentication, owner, repo): List<LanguageStats>
- getAccessToken(authentication): String

ApiController
- gitHubService: GitHubService
+ me(principal): ResponseEntity
+ repositories(authentication, principal): ResponseEntity
+ contributors(authentication, principal, owner, repo): ResponseEntity
+ commitTimeline(authentication, principal, owner, repo, period): ResponseEntity

SecurityConfig
+ filterChain(http, introspector): SecurityFilterChain
+ authorizedClientService(repository): OAuth2AuthorizedClientService

IndividualStats
+ getCommitStats(token, webClient, owner, repo, login, period): CommitStats

DTOs: Repository, Contributor, CommitStats, CommitTimeline, IssuesTimeline, PullRequestsTimeline, NetworkGraph, LanguageStats, TimelinePoint, IssuesTimelinePoint, PullRequestsTimelinePoint, CommitNode, BranchInfo


CommitPeriod
CommitPeriod()
values() CommitPeriod[]
valueOf(String) CommitPeriod
ContributionSlice
ContributionSlice(String, long)
login() String
score() long
IssuesTimelinePoint
IssuesTimelinePoint(String, int, int)
label() String
opened() int
closed() int
TimelinePoint
TimelinePoint(String, int)
label() String
count() int
GitstatsApplication
GitstatsApplication()
main(String[]) void
NoAuthorizedClientException
NoAuthorizedClientException(String)
IssuesTimeline
IssuesTimeline(String, List<IssuesTimelinePoint>, int, int)
points() List<IssuesTimelinePoint>
period() String
totalOpen() int
totalClosed() int
PullRequestsTimelinePoint
PullRequestsTimelinePoint(String, int, int)
merged() int
label() String
opened() int
PullRequestsTimeline
PullRequestsTimeline(String, List<PullRequestsTimelinePoin
period() String
points() List<PullRequestsTimelinePoint>
totalMerged() int
totalOpen() int
CommitTimeline
CommitTimeline(String, List<TimelinePoint>)
period() String
points() List<TimelinePoint>
ContributionStats
ContributionStats(String, String, CommitPeriod, List<Contrib
repo() String
period() CommitPeriod
owner() String
slices() List<ContributionSlice>
HomeController
HomeController()
repoDetailsSpa() String
logoutApp(HttpServletRequest, HttpServletResponse) void
root() String
githubLogout(HttpServletRequest, HttpServletResponse) void
listPage() String
Contributor
Contributor(String, String, String, int)
login() String
contributions() int
htmlUrl() String
avatarUrl() String
NetworkGraph
NetworkGraph(List<BranchInfo>, List<CommitNode>, String)
branches() List<BranchInfo>
defaultBranch() String
commits() List<CommitNode>
CommitNode
CommitNode(String, String, String, String, String, String,
shortSha() String
date() String
branches() List<String>
parentShas() List<String>
authorAvatarUrl() String
message() String
authorLogin() String
sha() String
WorkTypeStats
WorkTypeStats(String, String, CommitPeriod, long, long, lon
refactorCommits() long
testCommits() long
owner() String
documentationCommits() long
repo() String
featureCommits() long
period() CommitPeriod
bugfixCommits() long
SecurityConfig
SecurityConfig()
filterChain(HttpSecurity) SecurityFilterChain
authorizedClientService(ClientRegistrationRepository) OAuth2AuthorizedClientService
WorkType
WorkType()
valueOf(String) WorkType
values() WorkType[]
LanguageStats
LanguageStats(String, long, double, String)
bytes() long
color() String
percentage() double
name() String
IndividualStats
IndividualStats()
urlEncode(String) String
valueOf(String) IndividualStats
fetchCommitsForContributor(String, WebClient, String, String, String, Of
fetchCommitDetails(String, WebClient, String, String, String) JsonNode
periodToSince(CommitPeriod) OffsetDateTime?
fetchCommitsPaged(String, WebClient, String, String, String, OffsetDate
parseGithubDate(String, DateTimeFormatter) OffsetDateTime
fetchDefaultBranch(String, WebClient, String, String) String?
collectIssueAndPrStats(String, WebClient, String, String, String, OffsetDa
estimateLanguagesCount(Set<String>) int
getCommitStats(String, WebClient, String, String, String, CommitPeriod)
values() IndividualStats[]
GitHubService
GitHubService(OAuth2AuthorizedClientService)
getIssuesTimeline(OAuth2AuthenticationToken, String, String, String) esTimeline
getNetworkGraph(OAuth2AuthenticationToken, String, String, int) NetworkGraph
getLanguageColor(String) String
getContributors(OAuth2AuthenticationToken, String, String) List<Contributor>
getCommitTimeline(OAuth2AuthenticationToken, String, String, String) itTimeline
getWorkTypeStats(OAuth2AuthenticationToken, String, String, CommitPeriod) ts
getAllTimeStats(OAuth2AuthenticationToken, String, String, String) CommitStats
convertToRepository(JsonNode) Repository
getAccessToken(OAuth2AuthenticationToken) String
getLastMonthStats(OAuth2AuthenticationToken, String, String, String) mmitStats
getUserRepositories(OAuth2AuthenticationToken) List<Repository>
getPullRequestsTimeline(OAuth2AuthenticationToken, String, String, String) eline
getLastWeekStats(OAuth2AuthenticationToken, String, String, String) mmitStats
getLanguages(OAuth2AuthenticationToken, String, String) List<LanguageStats>
getContributionStats(OAuth2AuthenticationToken, String, String, CommitPeriod)
CommitStats
CommitStats(String, CommitPeriod, int, double, long
totalLinesAdded() long
prsMerged() int
totalLinesDeleted() long
prsOpen() int
topFilesModifiedCount() int
issuesClosed() int
period() CommitPeriod
mainLanguagesCount() int
avgCommitSizeLines() double
netLinesChanged() long
distinctFilesTouched() int
authorLogin() String
prsClosed() int
commitCount() int
issuesOpen() int
Repository
Repository(String, String, String, String, boolean, String, S
isPrivate boolean
name() String
fullName() String
ownerLogin() String
description() String
url() String
updatedAt() String
isPrivate boolean
ApiController
ApiController(GitHubService)
unauthorizedLoginFirst() ResponseEntity<?>
issuesTimeline(OAuth2AuthenticationToken, OAuth2User, String, String, String) ?>
unauthorizedLoginAgain() ResponseEntity<?>
me(OAuth2User) ResponseEntity<?>
contributors(OAuth2AuthenticationToken, OAuth2User, String, String) seEntity<?>
workTypeStats(OAuth2AuthenticationToken, OAuth2User, String, String, String) ?>
commitStatsLastWeek(OAuth2AuthenticationToken, OAuth2User, String, String, Str
repositories(OAuth2AuthenticationToken, OAuth2User) ResponseEntity<?>
isAuthenticated(OAuth2AuthenticationToken, OAuth2User) boolean
internalServerError(String, Exception) ResponseEntity<?>
pullRequestsTimeline(OAuth2AuthenticationToken, OAuth2User, String, String, Strir
commitTimeline(OAuth2AuthenticationToken, OAuth2User, String, String, String) >
networkGraph(OAuth2AuthenticationToken, OAuth2User, String, String, int) ity<?>
contributionStats(OAuth2AuthenticationToken, OAuth2User, String, String, String)
commitStatsLastMonth(OAuth2AuthenticationToken, OAuth2User, String, String, St
languages(OAuth2AuthenticationToken, OAuth2User, String, String) nseEntity<?>
commitStatsAllTime(OAuth2AuthenticationToken, OAuth2User, String, String, Strin
BranchInfo
BranchInfo(String, String, boolean)
isDefault boolean
name() String
sha() String
isDefault boolean

7.2 Design Patterns

Service Layer Pattern: GitHubService encapsulates all business logic and GitHub API communication, providing a clean interface to controllers.

DTO Pattern: Data Transfer Objects (Repository, Contributor, CommitStats, etc.) transfer data between layers without exposing internal representations or GitHub API response structures.

Dependency Injection: Spring framework manages dependencies through constructor injection with @Autowired, promoting loose coupling and testability.

Strategy Pattern: CommitPeriod enum represents different time period strategies (ALL_TIME, LAST_MONTH, LAST_WEEK) for metrics calculation.

Builder Pattern: WebClient is configured using builder pattern with method chaining for flexible HTTP client setup.

Singleton Pattern: Spring manages service beans as singletons, ensuring single instances throughout application lifecycle.

Factory Method Pattern: OAuth2AuthorizedClientService creates authorized client instances based on registration and principal.

7.3 Project Structure

Backend structure follows standard Maven layout:
src/main/java/pt/iscte/se/gitstats/
- GitstatsApplication.java (main entry point)
- SecurityConfig.java (security configuration)
- app/ (controllers and services)
  - ApiController.java
  - HomeController.java
  - GitHubService.java
  - IndividualStats.java
- dto/ (data transfer objects)
- utils/ (exceptions)

Frontend structure follows Vite conventions:
frontend/src/
- App.tsx (root component with routing)

- Api.ts (API client functions)
- Types.ts (TypeScript interfaces)
- Routes.ts (routing logic)
- RepoListView.tsx
- RepoDetailsView.tsx
- CommitTimelineView.tsx
- IssuesTimelineView.tsx
- PullRequestsTimelineView.tsx
- NetworkView.tsx
- LanguagesView.tsx

7.4 Security Implementation

OAuth 2.0 flow is implemented through Spring Security with GitHub as authorization server. SecurityConfig defines security filter chain with public endpoints (/, /index.html, /assets/**, /oauth2/**) and protected endpoints (/api/**, /repositories, /list) requiring authentication. OAuth2AuthorizedClientService stores access tokens in memory associated with user principals. All GitHub API calls include Bearer token in Authorization header retrieved from authenticated session.

7.5 Key Implementation Details

GitHub API Integration: WebClient with reactive programming model handles HTTP requests with 16MB buffer for large responses. Pagination is implemented for endpoints returning multiple pages (commits, issues, pull requests) by iterating until an empty response or less than 100 items per page.

Metrics Calculation: Timeline methods initialize all expected data points with zero counts to ensure consistent chart display even for periods without activity. Data is grouped by day/week/month using DateTimeFormatter with configurable patterns.

Error Handling: NoAuthorizedClientException is thrown when authentication fails or token is unavailable. Controllers catch exceptions and return appropriate HTTP status codes (401 Unauthorized, 500 Internal Server Error) with error messages.

Frontend State Management: React functional components use useState hooks for local state and useEffect for side effects. API calls are made asynchronously and loading states are managed to provide user feedback.

8. CONCLUSIONS

8.1 Objectives Achievement

All project objectives were successfully achieved. The application implements secure GitHub OAuth authentication, enables repository selection and exploration, automatically identifies contributors, generates comprehensive individual metrics including commits by period, lines added/removed, issues and pull requests tracking, provides interactive visualizations through multiple chart types, and delivers a fully responsive user interface. The SCRUM methodology was effectively applied throughout development with two completed sprints delivering incremental value.

8.2 Lessons Learned

Working with external APIs requires careful handling of rate limits, pagination, and response variations. OAuth 2.0 implementation demands thorough understanding of security flows and token management. Reactive programming with Spring WebFlux provides efficient API consumption but requires a different mental model than traditional blocking calls. React with TypeScript improves code quality through static typing but adds development overhead. SCRUM ceremonies provide structure and visibility but require discipline and consistent participation from all team members.

8.3 Challenges Encountered

GitHub API rate limits (5000 requests/hour) necessitated efficient query design and consideration for caching strategies. Calculating accurate lines added/removed metrics required understanding GitHub's commit data structure and handling edge cases for merge commits. OAuth token lifecycle management needed careful implementation to prevent expired tokens and maintain session security. Handling asynchronous data loading in frontend while maintaining responsive UI required proper loading state management. Coordinating sprint planning and execution within academic timeline constraints demanded effective time management.