Politecnico di Milano
Computer Science and Engineering
Software Engineering II

Design Document - CodeKataBlade

José Alejandro Sarmiento

January 7, 2024

v1.0

Contents

1 INTRODUCTION

1.1 Purpose

The purpose of this document is to provide a detailed description of the architecture of the system to be, CodeKataBlade, and to show how the requirements presented in the RASD document are met by the architecture. The document also contains the implementation, integration and test plan for the system.

1.2 Scope

CodeKataBlade is a software system designed to facilitate and enhance the practice of code katas, which are small coding exercises aimed at improving programming skills. The system provides a platform where educators can create tournaments and battles, and students can register, form teams, and submit their solutions to coding problems.

CodeKataBlade allows educators to create tournaments, which serve as spaces for organizing multiple battles. Educators can invite other educators to participate in tournaments and perform manual evaluations on the submissions of battles they own. The system also provides a leaderboard that is updated in real-time, displaying the rankings of participants based on their performance in the battles.

Students can register to tournaments and battles, either individually or as part of a team. They can submit their solutions to coding problems within the specified deadlines. The system automatically evaluates the submissions using build automation scripts and provides feedback to the students. A set of test cases is used to assess the correctness and efficiency of the solutions.

CodeKataBlade integrates with GitHub, a popular code hosting platform, allowing students to fork repositories and work on their solutions using version control. GitHub Actions, a CI/CD tool, is utilized to automate the evaluation process and provide continuous integration and delivery capabilities.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Tournament:** A tournament is a space where educators can create battles and students can register to them. It has a registration deadline and a leaderboard that is updated every time a battle ends.
- **Battle:** A battle is a space where students can register to and submit their solutions to a problem. It has a registration deadline, a final submission deadline, a leaderboard that is updated every time a new evaluation is performed and a set of test cases that will be used to evaluate the submissions.

- **GitHub:** GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.
- **GitHub Actions:** GitHub Actions is a CI/CD tool that allows you to automate your software development workflows in the same place you store code and collaborate on pull requests and issues.
- **Educator:** An educator, in the context of the system, is a user of the platform that can create tournaments and battles, invite other educators to tournaments, perform manual evaluations on the submissions of a battle they own and consolidate the results of a battle.
- **Student:** A student, in the context of the system, is a user of the platform that can register to tournaments and battles, create teams and submit their solutions to a battle.
- **Build Automation Scripts:** Build automation scripts are scripts that are run automatically by the system every time a commit is pushed to the main branch of the forked repository of a battle. They are used to evaluate the submissions of the students.
- **Test Case:** A test case is a set of conditions under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do.
- **Timeliness:** Timeliness is the quality of doing something or producing something at the right time. In the context of the system, it refers to the time in which a student submits their solution to a battle with respect to the start of the battle and the final submission deadline.

1.3.2 Acronyms

- **CKB:** CodeKataBattle
- **S2B:** System to Be
- **TDD:** Test-Driven Development
- **CI/CD:** Continuous Integration/Continuous Delivery
- **UI:** User Interface
- **API:** Application Programming Interface
- **UML:** Unified Modeling Language

1.3.3 Abbreviations

- **Gn:** Goal number n
- **Wn:** World Phenomena number n
- **SPn:** Shared Phenomena number n

- **Dn:** Domain Assumption number n
- **Rn:** Requirement number
- **UCn:** Use Case number n

1.4 Revision history

1.5 Reference Documents

The specification of the RASD and DD assignment of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2023/2024.

1.6 Document Structure

- **1. Introduction:** This section provides an overview of the entire document. It describes the purpose and scope of the system, the definitions, acronyms and abbreviations used in the document, the revision history and the reference documents.
- **2. Architectural Design:** This section describes the architecture of the system. It provides an overview of the high-level components and their interaction, the component view, the deployment view, the runtime view, the component interfaces, the selected architectural styles and patterns and other design decisions.
- **3. User Interface Design:** This section provides a mockup of the user interface of the system.
- **4. Requirements Traceability:** This section describes how the requirements defined in the RASD map to the design elements defined in this document.
- **5. Implementation, Integration and Test Plan:** This section describes the order in which the components of the system will be implemented, integrated and tested.
- **6. Effort Spent:** This section describes the amount of time spent by each group member to redact this document.
- **7. References:** This section provides a list of the reference documents used to redact this document.

2 ARCHITECTURAL DESIGN

2.1 Overview: High-level components and their interaction

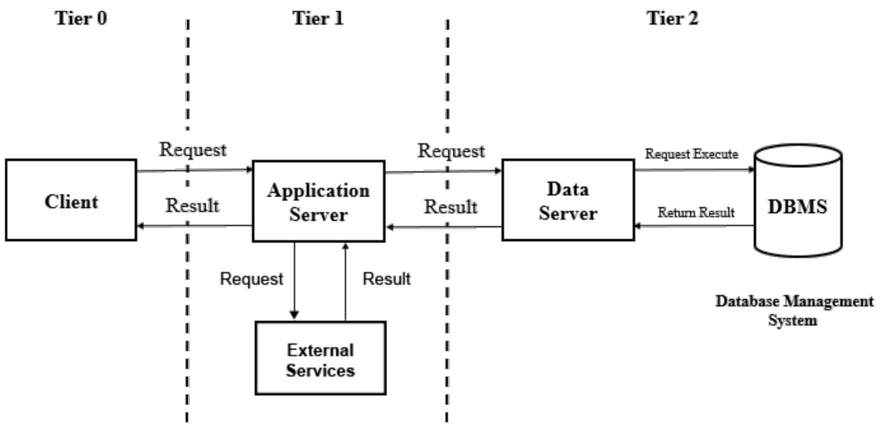

Figure 1: Three-tier Architecture

The chosen architecture for the system is a three-tier architecture, a widely adopted model for developing scalable and maintainable web applications. This architectural style divides the application into three interconnected layers: presentation, application, and data, with the addition of the external services like GitHub and email services. The rationale behind selecting this architecture is to achieve separation of concerns, modularity, and scalability.

Presentation Layer

The presentation layer constitutes the user interfaces for both students and educators, encompassing web or mobile interfaces. It is responsible for user interactions, displaying data, and triggering actions within the system.

Application Layer

The application layer serves as the business logic layer, managing core functionalities such as user authentication, tournament and battle processes, submissions, and evaluations. It facilitates bidirectional communication with both the presentation layer and the data layer.

Data Layer

The data layer, comprising the database and external services, handles the storage and retrieval of persistent data. It communicates with the application layer to respond to data requests, ensuring efficient data management.

2.1.1 Interaction

The interaction between the components is as follows:

Presentation Layer and Application Layer

The presentation layer communicates with the application layer through well-defined APIs. This interaction handles user inputs and actions, ensuring a smooth user experience.

Application Layer and Data Layer

The application layer interacts with the data layer to fetch or store information in the database. This interaction ensures that the platform has access to the necessary data for its operation.

Application Layer and External Components

External services, such as the GitHub API for code repositories and an email service for notifications, are integrated into the application layer. This interaction enhances the platform's functionality by incorporating external features.

2.1.2 Advantages of Three-Tier Architecture

- **Loose Coupling:** Separation of layers promotes loose coupling, enhancing modularity and ease of maintenance.
- **Scalability:** Each layer can be independently scaled based on specific requirements, allowing for better performance and resource utilization.

2.2 Component view

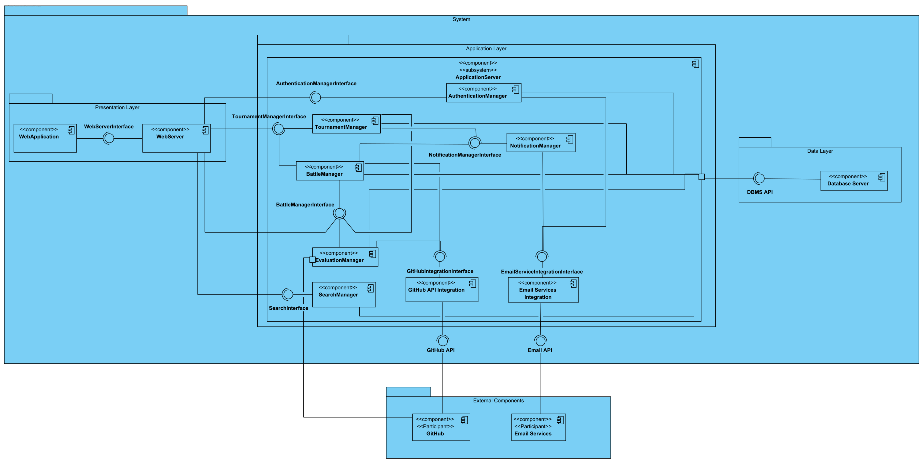
System
Presentation Layer
<<component>> WebApplication
WebServerInterface
<<component>> WebServer
Application Layer
<<component>> <<subsystem>> ApplicationServer
AuthenticationManagerInterface
<<component>> AuthenticationManager
TournamentManagerInterface
<<component>> TournamentManager
NotificationManagerInterface
<<component>> NotificationManager
<<component>> BattleManager
BattleManagerInterface
<<component>> EvaluationManager
<<component>> SearchManager
SearchInterface
GitHubIntegrationInterface
<<component>> GitHub API Integration
EmailServiceIntegrationInterface
<<component>> Email Services Integration
GitHub API
Email API
Data Layer
DBMS API
<<component>> Database Server
External Components
<<component>> <<Participant>> GitHub
<<component>> <<Participant>> Email Services

Figure 2: Component Diagram

2.2.1 Presentation Layer

Web Application:

- Represents the web application that will be used by educators and students.
- Allows educators to create tournaments and battles, invite other educators to tournaments, and perform manual evaluations on the submissions of battles they own.
- Allows students to register to tournaments and battles, create teams, and submit their solutions to battles.
- Provides a leaderboard that is updated in real-time, displaying the rankings of participants based on their performance in the battles.
- Provides a search functionality that allows users to search for tournaments and battles.
- Interfaces with the Web Server for communication with the Application Layer.

Web Server:

- Represents the component responsible for hosting the web application.
- Handles requests and responses from all users, including educators and students.

- Invokes APIs for communication with the Application Layer.

2.2.2 Application Layer

AuthenticationManager:

- Manages user authentication for educators and students.
- Generates and validates authentication tokens.
- Interfaces with email services for account confirmation and recovery.
- Interfaces with the Data Layer for user authentication.

TournamentManager:

- Manages the creation, moderation, and deletion of tournaments.
- Handles tournament-related processes and rules.
- Interfaces with the NotificationManager for tournament-related notifications.
- Interfaces with the Data Layer for tournament data.

BattleManager:

- Manages the creation, moderation, and deletion of battles within tournaments.
- Handles battle-related processes such as registrations, its deadlines, its GitHub repository creation, and its leaderboard.
- Interfaces with the TournamentManager for tournament-related processes such as tournament leadeboard updating.
- Interfaces with the NotificationManager for battle-related notifications.
- Interfaces with GitHub API Integration for repository creation.
- Interfaces with the Data Layer for battle data.

EvaluationManager:

- Handles the evaluation of code submissions based on set criteria.
- Utilizes build automation scripts and test cases for evaluation.
- Interfaces with GitHub API Integration for repository pulling.
- Interfaces with the Data Layer for storage of evaluation results.
- Has a port through which it receives messages from GitHub which are called during GitHub Actions on submissions and evaluations.

NotificationManager:

- Manages the sending of notifications to users.
- Interfaces with Email Services Integration for sending notifications via email.

SearchManager:

- Manages the search functionality of the platform.
- Allows users to search for tournaments and battles.
- Interfaces with the Data Layer for tournament and battle data.

External Components Integration

GitHub API Integration:

- Interacts with the EvaluationManager to notify of new submissions.
- Communicates with the external GitHub API for repository creation and pulling.

Email Services Integration:

- Integrates with Email Services for sending notifications.
- Sends notifications for tournament invitations, submission updates, etc.
- Communicates with the external Email Services API for sending emails.

2.2.3 Data Layer

Database:

- Manages the storage and retrieval of persistent data.
- Stores information related to tournaments, battles, users, submissions, etc.

2.3 Deployment view

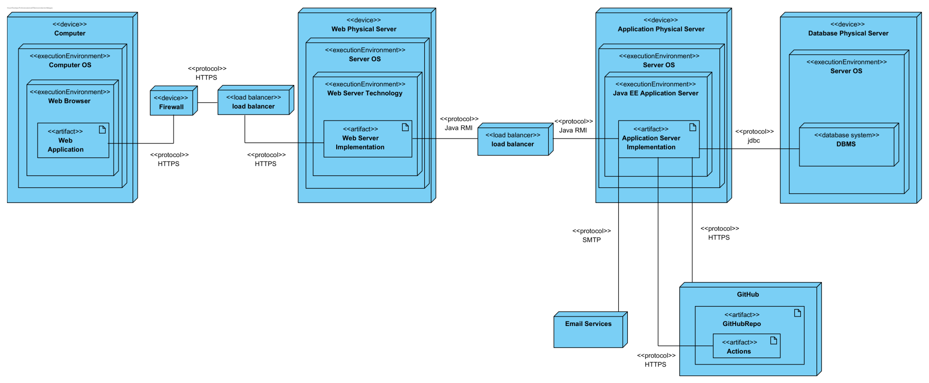

Figure 3: Deployment Diagram

On the deployment diagram, we can see the different components of the system and how they are deployed on the different nodes. In this case, the shown deployment assumes the application is developed on Java. The web application runs on the users' PC which communicates with the web server, which runs on a Web Server Technology which could be Apache Tomcat, Nginx or other. This two elements represent the presentation layer. The web server is connected to the application server, which runs on a Java EE Application Server. This element represents the application layer. The application server is connected to the database and to the external components. The database is managed by a Database Management System such as MySQL, PostgreSQL or other.

2.4 Runtime view

In the following diagrams we can see the sequence diagrams of the main functionalities of the system. These are more fleshed out versions of the use case diagrams presented in the RASD document. It is important to note that the calls to the database server are just not depicted as they would in a real scenario, since these are normally done via queries and not via custom method calls. It was represented via method calls so that it is easier to understand the meaning calls to the database server and the parameters used.

UC1 - Login

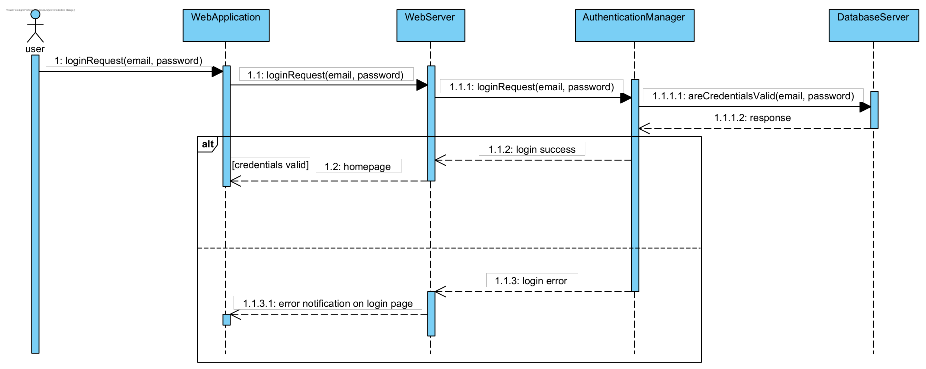
user
WebApplication
WebServer
AuthenticationManager
DatabaseServer
1: loginRequest(email, password)
1.1: loginRequest(email, password)
1.1.1: loginRequest(email, password)
1.1.1.1: areCredentialsValid(email, password)
1.1.1.2: response
alt
1.1.2: login success
[credentials valid]
1.2: homepage
1.1.3: login error
1.1.3.1: error notification on login page

Figure 4: Use Case 1

UC2 - Retrieve Password

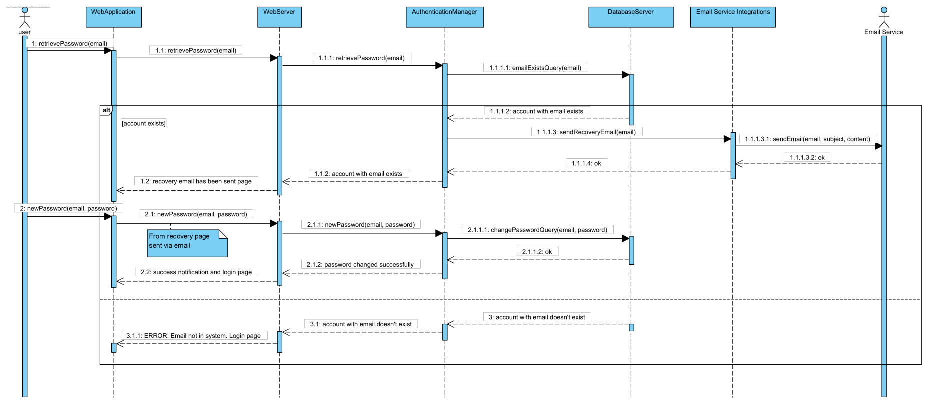
user
WebApplication
WebServer
AuthenticationManager
DatabaseServer
Email Service Integrations
Email Service
1: retrievePassword(email)
1.1: retrievePassword(email)
1.1.1: retrievePassword(email)
1.1.1.1: emailExistsQuery(email)
alt
[account exists]
1.1.1.2: account with email exists
1.1.1.3: sendRecoveryEmail(email)
1.1.1.3.1: sendEmail(email, subject, content)
1.1.1.3.2: ok
1.1.1.4: ok
1.1.2: account with email exists
1.2: recovery email has been sent page
2: newPassword(email, password)
2.1: newPassword(email, password)
From recovery page sent via email
2.1.1: newPassword(email, password)
2.1.1.1: changePasswordQuery(email, password)
2.1.1.2: ok
2.1.2: password changed successfully
2.2: success notification and login page
3: account with email doesn't exist
3.1: account with email doesn't exist
3.1.1: ERROR: Email not in system. Login page

Figure 5: Use Case 2

UC3 - Register

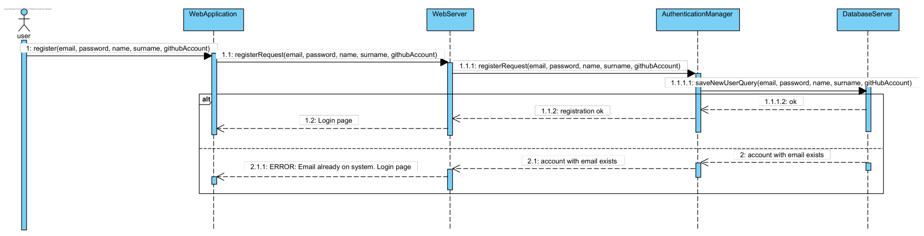
user
WebApplication
WebServer
AuthenticationManager
DatabaseServer
1: register(email, password, name, surname, githubAccount)
1.1: registerRequest(email, password, name, surname, githubAccount)
1.1.1: registerRequest(email, password, name, surname, githubAccount)
1.1.1.1: saveNewUserQuery(email, password, name, surname, gitHubAccount)
alt
1.1.1.2: ok
1.1.2: registration ok
1.2: Login page
2: account with email exists
2.1: account with email exists
2.1.1: ERROR: Email already on system. Login page

Figure 6: Use Case 3

UC4 - Educator Creates Tournament

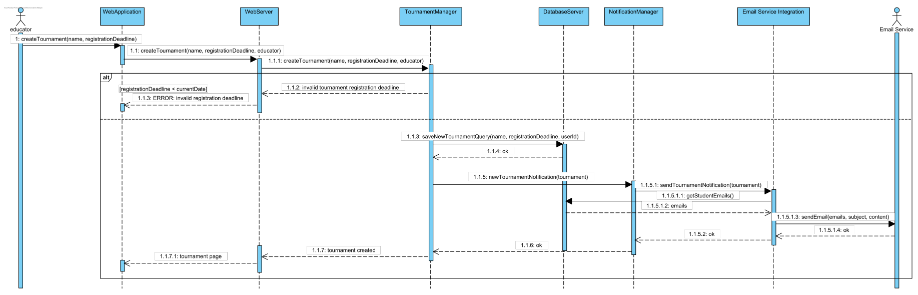
educator
WebApplication
WebServer
TournamentManager
DatabaseServer
NotificationManager
Email Service Integration
Email Service
1: createTournament(name, registrationDeadline)
1.1: createTournament(name, registrationDeadline, educator)
1.1.1: createTournament(name, registrationDeadline, educator)
alt
[registrationDeadline < currentDate]
1.1.2: invalid tournament registration deadline
1.1.3: ERROR: invalid registration deadline
1.1.3: saveNewTournamentQuery(name, registrationDeadline, userId)
1.1.4: ok
1.1.5: newTournamentNotification(tournament)
1.1.5.1: sendTournamentNotification(tournament)
1.1.5.1.1: getStudentEmails()
1.1.5.1.2: emails
1.1.5.1.3: sendEmail(emails, subject, content)
1.1.5.1.4: ok
1.1.5.2: ok
1.1.6: ok
1.1.7: tournament created
1.1.7.1: tournament page

Figure 7: Use Case 4

UC5 - Educator invites other educator to tournament

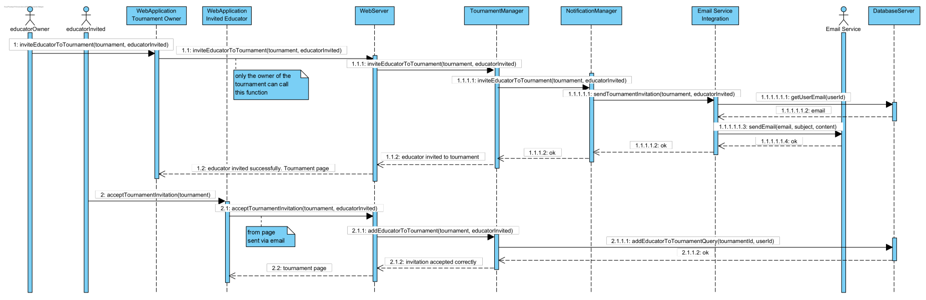
educatorOwner
educatorInvited
WebApplication Tournament Owner
WebApplication Invited Educator
WebServer
TournamentManager
NotificationManager
Email Service Integration
Email Service
DatabaseServer
1: inviteEducatorToTournament(tournament, educatorInvited)
1.1: inviteEducatorToTournament(tournament, educatorInvited)
only the owner of the tournament can call this function
1.1.1: inviteEducatorToTournament(tournament, educatorInvited)
1.1.1.1: inviteEducatorToTournament(tournament, educatorInvited)
1.1.1.1.1: sendTournamentInvitation(tournament, educatorInvited)
1.1.1.1.1.1: getUserEmail(userId)
1.1.1.1.1.2: email
1.1.1.1.1.3: sendEmail(email, subject, content)
1.1.1.1.1.4: ok
1.1.1.1.2: ok
1.1.1.2: ok
1.1.2: educator invited to tournament
1.2: educator invited successfully. Tournament page
2: acceptTournamentInvitation(tournament)
2.1: acceptTournamentInvitation(tournament, educatorInvited)
from page sent via email
2.1.1: addEducatorToTournament(tournament, educatorInvited)
2.1.1.1: addEducatorToTournamentQuery(tournamentId, userId)
2.1.1.2: ok
2.1.2: invitation accepted correctly
2.2: tournament page

Figure 8: Use Case 5

UC6 - Educator creates battle

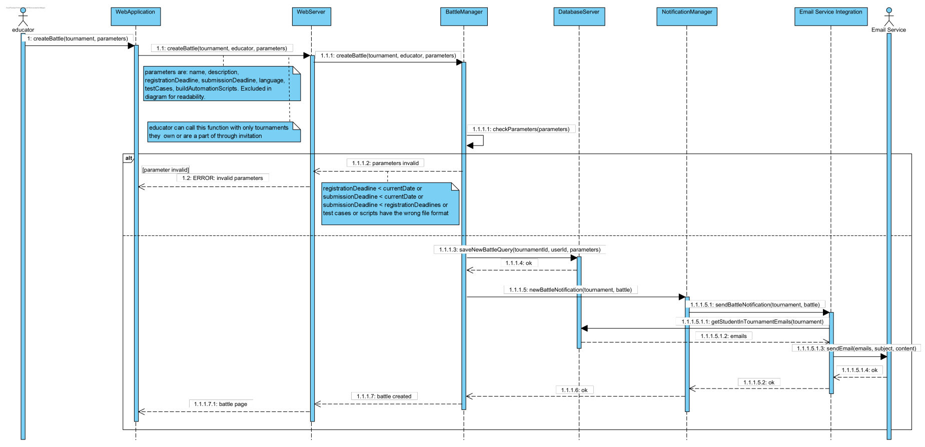
educator
WebApplication
WebServer
BattleManager
DatabaseServer
NotificationManager
Email Service Integration
Email Service
1: createBattle(tournament, parameters)
1.1: createBattle(tournament, educator, parameters)
1.1.1: createBattle(tournament, educator, parameters)
parameters are: name, description, registrationDeadline, submissionDeadline, language, testCases, buildAutomationScripts. Excluded in diagram for readability.
educator can call this function with only tournaments they own or are a part of through invitation
1.1.1.1: checkParameters(parameters)
alt
1.1.1.2: parameters invalid
[parameter invalid]
1.2: ERROR: invalid parameters
registrationDeadline < currentDate or submissionDeadline < currentDate or submissionDeadline < registrationDeadlines or test cases or scripts have the wrong file format
1.1.1.3: saveNewBattleQuery(tournamentId, userId, parameters)
1.1.1.4: ok
1.1.1.5: newBattleNotification(tournament, battle)
1.1.1.5.1: sendBattleNotification(tournament, battle)
1.1.1.5.1.1: getStudentInTournamentEmails(tournament)
1.1.1.5.1.2: emails
1.1.1.5.1.3: sendEmail(emails, subject, content)
1.1.1.5.1.4: ok
1.1.1.5.2: ok
1.1.1.6: ok
1.1.1.7: battle created
1.1.1.7.1: battle page

Figure 9: Use Case 6

UC7 - Tournament Registration

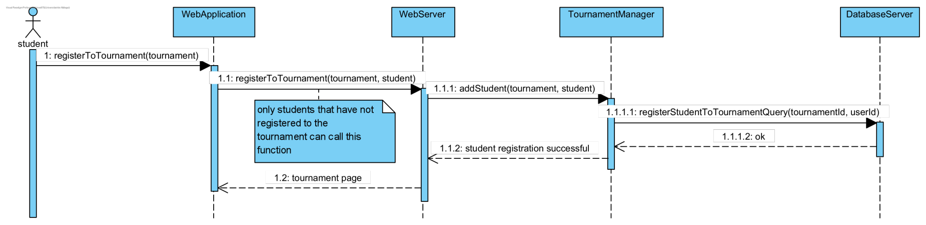
student
WebApplication
WebServer
TournamentManager
DatabaseServer
1: registerToTournament(tournament)
1.1: registerToTournament(tournament, student)
1.1.1: addStudent(tournament, student)
only students that have not registered to the tournament can call this function
1.1.1.1: registerStudentToTournamentQuery(tournamentId, userId)
1.1.1.2: ok
1.1.2: student registration successful
1.2: tournament page

Figure 10: Use Case 7

UC8 - Tournament Unregistration

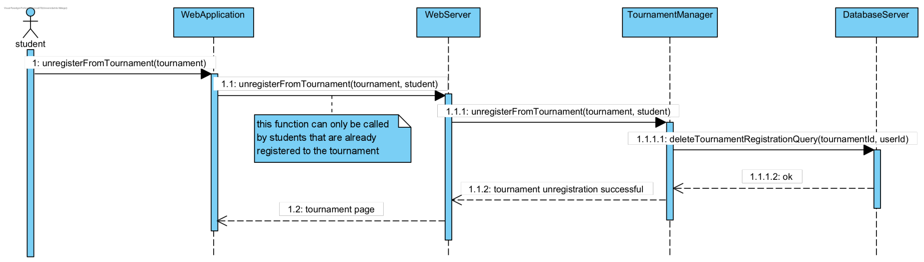
student
WebApplication
WebServer
TournamentManager
DatabaseServer
1: unregisterFromTournament(tournament)
1.1: unregisterFromTournament(tournament, student)
this function can only be called by students that are already registered to the tournament
1.1.1: unregisterFromTournament(tournament, student)
1.1.1.1: deleteTournamentRegistrationQuery(tournamentId, userId)
1.1.1.2: ok
1.1.2: tournament unregistration successful
1.2: tournament page

Figure 11: Use Case 8

UC9 - Battle Registration

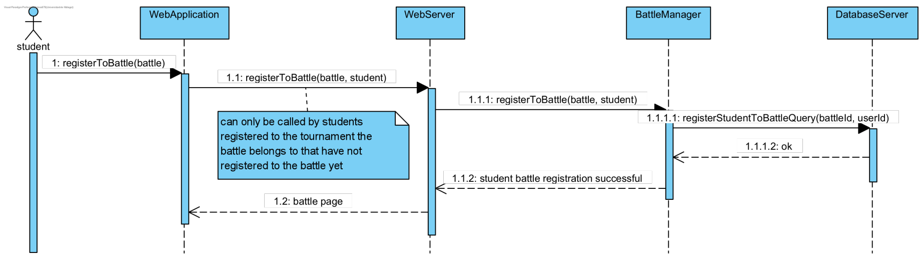
student
WebApplication
WebServer
BattleManager
DatabaseServer
1: registerToBattle(battle)
1.1: registerToBattle(battle, student)
can only be called by students registered to the tournament the battle belongs to that have not registered to the battle yet
1.1.1: registerToBattle(battle, student)
1.1.1.1: registerStudentToBattleQuery(battleId, userId)
1.1.1.2: ok
1.1.2: student battle registration successful
1.2: battle page

Figure 12: Use Case 9

UC10 - Student joins battle via invite

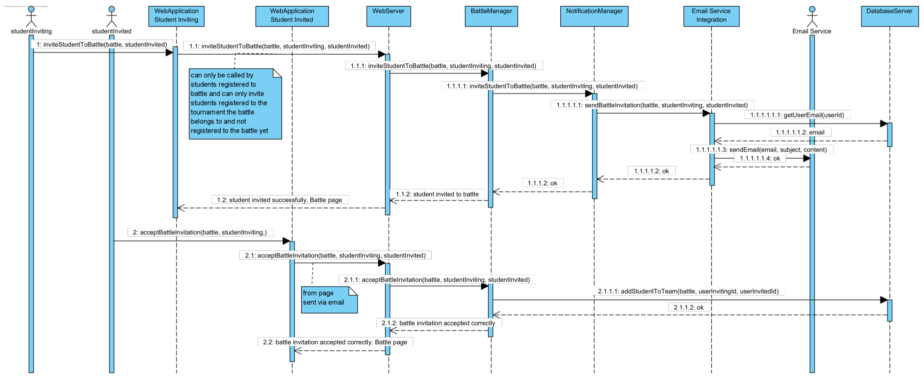
studentInviting
studentInvited
WebApplication Student Inviting
WebApplication Student Invited
WebServer
BattleManager
NotificationManager
Email Service Integration
Email Service
DatabaseServer
1: inviteStudentToBattle(battle, studentInvited)
1.1: inviteStudentToBattle(battle, studentInviting, studentInvited)
can only be called by students registered to battle and can only invite students registered to the tournament the battle belongs to and not registered to the battle yet
1.1.1: inviteStudentToBattle(battle, studentInviting, studentInvited)
1.1.1.1: inviteStudentToBattle(battle, studentInviting, studentInvited)
1.1.1.1.1: sendBattleInvitation(battle, studentInviting, studentInvited)
1.1.1.1.1.1: getUserEmail(userId)
1.1.1.1.1.2: email
1.1.1.1.1.3: sendEmail(email, subject, content)
1.1.1.1.1.4: ok
1.1.1.1.2: ok
1.1.1.2: ok
1.1.2: student invited to battle
1.2: student invited successfully. Battle page
2: acceptBattleInvitation(battle, studentInviting,)
2.1: acceptBattleInvitation(battle, studentInviting, studentInvited)
from page sent via email
2.1.1: acceptBattleInvitation(battle, studentInviting, studentInvited)
2.1.1.1: addStudentToTeam(battle, userInvitingId, userInvitedId)
2.1.1.2: ok
2.1.2: battle invitation accepted correctly
2.2: battle invitation accepted correctly. Battle page

Figure 13: Use Case 10

UC11 - Battle Unregistration

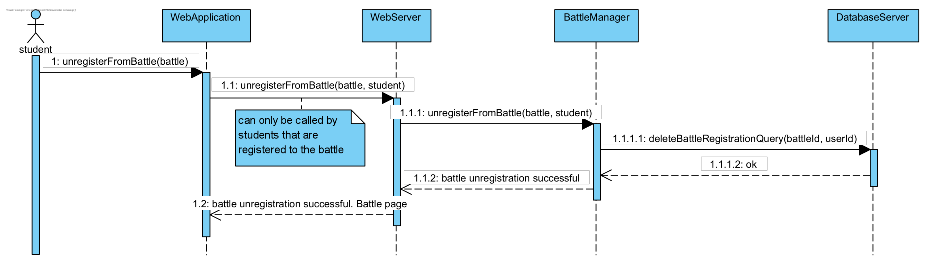
student
WebApplication
WebServer
BattleManager
DatabaseServer
1: unregisterFromBattle(battle)
1.1: unregisterFromBattle(battle, student)
can only be called by students that are registered to the battle
1.1.1: unregisterFromBattle(battle, student)
1.1.1.1: deleteBattleRegistrationQuery(battleId, userId)
1.1.1.2: ok
1.1.2: battle unregistration successful
1.2: battle unregistration successful. Battle page

Figure 14: Use Case 11

UC12 - Students makes submission

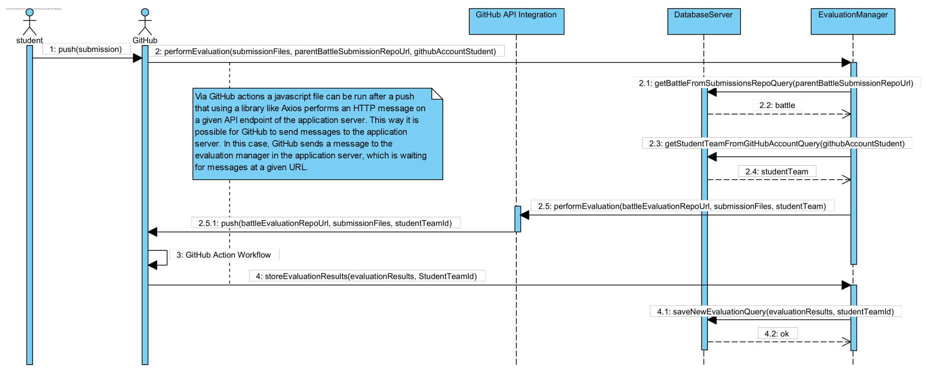
student
GitHub
GitHub API Integration
DatabaseServer
EvaluationManager
1: push(submission)
2: performEvaluation(submissionFiles, parentBattleSubmissionRepoUrl, githubAccountStudent)
2.1: getBattleFromSubmissionsRepoQuery(parentBattleSubmissionRepoUrl)
2.2: battle
Via GitHub actions a javascript file can be run after a push that using a library like Axios performs an HTTP message on a given API endpoint of the application server. This way it is possible for GitHub to send messages to the application server. In this case, GitHub sends a message to the evaluation manager in the application server, which is waiting for messages at a given URL.
2.3: getStudentTeamFromGitHubAccountQuery(githubAccountStudent)
2.4: studentTeam
2.5: performEvaluation(battleEvaluationRepoUrl, submissionFiles, studentTeam)
2.5.1: push(battleEvaluationRepoUrl, submissionFiles, studentTeamId)
3: GitHub Action Workflow
4: storeEvaluationResults(evaluationResults, StudentTeamId)
4.1: saveNewEvaluationQuery(evaluationResults, studentTeamId)
4.2: ok

Figure 15: Use Case 12

UC13 - Educator Performs Manual Evaluations and Closes Battle

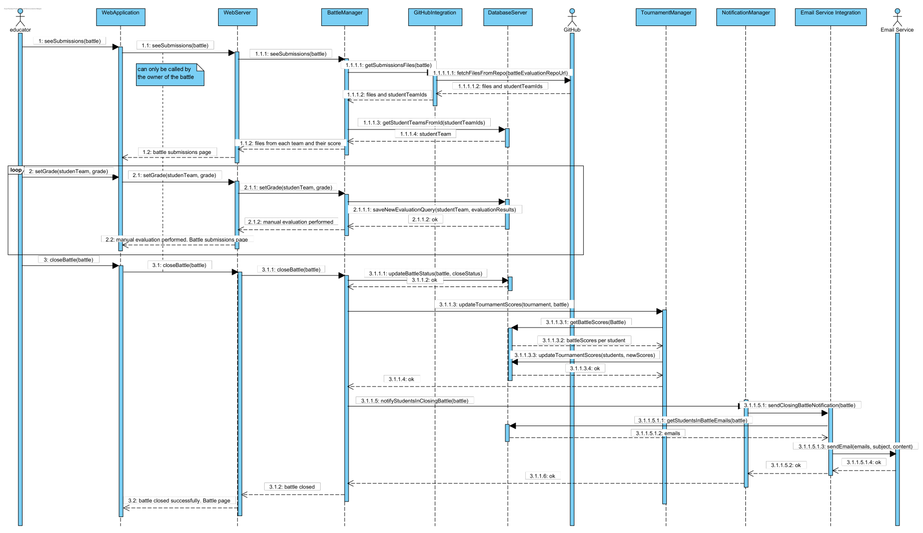
educator
WebApplication
WebServer
BattleManager
GitHubIntegration
DatabaseServer
GitHub
TournamentManager
NotificationManager
Email Service Integration
Email Service
1: seeSubmissions(battle)
1.1: seeSubmissions(battle)
1.1.1: seeSubmissions(battle)
can only be called by the owner of the battle
1.1.1.1: getSubmissionsFiles(battle)
1.1.1.1.1: fetchFilesFromRepo(battleEvaluationRepoUrl)
1.1.1.1.2: files and studentTeamIds
1.1.1.2: files and studentTeamIds
1.1.1.3: getStudentTeamsFromId(studentTeamIds)
1.1.1.4: studentTeam
1.1.2: files from each team and their score
1.2: battle submissions page
loop
2: setGrade(studenTeam, grade)
2.1: setGrade(studenTeam, grade)
2.1.1: setGrade(studenTeam, grade)
2.1.1.1: saveNewEvaluationQuery(studentTeam, evaluationResults)
2.1.1.2: ok
2.1.2: manual evaluation performed
2.2: manual evaluation performed. Battle submissions page
3: closeBattle(battle)
3.1: closeBattle(battle)
3.1.1: closeBattle(battle)
3.1.1.1: updateBattleStatus(battle, closeStatus)
3.1.1.2: ok
3.1.1.3: updateTournamentScores(tournament, battle)
3.1.1.3.1: getBattleScores(Battle)
3.1.1.3.2: battleScores per student
3.1.1.3.3: updateTournamentScores(students, newScores)
3.1.1.3.4: ok
3.1.1.4: ok
3.1.1.5: notifyStudentsInClosingBattle(battle)
3.1.1.5.1: sendClosingBattleNotification(battle)
3.1.1.5.1.1: getStudentsInBattleEmails(battle)
3.1.1.5.1.2: emails
3.1.1.5.1.3: sendEmail(emails, subject, content)
3.1.1.5.1.4: ok
3.1.1.5.2: ok
3.1.1.6: ok
3.1.2: battle closed
3.2: battle closed successfully. Battle page

Figure 16: Use Case 13

UC14 - Educator Closes Tournament

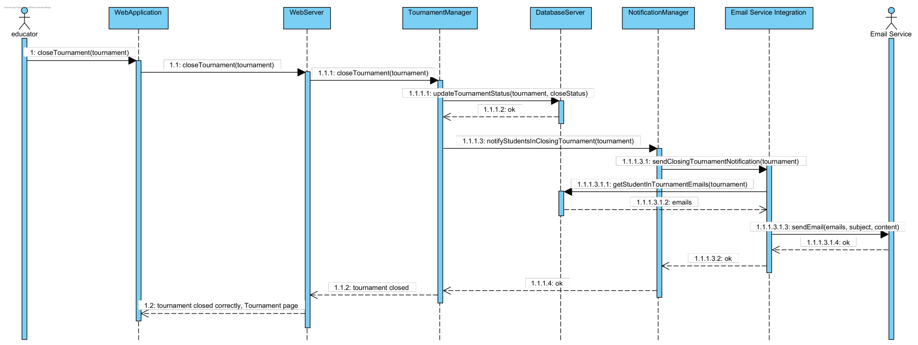
educator
WebApplication
WebServer
TournamentManager
DatabaseServer
NotificationManager
Email Service Integration
Email Service
1: closeTournament(tournament)
1.1: closeTournament(tournament)
1.1.1: closeTournament(tournament)
1.1.1.1: updateTournamentStatus(tournament, closeStatus)
1.1.1.2: ok
1.1.1.3: notifyStudentsInClosingTournament(tournament)
1.1.1.3.1: sendClosingTournamentNotification(tournament)
1.1.1.3.1.1: getStudentInTournamentEmails(tournament)
1.1.1.3.1.2: emails
1.1.1.3.1.3: sendEmail(emails, subject, content)
1.1.1.3.1.4: ok
1.1.1.3.2: ok
1.1.1.4: ok
1.1.2: tournament closed
1.2: tournament closed correctly, Tournament page

Figure 17: Use Case 14

UC15 - Find Tournament via search

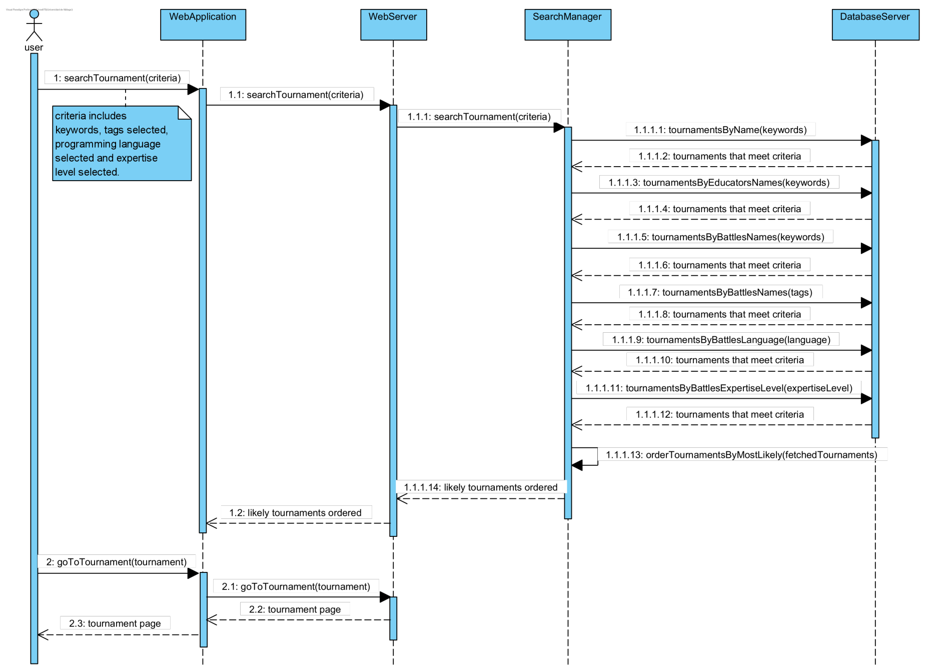
user
WebApplication
WebServer
SearchManager
DatabaseServer
1: searchTournament(criteria)
1.1: searchTournament(criteria)
criteria includes keywords, tags selected, programming language selected and expertise level selected.
1.1.1: searchTournament(criteria)
1.1.1.1: tournamentsByName(keywords)
1.1.1.2: tournaments that meet criteria
1.1.1.3: tournamentsByEducatorsNames(keywords)
1.1.1.4: tournaments that meet criteria
1.1.1.5: tournamentsByBattlesNames(keywords)
1.1.1.6: tournaments that meet criteria
1.1.1.7: tournamentsByBattlesNames(tags)
1.1.1.8: tournaments that meet criteria
1.1.1.9: tournamentsByBattlesLanguage(language)
1.1.1.10: tournaments that meet criteria
1.1.1.11: tournamentsByBattlesExpertiseLevel(expertiseLevel)
1.1.1.12: tournaments that meet criteria
1.1.1.13: orderTournamentsByMostLikely(fetchedTournaments)
1.1.1.14: likely tournaments ordered
1.2: likely tournaments ordered
2: goToTournament(tournament)
2.1: goToTournament(tournament)
2.2: tournament page
2.3: tournament page

Figure 18: Use Case 15

UC16 - Find Tournament on main page

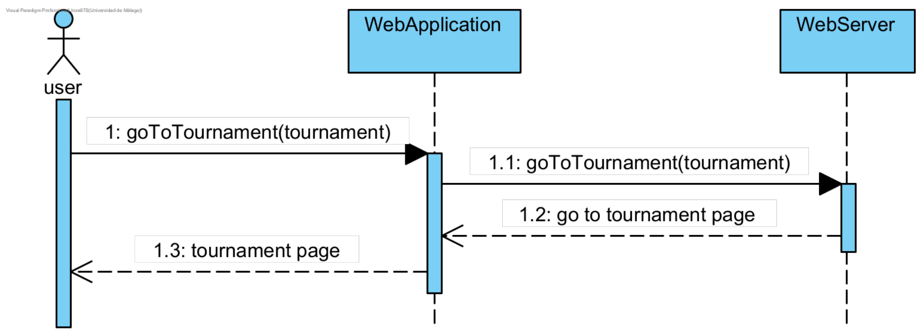
user
WebApplication
WebServer
1: goToTournament(tournament)
1.1: goToTournament(tournament)
1.2: go to tournament page
1.3: tournament page

Figure 19: Use Case 16

UC17 - Find Battle on main page or on tournament page

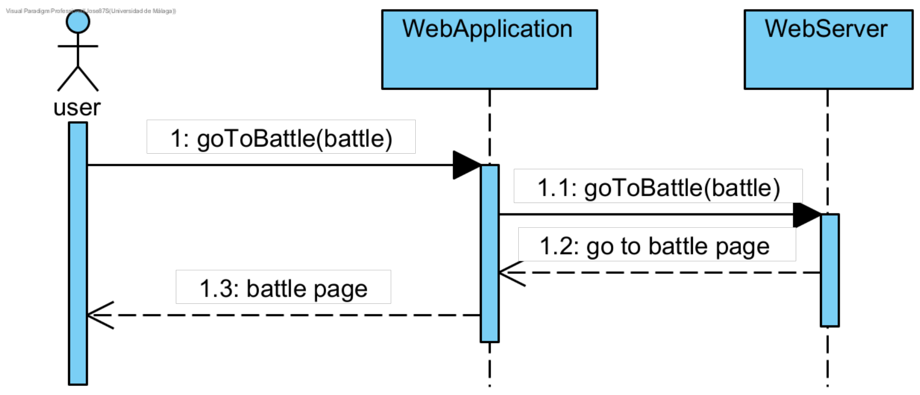

Figure 20: Use Case 17 and 18

2.5 Component interfaces

WebServer:

- void **loginRequest**(string **email**, string **password**)
- void **registerRequest**(string **email**, string **password**, string **name**, string **surname**, string **gitHubAccount**)
- void **retrievePassword**(string **email**)
- void **newPassword**(string **email**, string **password**)
- void **createTournament**(string **name**, Date **registrationDeadline**, Educator **educator**)
- void **inviteEducatorToTournament**(Tournament **tournament**, Educator **educatorInvited**)
- void **acceptTournamentInvitation**(Tournament **tournament**, Educator **educatorInvited**)
- void **declineTournamentInvitation**(Tournament **tournament**, Educator **educatorInvited**)
- void **createBattle**(Tournament **tournament**, Educator **educator**, string **name**, string **description**, Date **registrationDeadline**, Date **finalSubmissionDeadline**, string **language**, File[] **testCases**, File[] **buildAutomationScripts**)
- void **registerToTournament**(Tournament **tournament**, Student **student**)
- void **unregisterFromTournament**(Tournament **tournament**, Student **student**)
- void **registerToBattle**(Battle **battle**, Student **student**)
- void **inviteStudentToBattle**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)
- void **acceptBattleInvitation**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)
- void **unregisterFromBattle**(Battle **battle**, Student **student**)
- void **seeSubmissions**(Battle **battle**)
- void **setGrade**(StudentTeam **studentTeam**, float **grade**)
- void **closeBattle**(Battle **battle**)
- void **closeTournament**(Tournament **tournament**)
- void **searchTournament**(string **keyword**, string[] **tags**, string **language**, string **expertiseLevel**)

- void **goToTournament**(Tournament **tournament**)
- void **goToBattle**(Battle **battle**)

AuthenticationManager:

- void **loginRequest**(string **email**, string **password**)
- void **registerRequest**(string **email**, string **password**, string **name**, string **surname**, string **gitHubAccount**)
- void **retrievePassword**(string **email**)
- void **newPassword**(string **email**, string **password**)

TournamentManager:

- void **createTournament**(string **name**, Date **registrationDeadline**, Educator **educator**)
- void **inviteEducatorToTournament**(Tournament **tournament**, Educator **educatorInvited**)
- void **addEducatorToTournament**(Tournament **tournament**, Educator **educatorInvited**)
- void **addStudent**(Tournament **tournament**, Student **student**)
- void **unregisterFromTournament**(Tournament **tournament**, Student **student**)
- void **updateTournamentScores**(Tournament **tournament**, Battle **battle**)
- void **closeTournament**(Tournament **tournament**)

BattleManager:

- void **createBattle**(Tournament **tournament**, Educator **educator**, string **name**, string **description**, Date **registrationDeadline**, Date **finalSubmissionDeadline**, string **language**, File[] **testCases**, File[] **buildAutomationScripts**)
- void **registerToBattle**(Battle **battle**, Student **student**)
- void **inviteStudentToBattle**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)
- void **acceptBattleInvitation**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)
- void **unregisterFromBattle**(Battle **battle**, Student **student**)
- Map¡StudentTeam, File[]¿ **seeSubmissions**(Battle **battle**)

- void **setGrade**(StudentTeam **studentTeam**, float **grade**)
- void **closeBattle**(Battle **battle**)

NotificationManager:

- void **newTournamentNotification**(Tournament **tournament**)
- void **inviteEducatorToTournament**(Tournament **tournament**, Educator **educatorInvited**)
- void **newBattleNotification**(Tournament **tournament**, Battle **battle**)
- void **inviteStudentToBattle**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)
- void **notifyStudentsInClosingBattle**(Battle **battle**)
- void **notifyStudentsInClosingTournament**(Tournament **tournament**)

EvaluationManager:

- void **performEvaluation**(File[] **submissionFiles**, string **parentBattleSubmissionRepoUrl**, string **gitHubAccountStudent**)
- void **storeEvaluationResults**(float **evaluationResult**, int **StudentTeamId**)

SearchManager:

- void **searchTournament**(string **keyword**, string[] **tags**, string **language**, string **expertiseLevel**)

GitHub API Integration:

- void **performEvaluation**(String **battleEvaluationRepoUrl**, File[] **submissionFiles**, StudentTeam **studentTeam**)
- Map¡StudentId, File[]¿ **getSubmissionFiles**(Battle **battle**)

Email Service Integration:

- void **sendRecoveryEmail**(string **email**)
- void **sendTournamentNotification**(Tournament **tournament**)
- void **sendTournamentInvitation**(Tournament **tournament**, Educator **educatorInvited**)
- void **sendBattleNotification**(Tournament **tournament**, Battle **battle**)
- void **sendBattleInvitation**(Battle **battle**, Student **studentInviting**, Student **studentInvited**)

- void **sendClosingBattleNotification**(Battle **battle**)
- void **sendClosingTournamentNotification**(Tournament **tournament**)

2.6 Selected architectural styles and patterns

2.7 Other design decisions

3 USER INTERFACE DESIGN

4 REQUIREMENTS TRACEABILITY

5 IMPLEMENTATION, INTEGRATION AND TEST PLAN

6 EFFORT SPENT

7 REFERENCES