AY 2023/2024

POLITECNICO
MILANO 1863

DD: Design Document

Tommaso Capacci Gabriele Ginestroni

Professor
Elisabetta Di Nitto

Version 1
January 2, 2024

Contents

1 Introduction

1.1 Scope

Traditional software programming education often lacks of hands-on experience and continuous evaluation. CodeKataBattle addresses these issues by providing a platform for competitive programming challenges which promotes teamwork and emphasizes the test-first approach in software development. It allows students to apply theoretical knowledge in tournaments with an automated scoring system. Instructors benefit from closer mentorship through code reviews and manual evaluation, creating a cycle of learning through practice, feedback, and community engagement.

CodeKataBattle's platform will employ a microservices architecture, emphasizing the decomposition of the system into small, independently deployable services. Each microservice will be dedicated to a specific business capability, promoting modular development and ease of maintenance.The architecture facilitates scalability, allowing individual services to be scaled independently based on demand. Other important design choices include:

- **Service Discovery**: A service registry will be used to allow services to locate each other without prior knowledge of their location. This enables efficient communication between services by providing up-to-date information. Service discovery enhances fault tolerance and load balancing, contributing to the overall reliability of the system.
- **API Gateway**: An API gateway will be used to provide a single entry point for clients to access the system. This simplifies the client interface by abstracting the underlying microservices and provides a centralized location for authentication and authorization. The API Gateway enhances security measures, ensuring controlled and secure access to the microservices.
- **Hybrid Communication Framework**: While most services exploit synchronous communication via REST api calls, an event-driven communication framework is also used to facilitate communication between specific microservices, allowing them to be loosely coupled and promoting both modularity and scalability. This framework enhances the

reliability of these services by providing a mechanism for asynchronous communication between them (i.e. queues).

- **Data Management Strategies**: The system employs tailored data management strategies, utilizing databases suitable for microservices. Both relational and NoSQL databases are considered for each specific service in order to provide flexibility and scalability.

Incorporating these key properties into the CodeKataBattle system should provide a robust and scalable architecture, ensuring modularity, responsiveness, security, reliability, and effective data management.

1.2 Definitions, Acronyms, Abbreviations

TODO: remove unused ones

1.2.1 Definitions

- **User:** anyone that has registered to the platform
- **Student:** the first kind of users and, basically, the people this product is designed for. Their objective is to submit solutions to battles
- **Team:** students can decide to group up and form a team to partecipate to a battle. The score assigned to the submission of a team will be assigned also to each one of its members
- **Educator:** the second kind of users. They create tournaments, set-up battles, and eventually, evaluate the solutions that teams of students have submitted during the challenge
- **Tournament:** collection of coding exercises (battles). Interested students can subscribe to it and participate to its battles as soon as they are published
- **Code Kata Battle (or battle):** the atomic unit of a tournament. Usually students are asked to implement an algorithm or to develope a simple project that solves the task. Each battle belongs to a specific tournament: students submitting solutions for a battle will obtain a score that will be used to compute both the team's rank for the battle and the members' tournament rank

- **Code Kata:** description and software project necessary for the battle, including test cases and build automation scripts. These are uploaded by the educator at battle creation time
- **Tournament collaborator:** other educator that is added by the tournament creator to help him in the management of the tournament. He can create battles and evaluate their submissions
- **GitHub:** web-based hosting service for version control, mostly used for programming. It offers both distributed version control and source code management functionalities
- **GitHub repository:** a repository is a storage space where some project files are stored. It can be either public or private. In the context of the platform, each battle is associated with a GitHub repository that is created by the system and shared with the teams
- **GitHub collaborator:** person who is granted access to a GitHub repository with write permission
- **GitHub Actions:** GitHub feature that allows to automate tasks directly on GitHub, such as building and testing code, or deploying applications. In the context of the platform, GitHub Actions is used to automatically notify the system when a new submission is pushed to the repository of a team
- **Test cases:** each battle is associated with a set of test cases, which are input-output value pairs that describe the correct behavior of the ideal solution
- **Static analysis:** is the analysis of programs performed without executing them, usually achieved by applying formal methods directly to the source code. In the context of the platform this kind of analysis is used to extract additional information about the level of security, reliability and maintainability of a battle submission
- **Functional analysis:** measures the correctness of a solution in terms of passed test cases
- **Timeliness:** measures the time passed between the start of the battle and the last commit of the submission

- **Score:** to each solution is assigned a score which is computed taking into account timeliness, functional and static analysis and, eventually, manual score assigned by the educator that created the challenge. The score is a natural number between 0 and 100 (the higher the better)
- **Rank:** during a battle, students can visualize the ranking of teams taking part to the battle. Moreover, at the end of each battle, the platform updates the personal tournament score of each student. Specifically, the score is computed as the sum of all the battles scores received in that tournament. This overall score is used to fill out a ranking of all the students participating to the tournament which is accessible by any time and by any user subscribed to the platform
- **Notification:** it's an email alert that is sent to users to inform them that a certain event occurred such as the creation of a new tournament and battle or the publication of the final rank of a battle

1.2.2 Acronyms

- **DD:** Design Document
- **RASD:** Requirements Analysis and Specification Document
- **CKB:** Code Kata Battles
- **API:** Application Programming Interface
- **UML:** Unified Modeling Language
- **HTML:** HyperText Markup Language
- **CSS:** Cascading Style Sheets
- **JSON:** JavaScript Object Notation
- **OS:** Operating System
- **REST:** REpresentational State Transfer
- **URL:** Uniform Resource Locator
- **HTTPS:** HyperText Transfer Protocol Secure
- **DMZ:** Demilitarized Zone

1.2.3 Abbreviations

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

1.3 Revision History

- January 2, 2024: version 1.0 (first release)

1.4 Reference Documents

- Specification document: "Assignment RDD AY 2023-2024"
- DD reference template: "04e.QualitiesAndCreatingDD.pdf"
- UML official specification https://www.omg.org/spec/UML
- GitHub API official documentation: https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api?apiVersion=2022-11-28

1.5 Document Structure

- **Section 1: Introduction**
  In this section a general description of the document and the system to be developed is provided, also including a glossary of terms used and a list of reference documents.
- **Section 2: Architectural Design**
  Here the high-level structure of the software is outlined. This includes the identification of major components, their interactions, and the overall flow of data within the system. Dependencies on external factors or third-party integrations are also detailed, offering a comprehensive view of the software's architecture.

- **Section 3: User Interface Design**
  Focused on the end-user experience, this section describes the layout, interactivity, and visual elements of the software's user interface. It includes mockups of the main pages of the web application.

- **Section 4: Requirements Traceability**
  Here the relation between software requirements and design elements is highlighted. This is achieved through the use of a traceability matrix.

- **Section 5: Implementation, Integration and Test Plan**
  This section is a comprehensive guide that covers the main aspects of the software development lifecycle. It outlines the details of implementation and integration plan, as well as the testing strategy.

- **Section 6: Effort spent**
  The sixth and last chapter contains the time spent by each contributor of this document.

2 Architectural Design

2.1 Overview

The following diagram represents the high level architecture of the system, including the external entities that will interact with it.

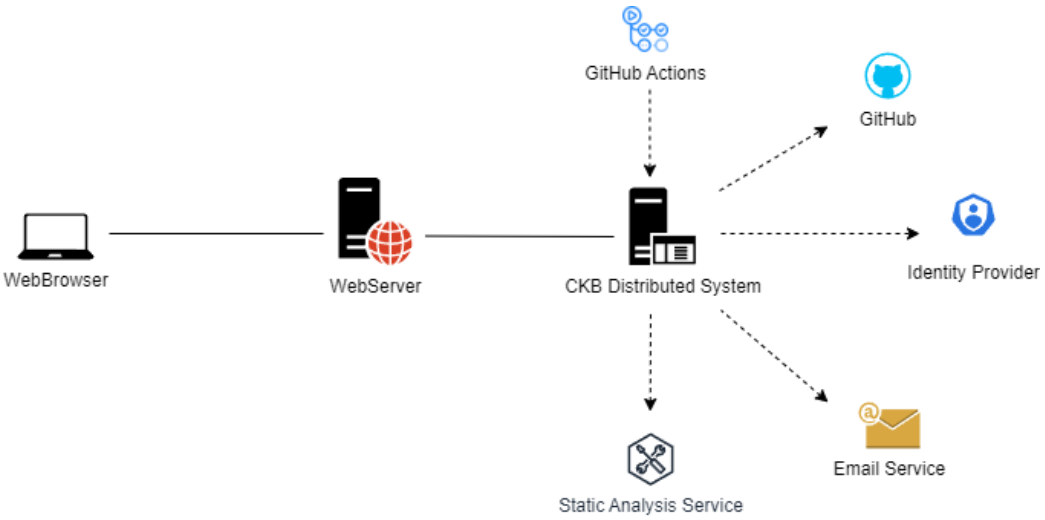


Figure 1: CKB system diagram

The main elements contained in Figure 1 are:

- **WebBrowser** used by students and educators to access the system functionalities through the web application.
- **WebServer** whose functions are:
  - Serving static assets (HTML, CSS, JavaScript) to the client, necessary for handling the initial rendering of the UI.
  - Managing the client-side application and routing.
  - Generating requests directed to the backend system.
- **CKB Distributed System** which is the distributed system composed of multiple microservices that implement the core functionalities of the

system. Its in charge of the data management, application and integration logic of the whole CKB platform.

- **External Entities** the CKB Distributed System must be able to integrate with to accomplish its functionalities. The arrow in the diagram highlights the direction of the interaction between the system and the external entities.

2.2 Component view

The following component diagram highlights the main components of the system and their interaction with external entities and services. In the diagram, the components have been organized to highlight the logical grouping of the system elements.

The WebApplication component represents the presentation layer of the system, being the only entry point for the users. The application and integration logics are represented together due to their tight interaction, while the data layer contains the databases accessed by the respective microservices.

Different colors are used to highlight components that share similar roles in the system.

- Orange components represent the system's microservices. Complex microservices have been further decomposed into subcomponents, for a more fine grained representation. Some microservices have an important role in the integration and communication with external entities as well, but have been depicted with their orange color used for microservices. This aspect will be clarified in the detailed description of the components that follows the diagram.

- Yellow components represent the model of the database accessed by its microservice. The model offers to the microservice an abstraction of the database, allowing it to access the data without knowing the underlying database implementation technology.

- Violet has been used to highlight components or services that cover an important role for the communication between microservices. It has been used for the queues subcomponents, which are used to implement the asynchronous and concurrent communication between specific microservices. Also the ServiceRegistry has been represented with the

this color, since it enables the communication in the system. It's important to notice that for the sake of simplicity and readability, the interface offered by the ServiceRegistry has been depicted with some dotted arrows that connect all the microservices to it.

- Red components represent the external services that interact with the system.
- Finally, the green color has been used to highlight the databases components that are used to store the data of the system.

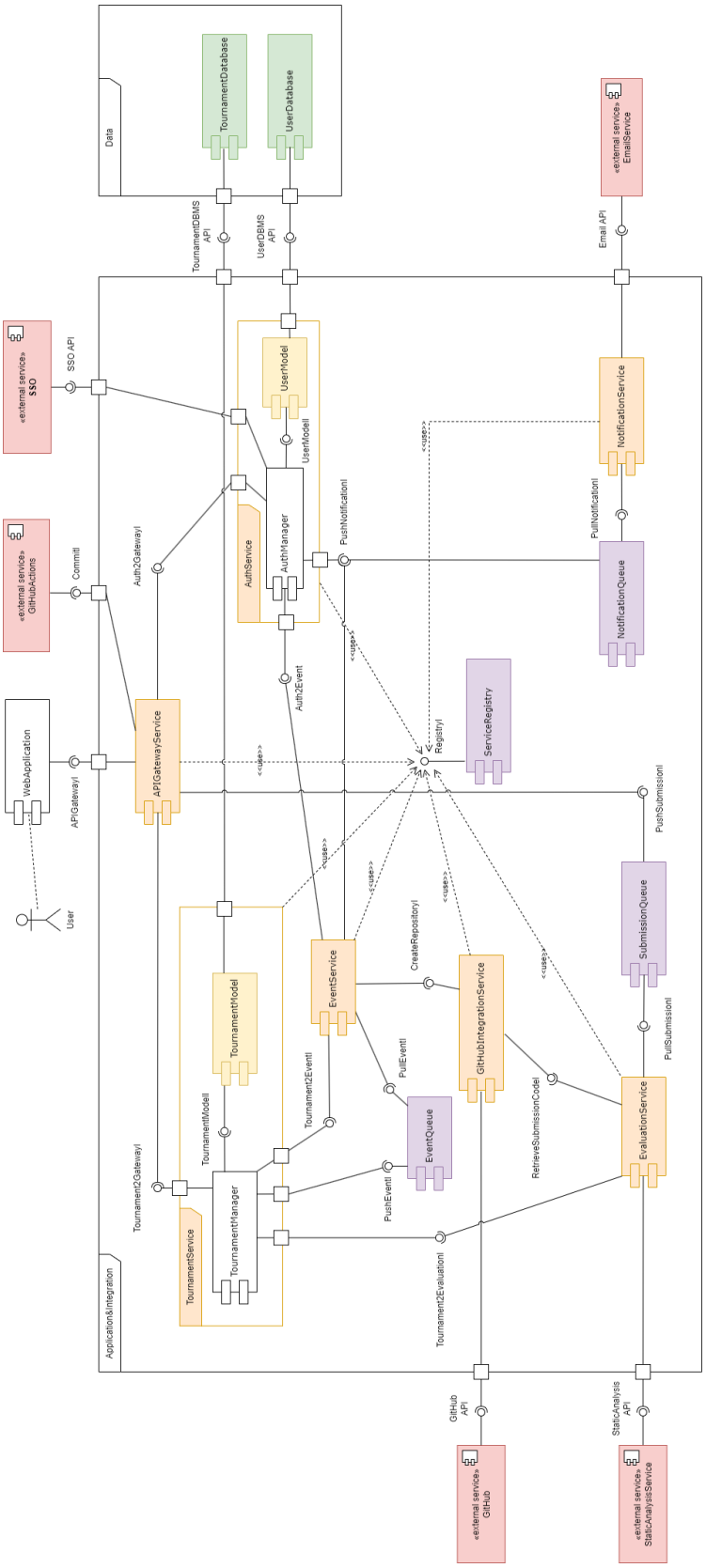


Figure 2: Component diagram

The components in Figure 2 are :

- **WebApplication**: the web app to which users of the CKB platform (students and educators) connect through a modern web browser. It is the front end of the system, and thanks to the interface offered by the **APIGatewayService**, allows users to manage and access the most important aspects of tournaments and battles.
- **APIGatewayService**: this component is the microservice that exposes the REST API used by the WebApplication. Indeed, it allows the implementation of the main functionalities needed by the users of the web application by orchestrating the microservices. It also offers a REST API, used by the GitHub Actions Service, to notify a submission of a student on the Github repository. It's also responsible for the following functionalities:
  - Acts as a single entry point for client requests.
  - Handles authentication, authorization through the interaction with the AuthService.
  - Routes requests to the appropriate microservices that provide the required business functionality.
  - Aggregates responses from multiple microservices if needed.
  - Load balancing and API rate limiting.
- **AuthService**: this component is the microservice that handles the authentication of the users of the system. To accomplish this task, it may be required to interact with external identity providers, depending on the user preferences. It is responsible also for all the data related to the users of the system, such as their personal information and their roles. It includes the following subcomponents:
  - **AuthManager**: implements the main logical functionalities of the AuthService, exposing APIs used by other microservices to authenticate and to retrieve user information.
  - **UserModel**: represents the model of the database used by the AuthService to store the data related to the users of the system.
- **TournamentService**: this component is the microservice that implements most of the functionalities needed by the users. It handles:

  - Management of tournaments and battles: it allows the creation of battle and tournaments, enrollment of students to tournaments and battles.
  - Management of events regarding tournaments and battles.
  - Management of the ranking of the students.
  - Management of data related to tournaments, battles and submissions
  - Manual evaluation of the submissions of the students.

  It is composed of the following subcomponents:

  - **TournamentManager**: implements the main logical functionalities of the TournamentService, exposing APIs used by other microservices to manage tournaments, battles, team and their data.
  - **TournamentModel**: represents the model of the database used by the TournamentService to store the data related to tournaments and battles.

- **EventService**: periodically checks the EventQueue for events and processes them once the deadlines are reached. It is the consumer of events created by the TournamentService.

- **GitHubIntegrationService**: this component is the microservice that handles the integration with GitHub. It is responsible for the following functionalities:

  - Creation of the GitHub repository of the battle.
  - Retrieval of the code of the submission from the GitHub repository.

- **EvaluationService**: this component is the microservice that handles the evaluation of the submissions of the students. It periodically checks the SubmissionQueue for submissions to evaluate and processing them, so it is the consumer of the events appended by the APIGatewayService on behalf of the GitHubActions. The submission notification contains only the token associated to the team (which corresponds to the teamId), so the service retrieves the code from the GitHub repository, interacting with the GitHubIntegrationService. This service is responsible for the following functionalities:

  - Evaluation of the submissions, in terms of timeliness and functional analysis
  - Integration with external static code analysis tools to evaluate the quality of the code of the submissions.

- **NotificationService**: this component is the microservice that handles the notifications of the users of the system. It periodically checks the NotificationQueue for new notifications to send and processing them. To perform this task, the NotificationService interacts with the external email service provider to send notifications to the users. It is responsible for the following functionalities:
  - Dispatch of confirmation email to new registered users.
  - Dispatch of email notifications to users in case of events related to tournaments and battles.
- **ServiceRegistry**: this component handles the registration of the microservices to the system. It offers to all the microservices the following:
  - Registration of the microservices istances to the system.
  - Discovery of the microservices istances by the other microservices.
  - Availability check of the microservices istances, by receiving periodic heartbeats from them.
- **TournamentDatabase**: this component is the database used by the system to store the data related to tournaments and battles, including also scores and ranks.
- **UserDatabase**: this component is the database used by the system to store the data related to the users of the system, such as kind of user, email and usernames.
- **NotificationQueue**: queue that stores new notifications to be dispatched.
- **SubmissionQueue**: queue that stores notifications about new pending submissions, appended by the GitHubActions through the REST API exposed by the APIGatewayService, yet to be evaluated.

- **EventQueue**: the queue used by the TournamentService to manage the events related to tournaments and battles. It is implemented as a priority queue, so that events are ordered by their deadlines.

The Figure 2 contains also some external entities the system interacts with:

- **GitHub**: used by the system to retrieve the code of GitHub repositories and to create new repositories.
- **GitHubActions**: configured by the students on their GitHub repository to automatically notify the system when a new submission is pushed to the repository.
- **StaticAnalysisService**: used by the system to evaluate the quality of the code of the submissions.
- **EmailService**: used by the system to send emails to the users of the system.
- **SSO**: used by the system to offer the users the possibility to signup and login with their preferred identity provider.

2.3 Deployment view

The following diagrams represent the deployment architecture of the system, highlighting the physical nodes on which the system will be deployed and the communication channels between them.

Figure 3 shows how the network infrastructure of the system is organized: a first firewall is used separate the system from the Internet, creating a DMZ which will contain all the services that must be accessible from outside the system (i.e. Web server and API gateway), while a second firewall is used to separate the DMZ from the internal network of the organization.

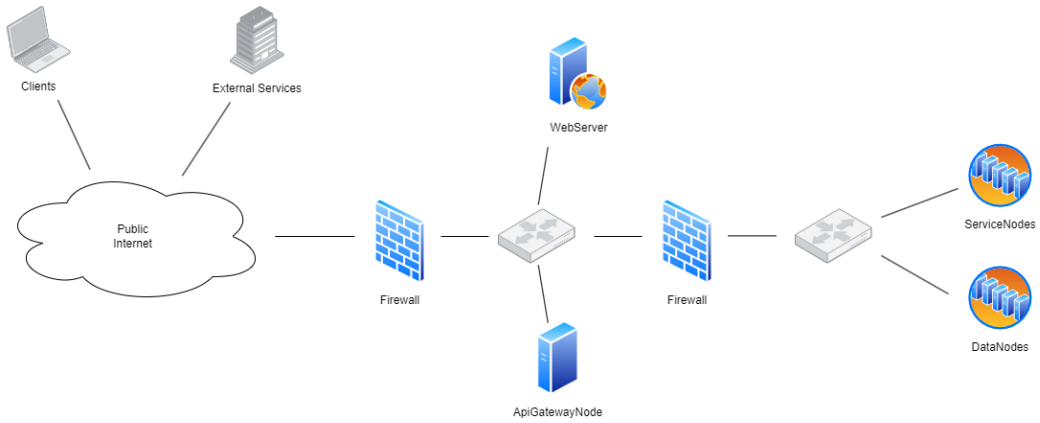


Figure 3: Network perspective deployment diagram

Figure 4 instead represents how different software modules will be deployed on physical nodes. These nodes have been colored according to the color scheme defined in the component view section, with the only difference that the red color here has been used to identify the user's web browser and the yellow one to represent the system's web server.
Important aspects that are worth noticing are:

- Queues are deployed on a single phisical node. As we can see from the artifacts, all queues are built over the same code but replicated in a way that each type of microservice uses its own: this should increase modularity and maintainability of the system. Moreover these queues are logged (i.e. also persisted in secondary memory) so that, in case of failure, a new service can be deployed and the previous state of the

queue can be restored: this allows the system to be fault tolerant by ensuring that no communication data is lost.

- Most of the architecture is deployed within Docker containers, providing a lightweight and scalable environment. Each instance of a microservice, instantiated as a separate container, follows a **thread-per-request model** to handle multiple incoming requests concurrently. This design choice optimizes resource utilization and responsiveness, allowing each microservice instance to efficiently manage concurrent requests through the creation of dedicated threads. The use of Docker ensures portability and consistency across different environments, enabling seamless deployment and scaling.
  The Service Registry instead is deployed in a non-containerized environment, since it is a critical component of the system and it is important to ensure its availability. In particular, in order to achieve this, a **master-slave schema** is used. The master instance is deployed on a different physical node with respect to other slave instances (only one in Figure 4): these have the task of monitoring the master instance and to copy its logs. In this way, if the master instance fails, one of the slave instances (e.g. chosen through a consensus algorithm) can be promoted to master so the system can continue to work from the point it left without any significant interruption.

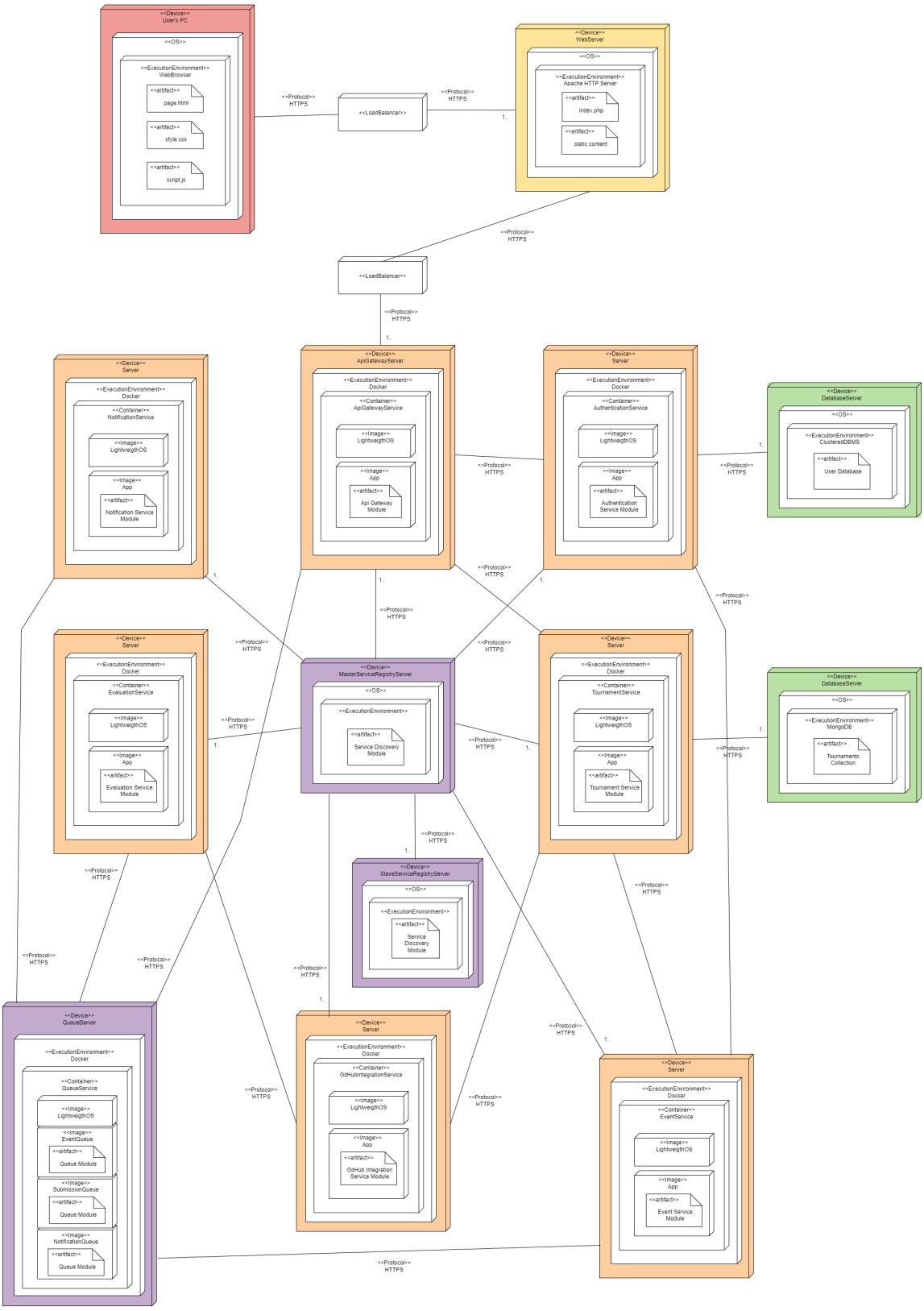


Figure 4: Deployment diagram

2.4 Component interfaces

The interfaces of the system's components are described in terms of the methods they offer, the parameters they require and the type of the values they return:

- **APIGatewayI**
  - login(credentials): userId
  - loginSSO(identityProvider,ssoToken): userId
  - signup(credentials,userType): userId
  - signupSSO(identityProvider,ssoToken,userType): userId
  - createBattle(userId,tournamentId,battleData): bool
  - getTournaments(): tournamentDataList
  - getEnrolledTournaments(userId): tournamentsDataList
  - createTeam(userId,battleId,teamName,teamPrivacy): bool
  - joinTeam(userId,battleId,joinOption,joinTextInput): bool
  - readTeamSettings(userId,tournamentId,battleId,teamId): teamData
  - updateTeamSettings(userId,tournamentId,battleId,teamId,gitHubRepoUrl): bool
  - getTournamentRank(tournamentId): tournamentRank
  - getBattleRank(battleId): battleRank
  - getTeamSubmissions(userId,tournamentId,battleId,teamId): submissionsDataList
  - joinTournament(userId,tournamentId): bool
  - createTournament(userId,tournamentData): bool
  - setManualScore(userId,teamId,score): bool
  - closeBattleConsolidationPhase(userId,tournamentId,battleId): bool
  - addCollaborator(userId,tournamentId,collaboratorEmail): bool
  - closeTournament(userId,tournamentId): bool

- **Auth2GatewayI**
  - authenticateUser(credentials): userId
  - createUser(credentials,userType): userId
  - completeLoginSSO(identityProvider,ssoToken): userId
  - completeSignupSSO(identityProvider,ssoToken,userType): userId
  - getUserType(userId): userType
  - getUserDataByEmail(userEmail): userData
- **UserModelI**
  - retrieveUserId(credentials): userId
  - createUser(credentials, userType): userId
  - retrieveUserData(userId): userData
  - retrieveUsersData(userIdList): userDataList
  - retrieveUserType(userId): userType
  - retrieveUserDataByEmail(userEmail): userData
  - retrieveAllStudentsData(): usersDataList
- **Auth2EventI**
  - getUserData(userId): userData
  - getUsersData(userIdList): userDataList
  - getAllStudentsData(): usersDataList
- **CommitI**
  - notifySubmission(teamId): none
- **Tournament2GatewayI**
  - createBattle(userId,tournamentId,battleData): bool
  - setManualScore(userId,teamId,score): bool
  - getTournaments(): tournamentDataList
  - getTournamentRank(tournamentId): tournamentRank

  - getBattleRank(battleId): battleRank
  - getEnrolledTournaments(userId): tournamentsDataList
  - createTeam(userId,battleId,teamName,teamPrivacy): bool
  - joinTeam(userId,battleId,joinOption,joinTextInput): bool
  - readTeamSettings(userId,tournamentId,battleId,teamId): teamData
  - updateTeamSettings(userId,tournamentId,battleId,teamId,gitHubRepoUrl): bool
  - getTeamSubmissions(userId,tournamentId,battleId,teamId): submissionsDataList
  - joinTournament(userId,tournamentId): bool
  - createTournament(userId,tournamentData): bool
  - closeBattleConsolidationPhase(userId,tournamentId,battleId): bool
  - addCollaborator(userId,tournamentId,collaboratorEmail,collaboratorId): bool
  - closeTournament(userId,tournamentId): bool

- **Tournament2EventI**
  - getTournamentUsers(tournamentId): userIdList
  - getKataData(battleId): codeKataData
  - getBattleStudents(battleId): userIdList
  - getTeamStudents(teamId): userIdList
  - setNextBattleState(battleId): none
  - closeTournamentSubscriptionPhase(tournamentId): none
  - endBattle(battleId): none
  - updateBattleRank(tournamentId,battleId,battleRank): none
  - endTournament(tournamentId): none

- **Tournament2EvaluationI**
  - setAutomaticScore(teamId,scores): none
  - getTeamRepoUrl(teamId): repoUrl

  - getBattleByTeamId(teamId): battleId
  - getBattleSettings(battleId): battleSettings
  - notifyTeamEvaluation(teamId): none
  - getBattleDataByTeamId(teamId): battleData

- **TournamentModelI**
  - retrieveEducators(tournamentId): userIdList
  - retrieveKataData(battleId): codeKataData
  - retrieveBattleCreator(battleId): userId
  - retrieveTournamentCreator(tournamentId): userId
  - retrieveBattleStudents(battleId): userIdList
  - retrieveTeamRepoUrl(teamId): repoUrl
  - retrieveTeamStudents(teamId): userIdList
  - setAutomaticScore(teamId,scores): none
  - setManualScore(teamId,score): bool
  - retrieveTournaments(): tournamentsDataList
  - retrieveTournamentByBattle(battleId): tournamentData
  - retrieveTournamentUsers(tournamentId): userIdList
  - retrieveTournamentState(tournamentId): tournamentState
  - retrieveBattleState(battleId): battleState
  - retrieveEnrolledTournaments(userId): tournamentsDataList
  - setBattleState(battleId,state): none
  - createBattle(userId,tournamentId,battleData): battleId
  - retrieveBattleByTeamId(teamId): battleId
  - retrieveBattleSettings(battleId): battleSettings
  - createTeam(userId,tournamentId,battleId,teamName,teamPrivacy): bool
  - addStudentToTeam(userId,tournamentId,battleId,teamId): none
  - retrieveTeamData(tournamentId,battleId,teamId): teamData

  - updateTeamData(tournamentId,battleId,teamId,gitHubRepoUrl): bool
  - retrieveTournamentRank(tournamentId): tournamentRank
  - retrieveBattleRank(battleId): battleRank
  - updateBattleRank(battleId,battleRank): none
  - retrieveTeamSubmissions(tournamentId,battleId,teamId): submissionsDataList
  - addStudentToTournament(userId,tournamentId): none
  - createTournament(userId,tournamentData): tournamentId
  - setTournamentState(tournamentId,tournamentState): none
  - retrieveBattleData(tournamentId,battleId): battleData
  - retrieveTournamentData(tournamentId): tournamentData
  - updateTournamentRankAndScores(tournamentId,tournamentRank,tournamentScores): none
  - updateBattleRank(tournamentId,battleId,battleRank): none
  - addCollaborator(tournamentId,collaboratorId): none
  - retrieveBattleDataByTeamId(teamId): battleData

- **PushEventI**
  - push(event): bool
- **PullEventI**
  - fetchHeadTimestamp(): timestamp
  - pull(): event
- **CreateRepositoryI**
  - createRepo(repoData): none
- **RetrieveSubmissionCodeI**
  - retrieveRepo(repoUrl): code

- **PushSubmissionI**
  - push(teamId): bool
- **PullSubmissionI**
  - isEmpty(): bool
  - pull(): teamId
- **PushNotificationI**
  - push(notification): bool
- **PullNotificationI**
  - isEmpty(): bool
  - pull(): notification
- **RegistryI**
  - getInstance(serviceType): instanceUrl
  - subscribeInstance(serviceType,instanceUrl): bool
  - heartbeat(serviceType,instanceUrl): bool

The system's components will exploit the following APIs, provided by already existing services or components:

- **TournamentDBMS API**
- **UserDBMS API**
- **SSO API**:
  - checkToken(ssoToken): userData
  - initializeSSO(ssoCredentials): ssoToken
- **GitHub API**:
  - createRepo(repoData): none
  - retrieveRepo(repoUrl): code

- **StaticAnalysis API**:
  - analyze(code,qualityAspect): score
- **Email API**:
  - sendEmail(recipientEmail,object,body): none

This representation of the system's interfaces is only for illustrative purposes: actual function calls will be performed through proper HTTP requests in which parameters will be included in the request body and the response will be encoded in JSON format. This is allowed by the use of REST APIs, as described in the next section.

2.5 Runtime view

TODO: check for missing text
In this section the most important runtime scenarios of the system are presented, highlighting the interactions between different components and, eventually, external entities.

For the sake of brevity, the retrival of dynamic content used to populate the pages rendered by the web server is not shown, so that only the most important exchanges can be represented.

2.5.1 Service Registry

The Service Registry is a critical component of the system, since it is responsible for the service discovery functionality, on which the microservice architecture is based. This is achieved by allowing 3 fundamental operations:

- **Service registration**: each microservice instance can register itself to the Service Registry, specifying the type of service it provides and its URL.

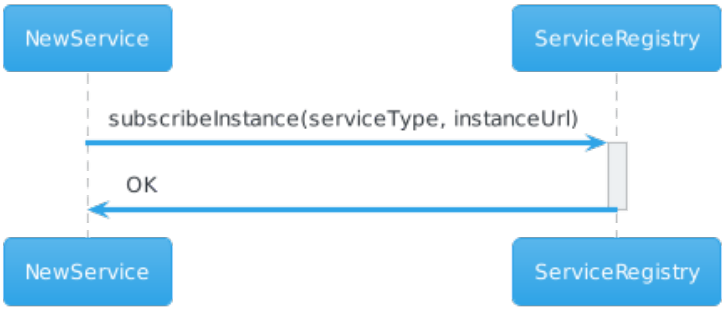


Figure 5: Service registration

- **Service discovery**: each microservice instance can query the Service Registry to retrieve the URL of an instance of a specific type.

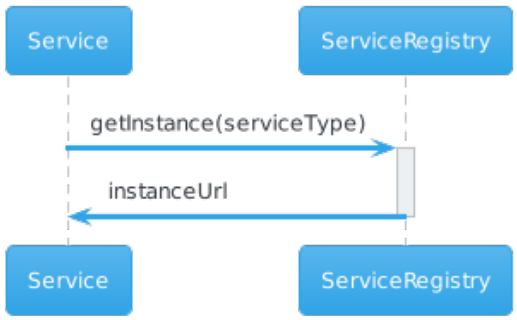


Figure 6: Service discovery

- **Heartbeat**: each microservice instance periodically sends a heartbeat to the Registry to notify that it is still alive and update its availability status.

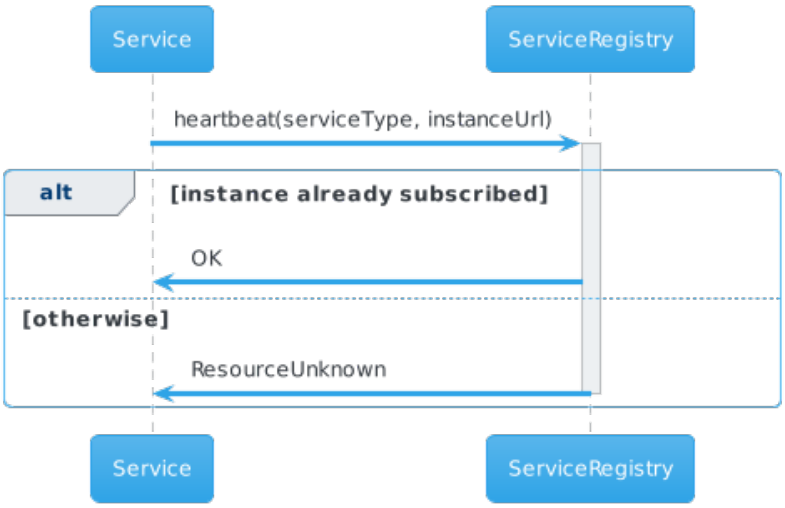


Figure 7: Heartbeat reception

2.5.2 Signup

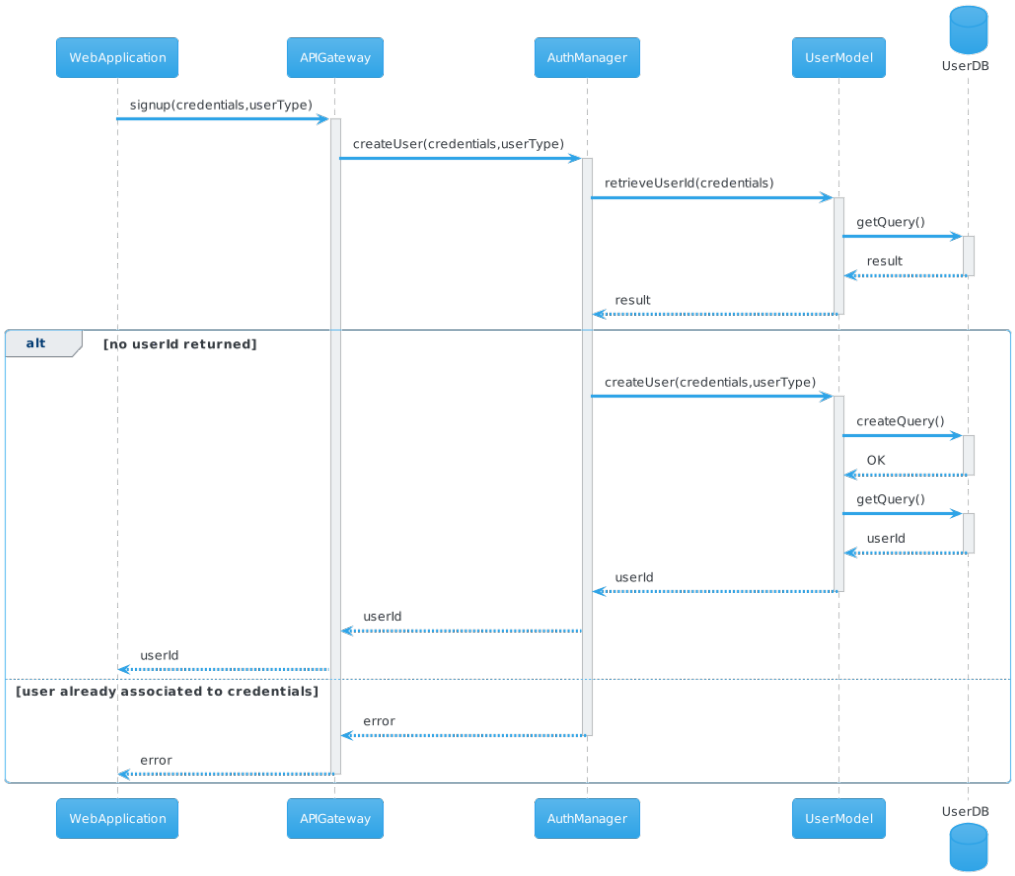


Figure 8: Normal sign up process

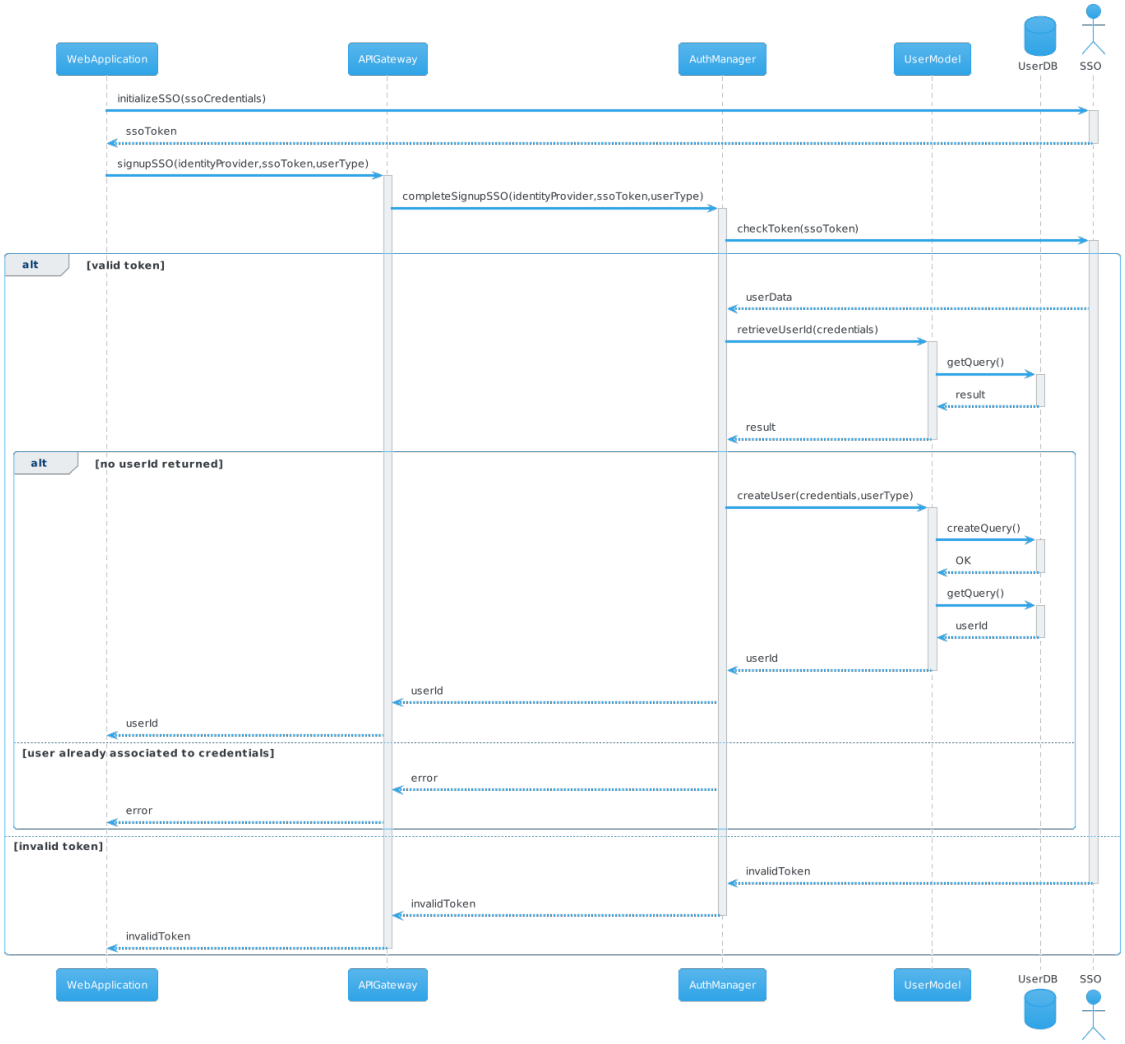


Figure 9: Sign up through SSO

2.5.3 Login

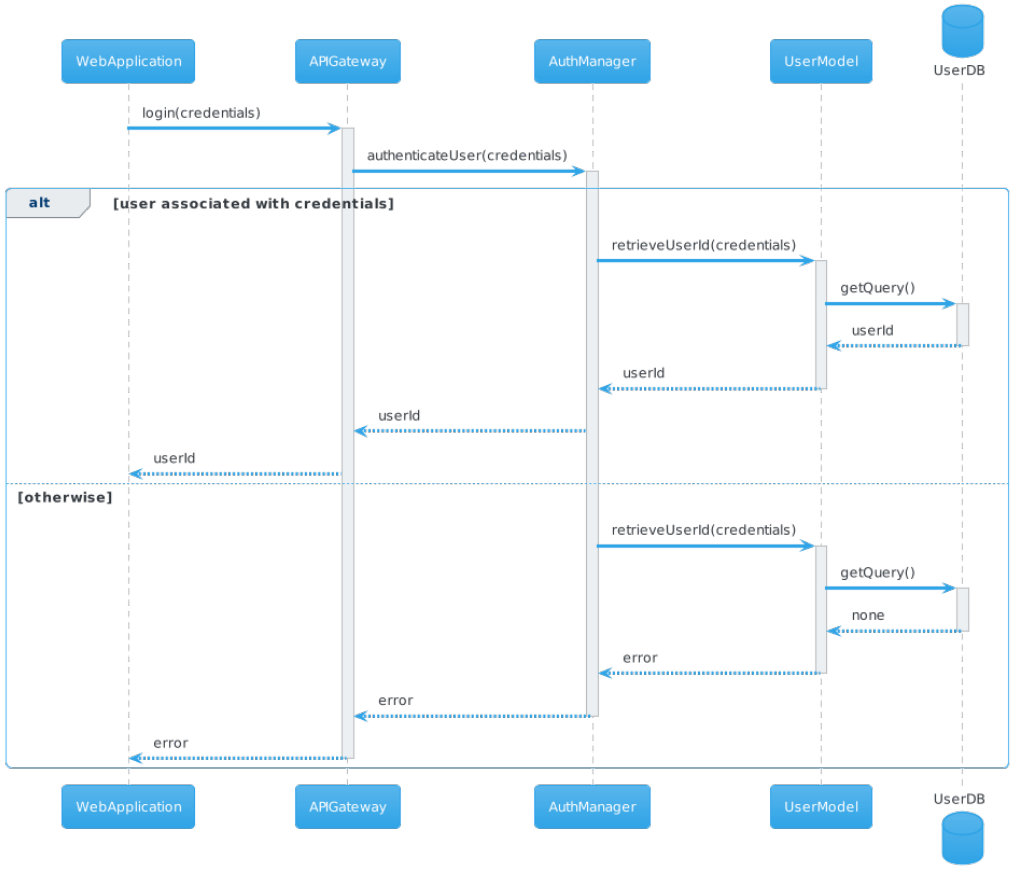


Figure 10: Log in process

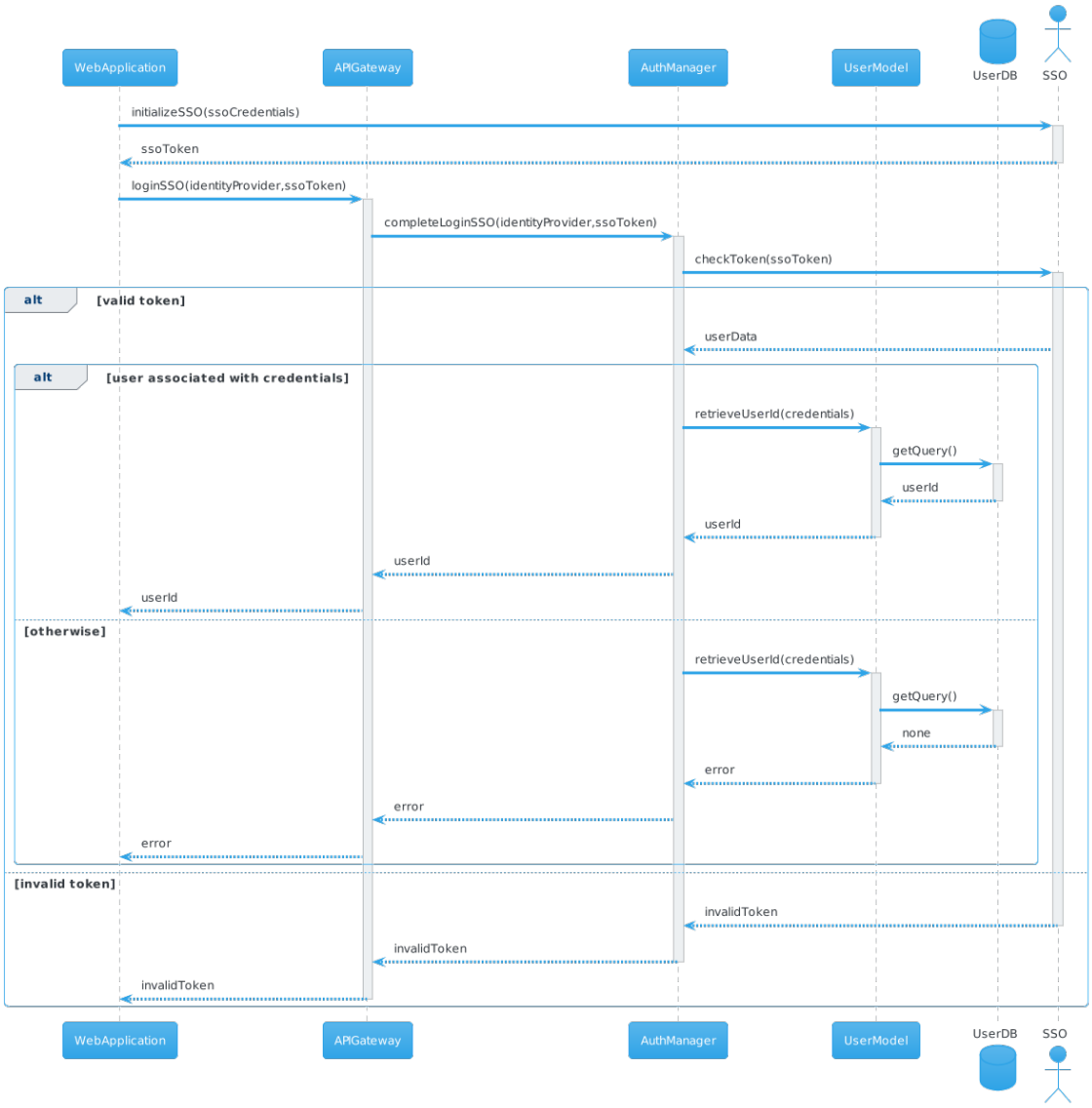


Figure 11: Log in through SSO

2.5.4 Create a tournament

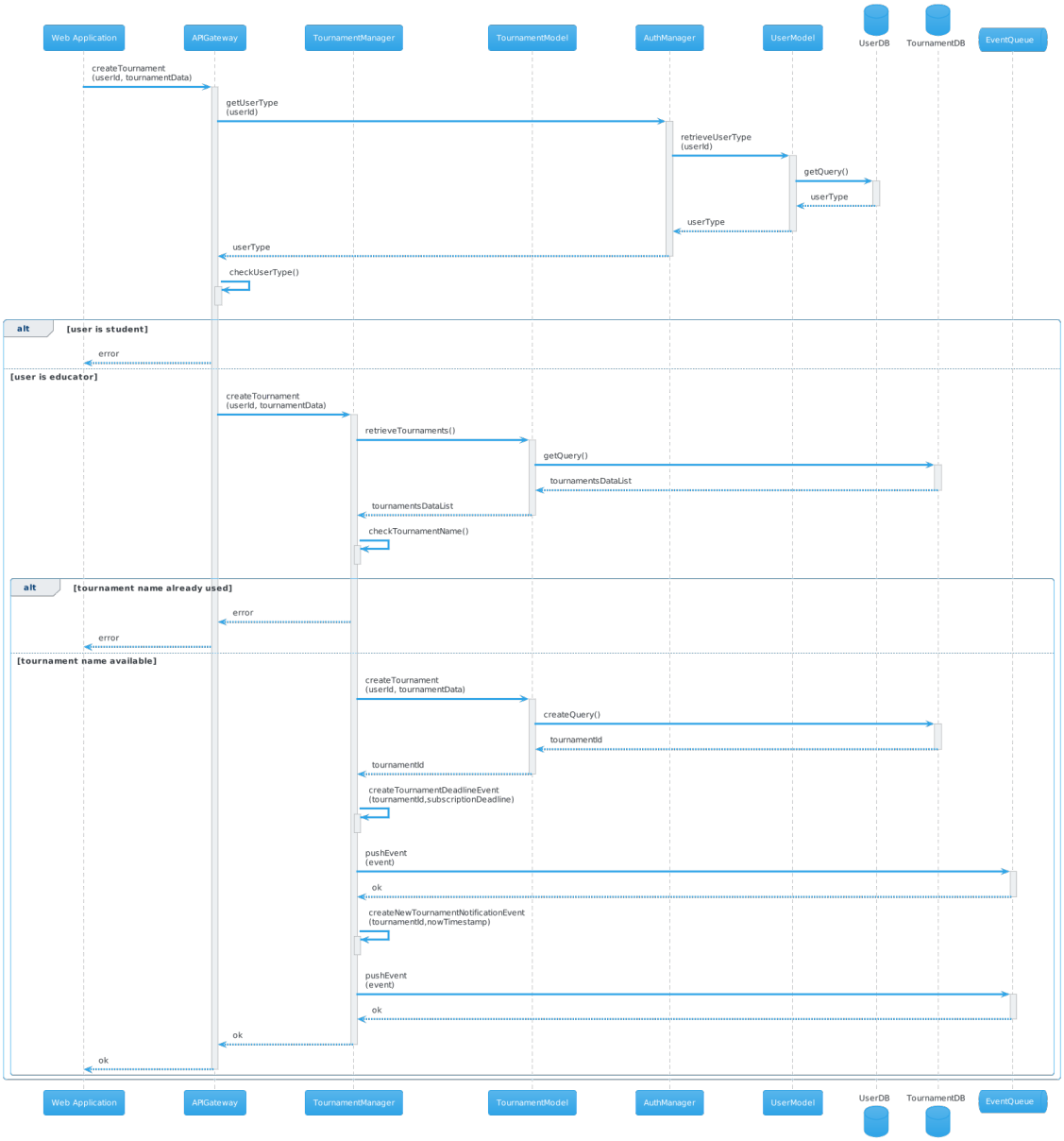


Figure 12: Tournament creation process

As it will happen for many other use cases, the asynchronous communication framework used between specific services of the system has the effect of decoupling the components, allowing them to work independently in separated points in time. For this reason some of the scenarios have been completed with an additional sequence diagram in which is shown the dedicated service pulling the message generated by the previous scenario, with all the effects of this event.

Moreover, some system's processes will generate multiple messages, so the additional diagrams will be multiple as well, as in this case.

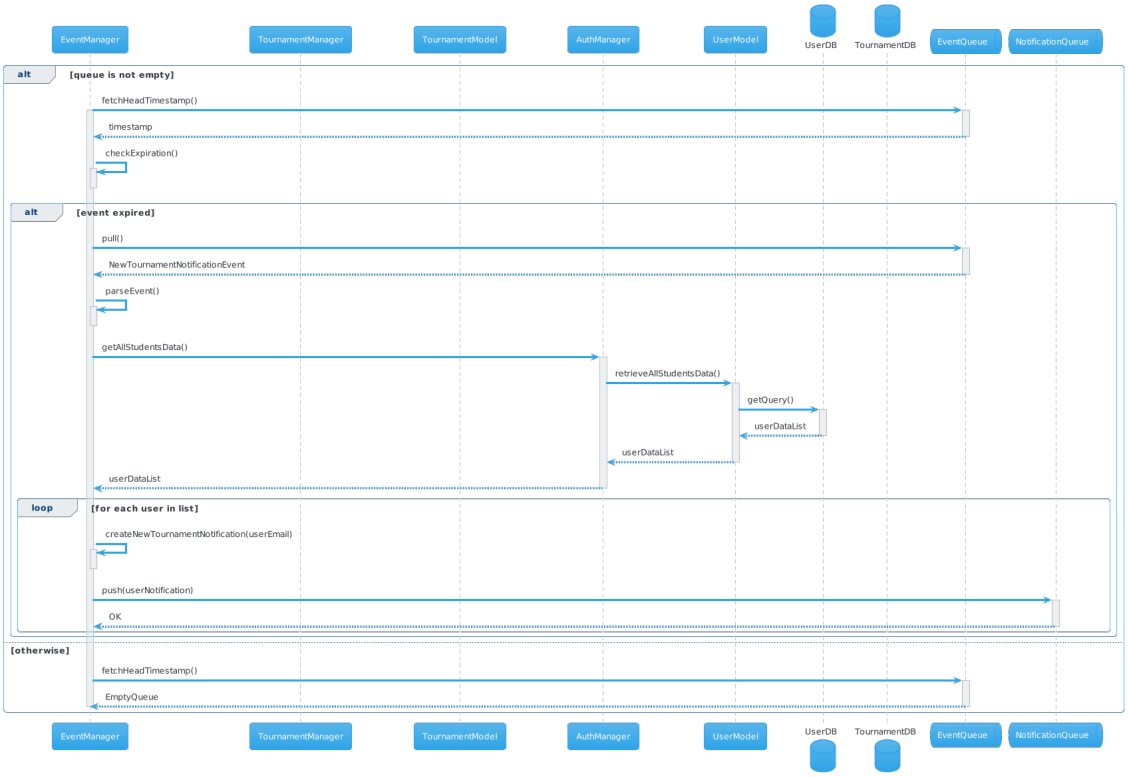


Figure 13: Pull of the notification event generated by the tournament creation

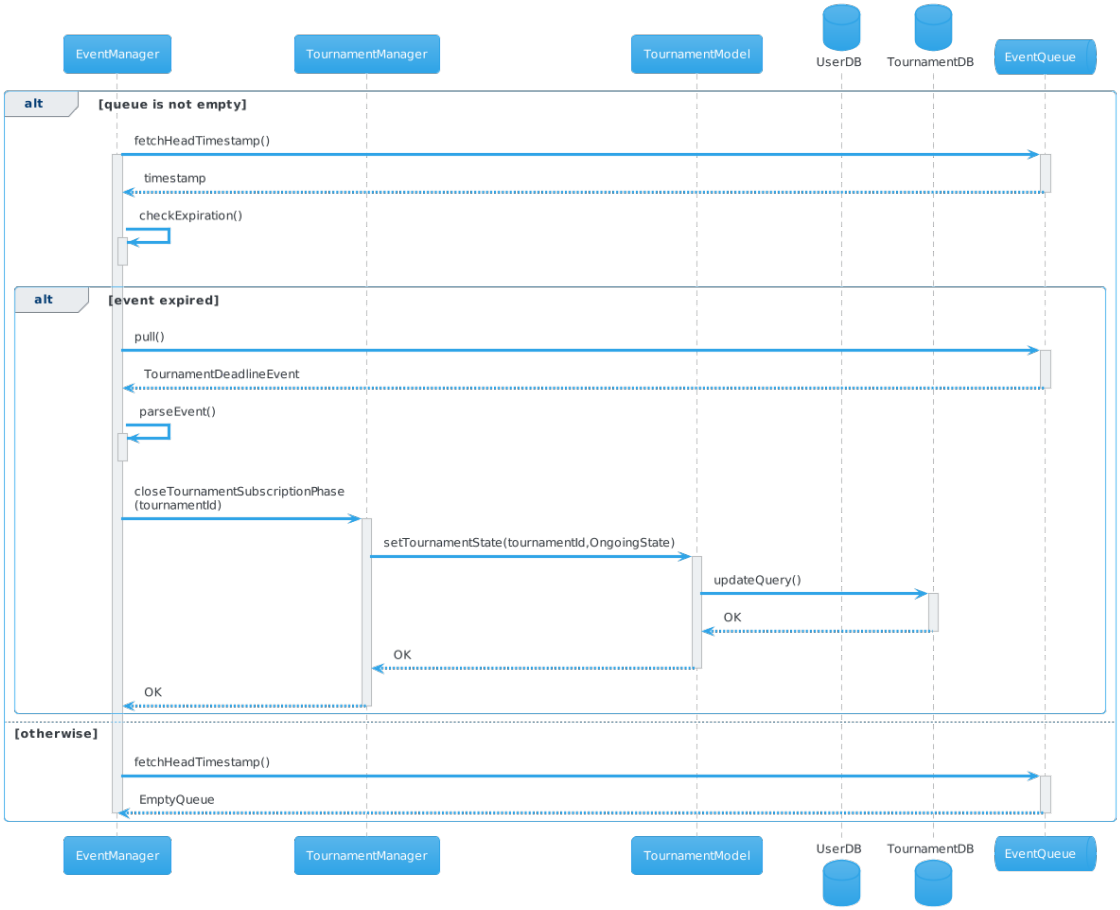


Figure 14: Pull of the deadline event, representing the end of the registration phase, generated by the tournament creation

2.5.5 Add a collaborator to a tournament

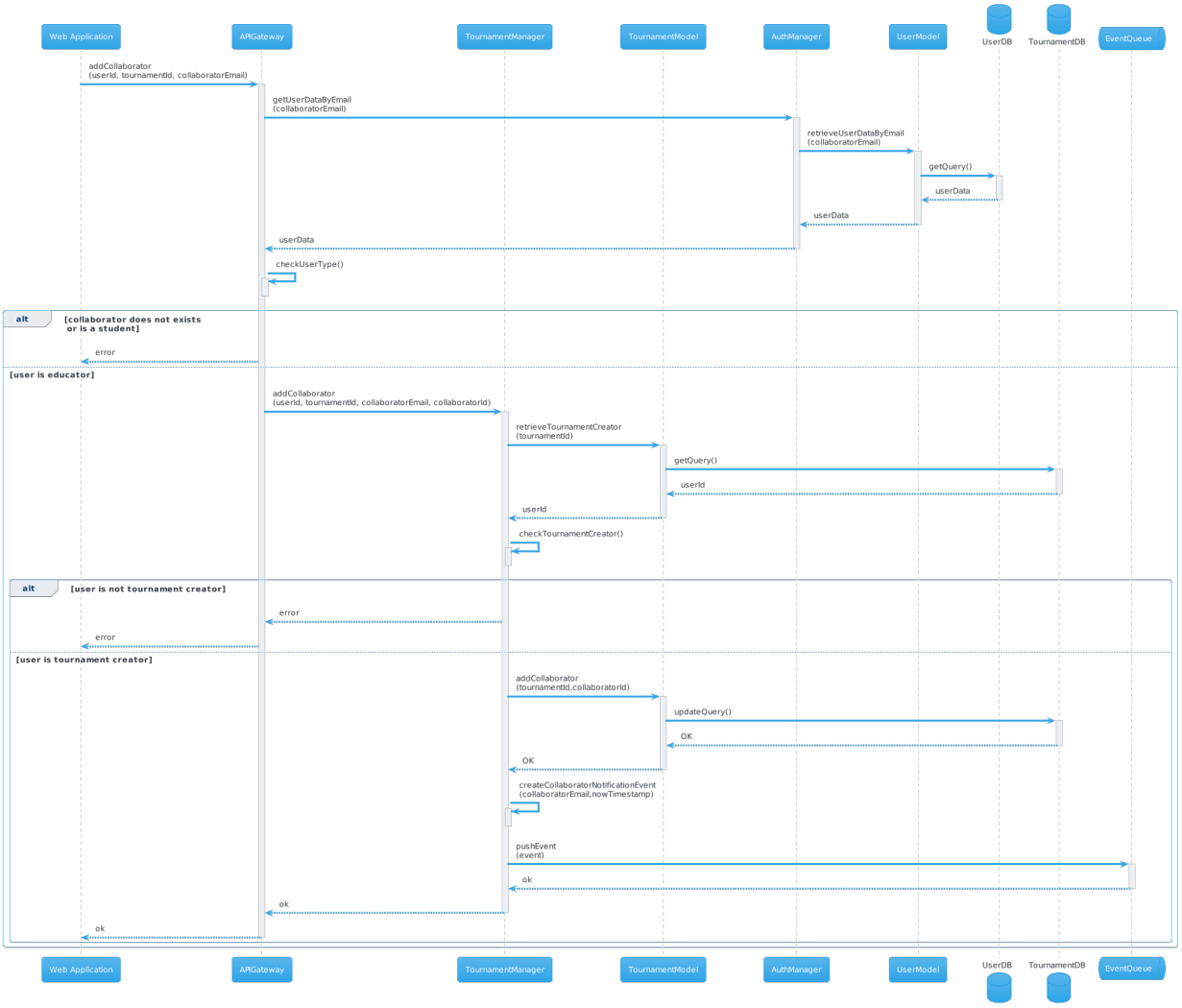


Figure 15: Process of adding a collaborator to a tournament

2.5.6 Join a tournament

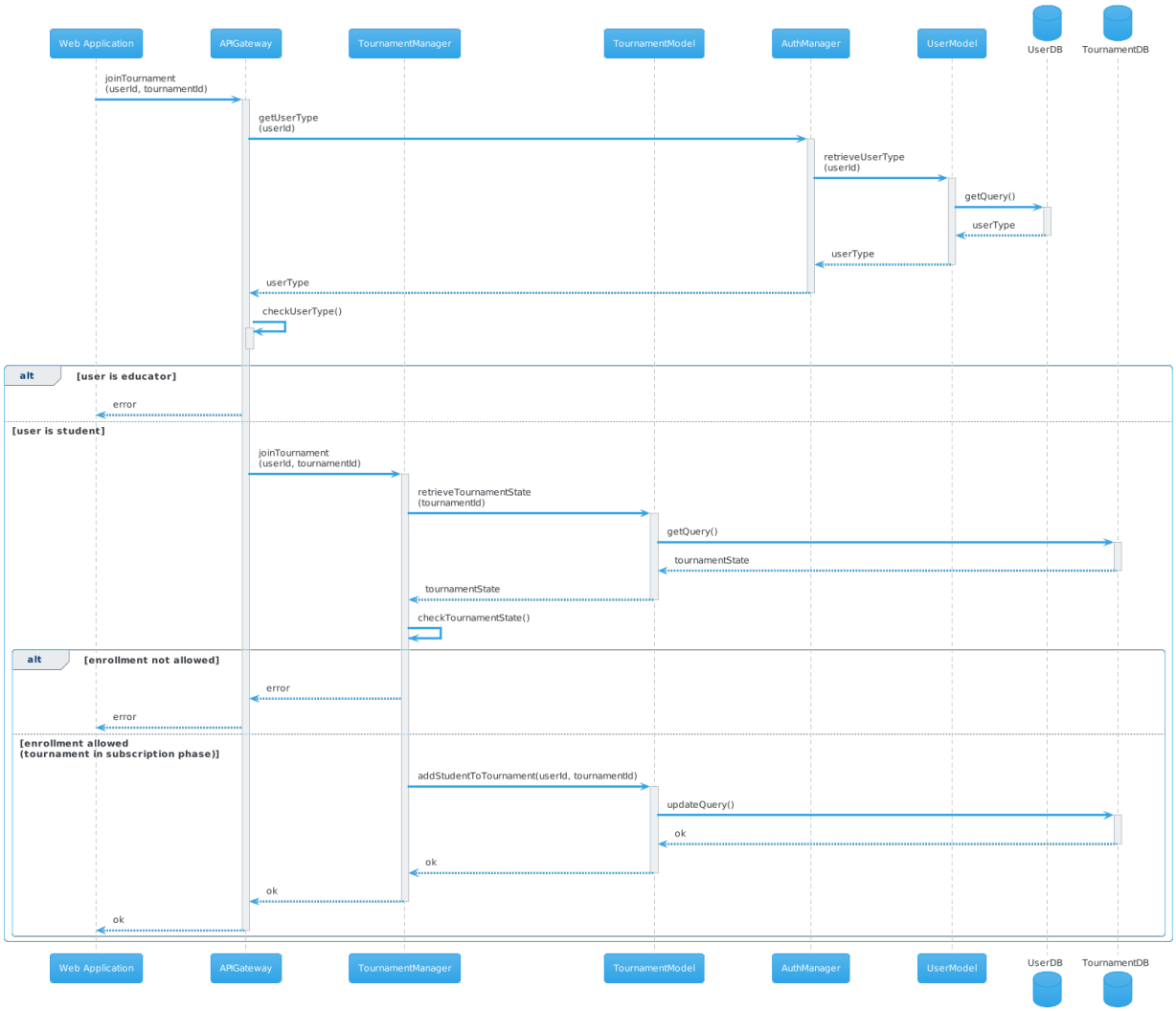


Figure 16: Process of subscribing to a tournament

2.5.7 Create a battle

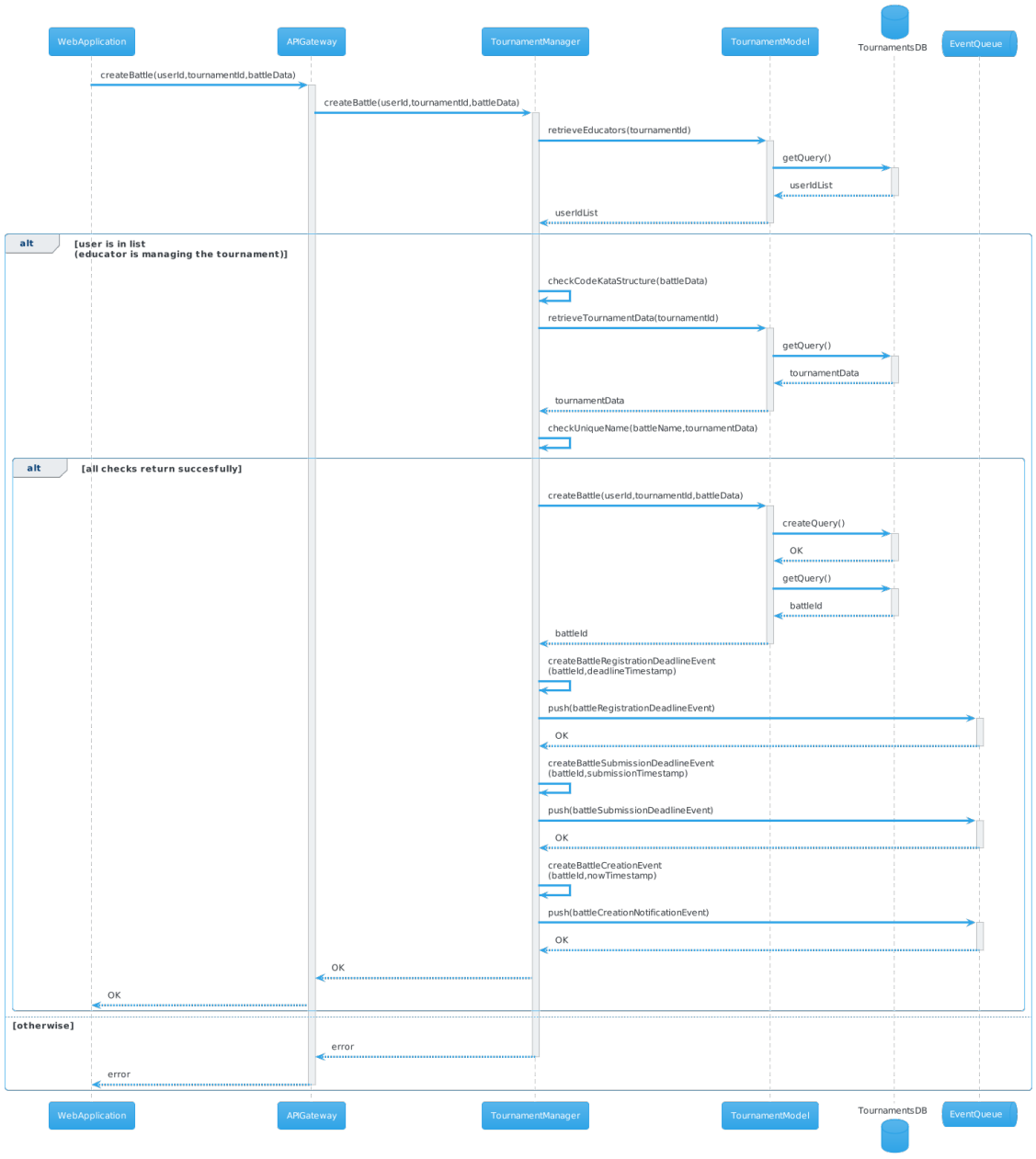


Figure 17: Battle creation process

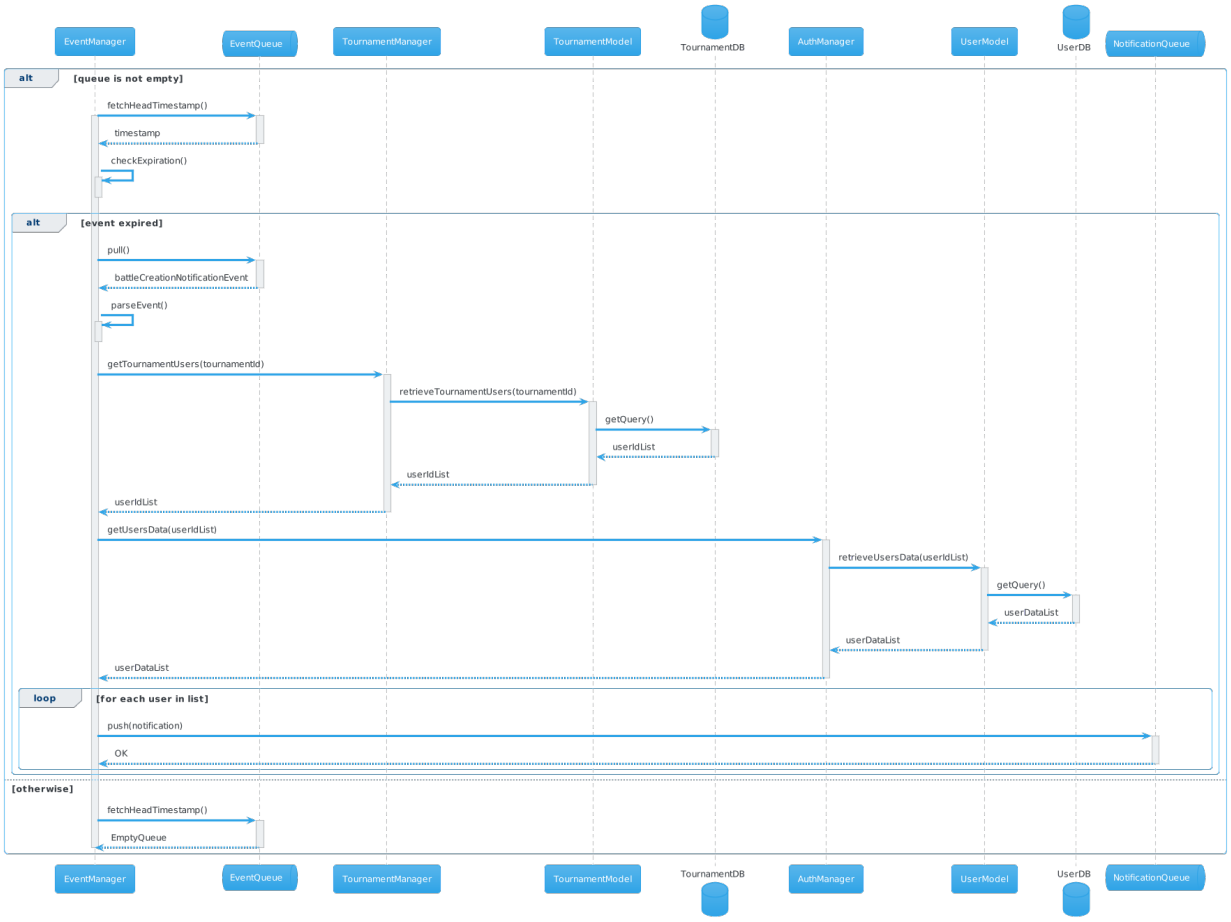


Figure 18: Pull of the notification event generated by the battle creation

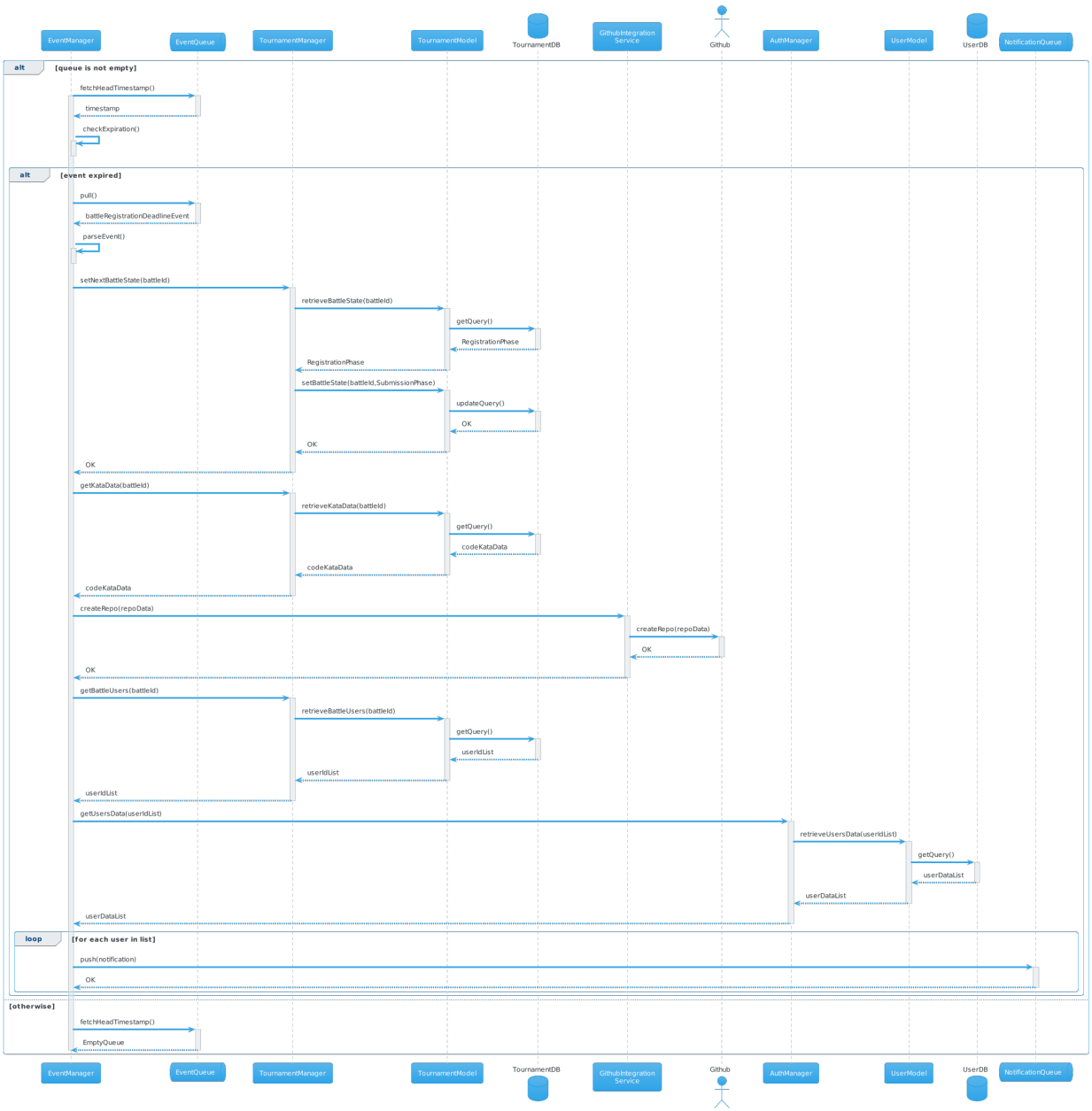


Figure 19: Pull of the deadline event, representing the end of the registration phase, generated by the battle creation

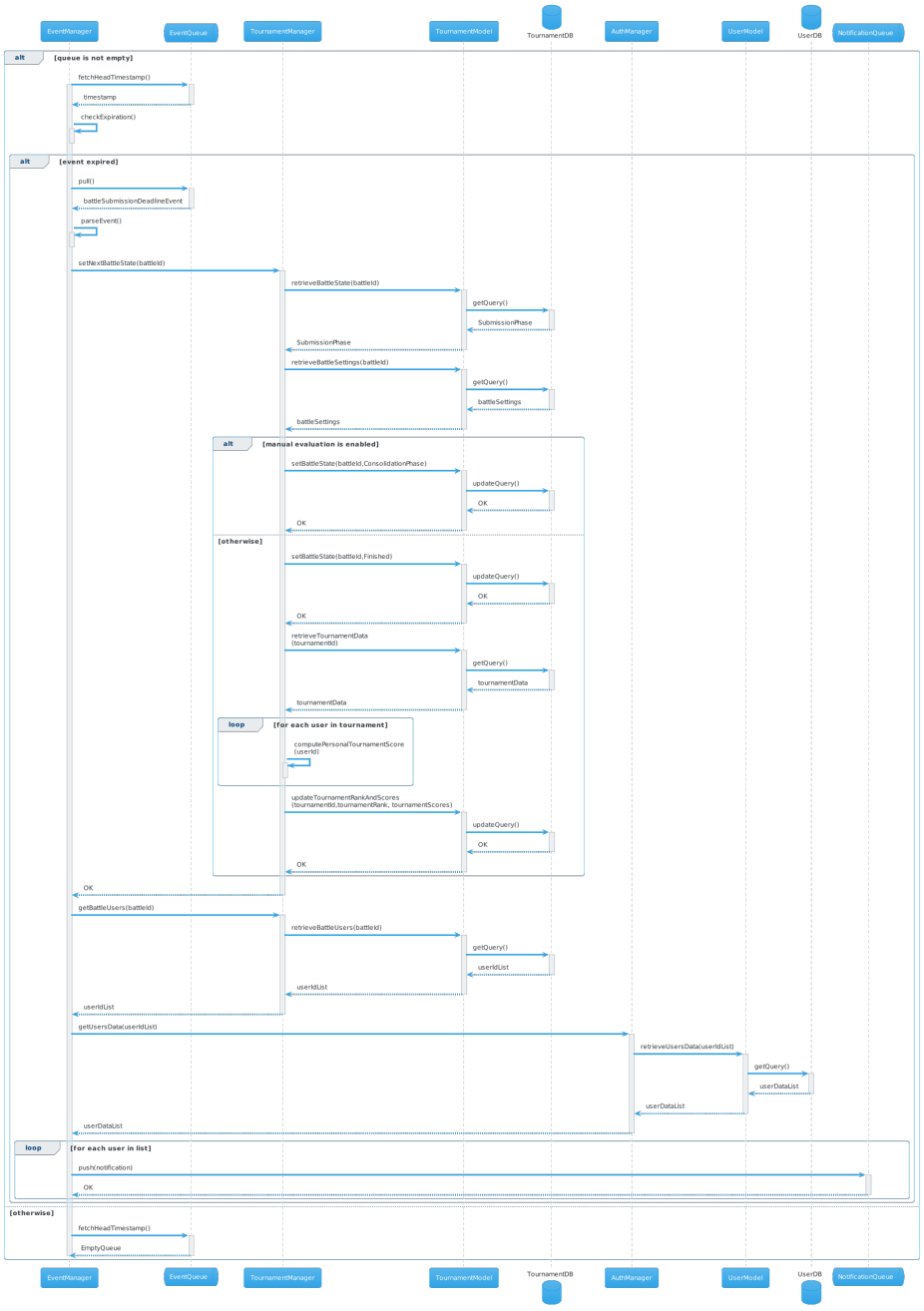


Figure 20: Pull of the deadline event, representing the end of the submission phase, generated by the battle creation

2.5.8 Create a team

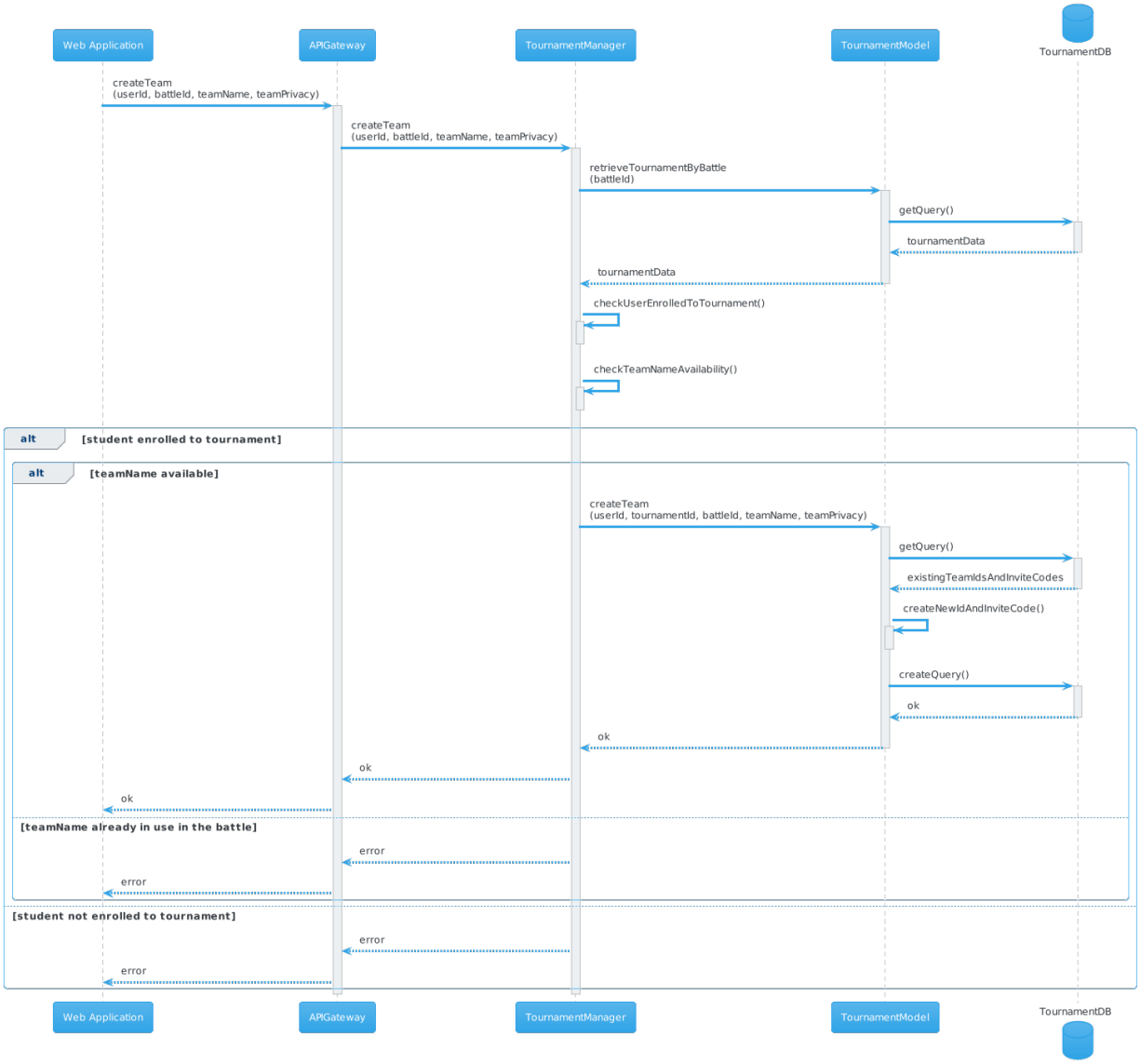


Figure 21: Process of subscribing to a battle by creating a team

2.5.9 Join a team

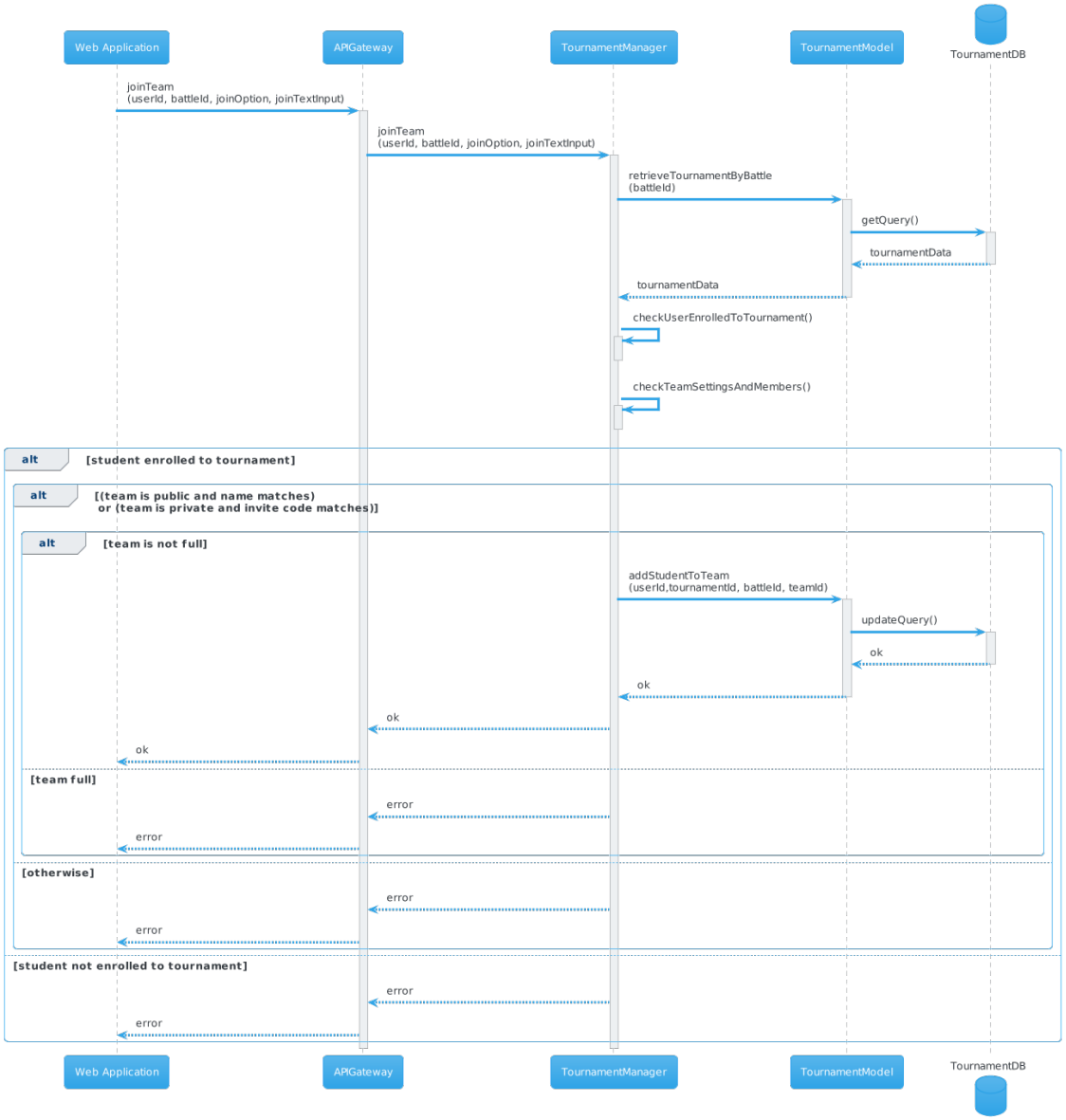


Figure 22: Process of subscribing to a battle by joining an already existing team

2.5.10 Configure team settings

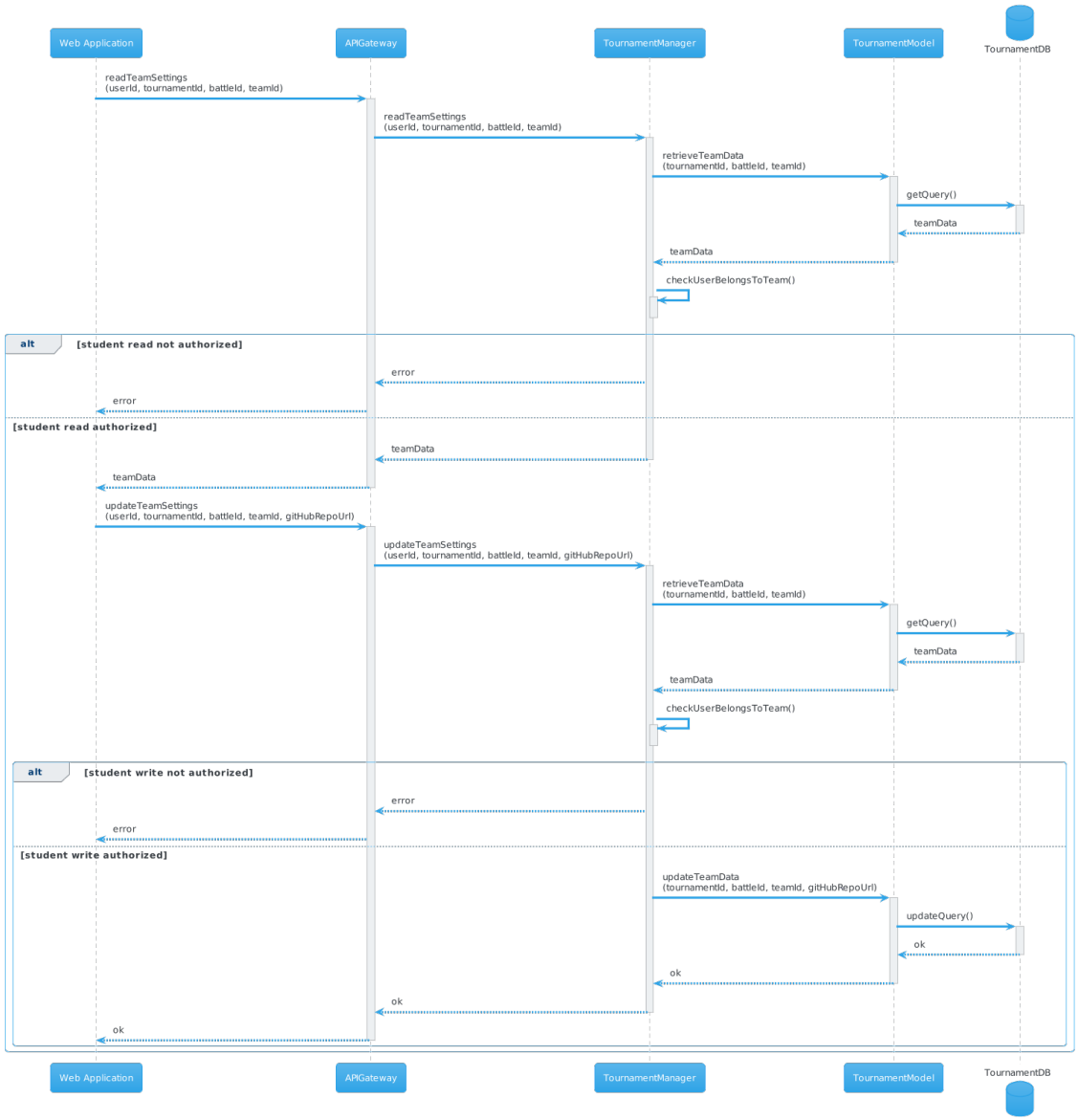


Figure 23: Team settings configuration process

2.6 Architectural styles and patterns

- **Client-Server**: the system can be seen as a client-server architecture from the point of view of the client, since it is composed of a front-end application (the client) and a back-end system (the server). The client is the web application that users interact with, while the server is the CKB Distributed System, which is constitued by multiple microservices. The client and the server communicate over a network to perform specific tasks and to exchange data. The client sends requests to one of the available webserver instances, which in turn processes the request and replies with the appropriate response. The webserver, to process the request, redirects it to an API Gateway instance, which is the entry point of the backend system.

- **Microservices**: the back-end, as previously described, is composed of multiple microservices. This architectural style has been chosen because it allows to decompose the system into multiple almost-independent components, each one responsible for specific tasks. This is particularly useful in terms of deployment and scalability, since each node can be scaled independently from the others, which is fundamental in a system that is expected to handle a large variety of requests with different workloads. Concerning maintainability, this also allows to easily modify and update the system, since each microservice can be updated independently from the others. Also availability is improved, since the failure of a single microservice may not affect the availability of some functionalities of the system.

- **API gateway**: the API gateway pattern is similar to the facade pattern from object-oriented design, but it is part of a distributed system reverse proxy or gateway routing for using as a synchronous communication model. More in details, it acts as the system's facade, so it provides a single entry point to the APIs allowing the encapsulation of the underlying system architecture. It also works as a reverse proxy, since it route requests to internal microservices endpoints and is responsible of cross-cutting concerns such as load balancing, caching, security.

- **REST APIs**: the REST architectural style has been chosen to implement the communication between the client, the system and microser-

vices, since it is the most common and widely used style for building web APIs. Additionally, REST APIs are stateless, which means that no client context is stored on the server between requests, simplifying implementation and horizontal scaling.. HTTPS is used as communication protocol and data exchanges are performed through JSON objects.

- **Model View Controller**: the MVC pattern decomposes the program logic into three interconnected elements:
  - **Model**, which represents the dynamic data structure of the application, independent of the user interface. It is responsible for managing the data of the application and receives user input from the controller.
  - **View**, which is responsible for the presentation of the data to the user. It sends user input to the controller.
  - **Controller**, which is responsible for handling user requests and updating the model and the view accordingly. The state of the controller is inferred from the state of system which is represented by the model.
- **Routing patterns**: the ServiceRegistry component enables the service discovery. Each microservice mantains a local cache of the registry, and periodically refreshes it by contacting the ServiceRegistry. Microservices use the local cache to contact other microservices and spread the load across the available instances following a round-robin schema. The ServiceRegistry is replicated through a master-slave schema, to ensure the availability of the service discovery functionality.
- **Security patterns**: the system uses the OAuth2 protocol to authenticate users. This protocol allows the system to delegate the authentication to an external identity provider. The system only receives a token from the identity provider, which is used to identify the user in the system.
- **Communication patterns**: both synchronous and asynchronous patterns are used in the system. The asynchronous communication pattern, which is implemented through the use of one-way queues, has been used for the communication between microservices that do not require any response from the receiver. This event-driven approach enforces

the decoupling between microservices and concurs to the scalability of the system, since multiple instances of the same microservice can consume from the same queue and process the requests concurrently. The choice of a hybrid approach, is driven by the following advantages related to the synchronous communication:

- **Reduced latency**: synchronous requests allow services to retrieve data with minimal delay, providing fast response times. This is important for user-facing services like the API Gateway.
- **Simplicity**: does not require the use of queues and complex mechanism to handle bidierectional communication. This reduces complexity for services with simple integration needs.
- **Transactional integrity**: allow several operations to be executed as an atomic transaction, ensuring data consistency across services. Useful when multiple services need to update related data.
- **Request-response pattern**: some use cases have an inherent request-response flow that fits synchronous communication. For example, the API Gateway may need to query other services to aggregate the response for the client.

2.7 Other design decisions

In this section some other design decisions are presented and justified.

- **Database solutions**: the database technologies and models have been chosen analyzing the most common interactions and queries that the system will perform. A **document-oriented** database fits particularly well the TournamentDatabase, given the hierarchical structure of tournaments, battles and teams. The use of a unique tournament collection allows to perform useful and common queries without performing complex joins, which would be required in a relational database. Additionally, no drawbacks are expected in terms of redundancy, as no users data is stored in the TournamentDatabase (only the userIds are stored). However, **indexes** will be needed to quickly retrieve some data without performing a full scan of the collection, such as tournaments and battles data related to a specific user. These kind of indexes are indeed useful to populate the customized user's homepage or accessing

battle or team data directly using their respective ids. A document-oriented database is also a good choice for the tournament collection, as it allows to conveniently store complex and unstructured data, such as tournament/battle scores and ranks within the same data structure. The UserDatabase instead is a **relational** database, since the data stored in it is structured and simple and no complex queries are expected to be performed on it.

- **Code Kata files**: as described in the RASD document, the educators are asked to provide some files to create a new battle. The system expects the project files to be correctly formatted, in order to be able to create the battle repository and to evaluate the submissions. The uploaded file must be a zip file containing the following:
  - **src** folder containing the source code of the project, including build automation scripts (e.g. pom.xml for Maven projects, build.gradle for Gradle projects, make files for C projects and so on).
  - **test.txt** file containing the test cases to be used to evaluate the submissions. The file must contain one test case per line, and each test case must be formatted as follows:
    input1,input2,...,inputN;output

3 User Interface Design

TODO: check impagination and write about sequence of mockups instead of interaction diagrams

3.1 Signup and Login

The signup page asks to the unregistered user to provide some personal information like name, surname, email address and password. Moreover, the user has to specify if he is a student or an educator by flagging the corresponding checkbox. In case he is a student, he is also asked to provide his GitHub's username. Alternatively, users can choose to sign up using third party services like Google or GitHub.

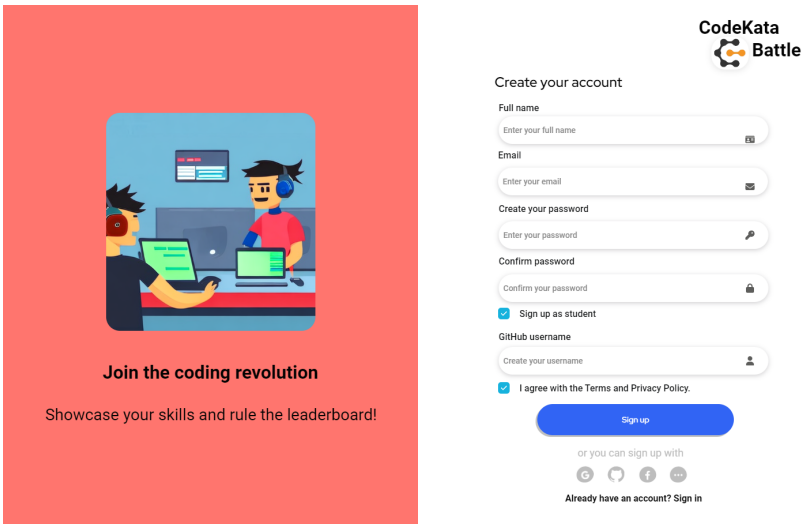
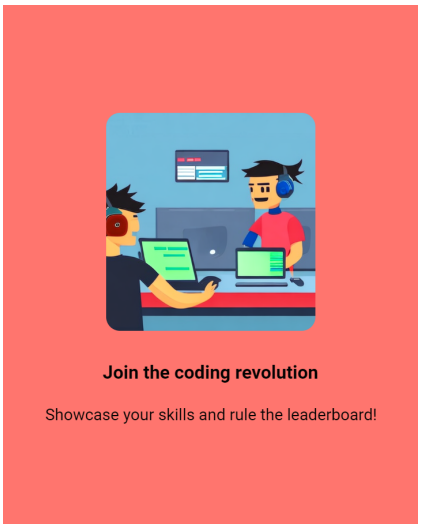


Figure 24: Signup page

The login page asks to the registered user to provide his email address and password. Alternatively, users can choose to login using the same third party services.



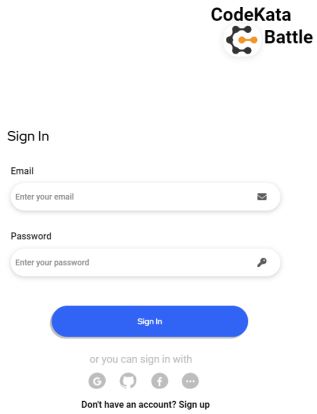


Figure 25: Login page

3.2 Home page

The home page is the first page that the user sees after logging in. If the logged user is a student, the homepage contains a brief overview of platform's ongoing tournaments, battles the user is participating in, a calendar with upcoming deadlines and some statistics of past battles. Educator's homepage is similar, but will include some information about tournaments and battles he is responsible of.

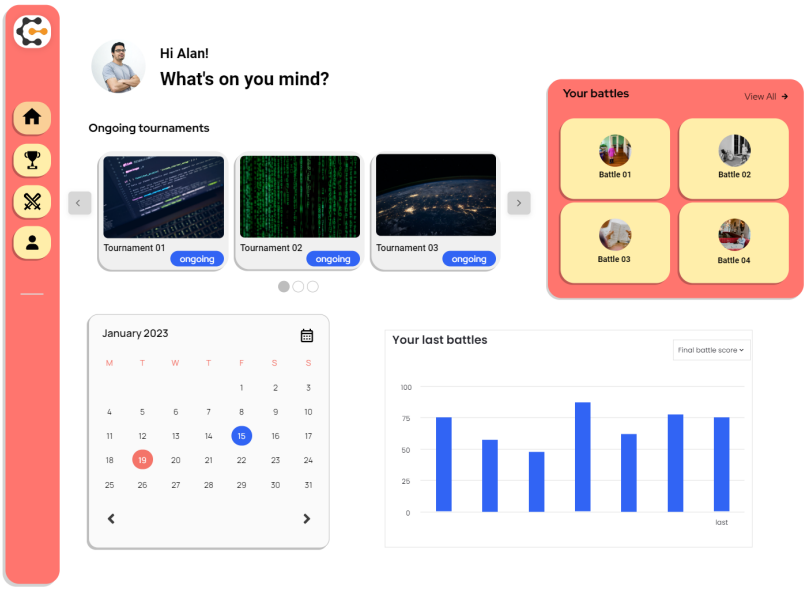


Figure 26: Home page from the perspective of an user logged as student

3.3 Tournaments

The tournaments section will provide to the student a list of all ongoing tournaments in the platform and a list with all tournaments the student is enrolled in. The page will contain also some collections, to access for example popular tournaments or all past tournaments. By clicking on a tournament, the user will be redirected to the tournament page.

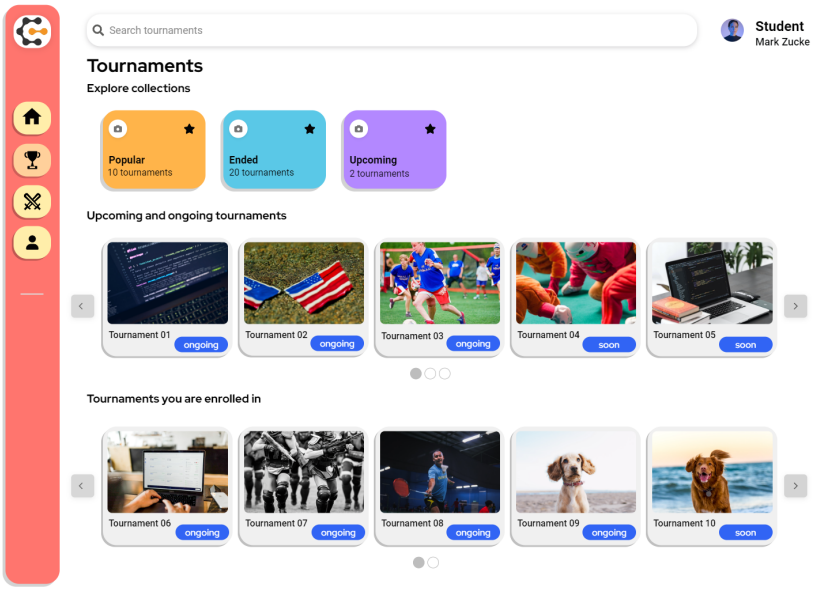


Figure 27: Tournaments page from the perspective of an user logged as student

The corresponding page for educators will show basic similar information about tournaments, but will also provide a button to create a new tournament.

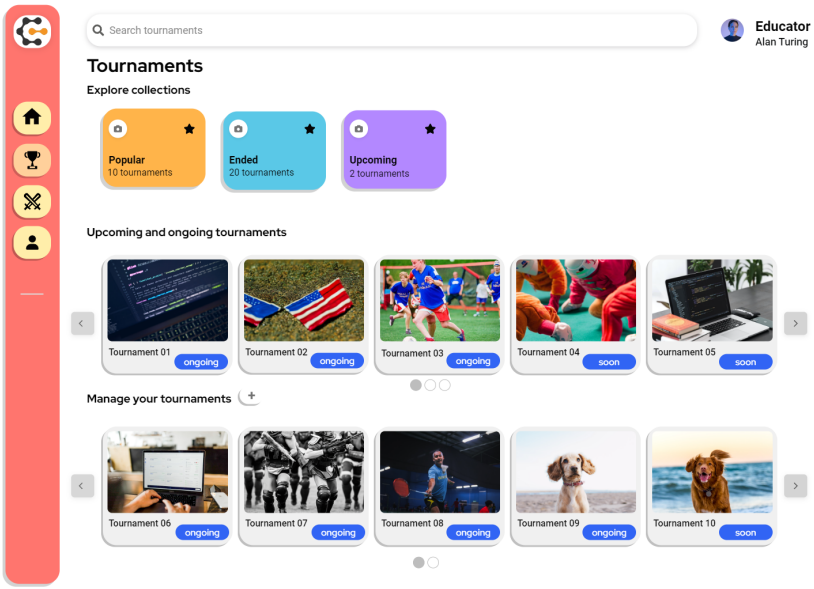


Figure 28: Tournaments page from the perspective of an user logged as educator

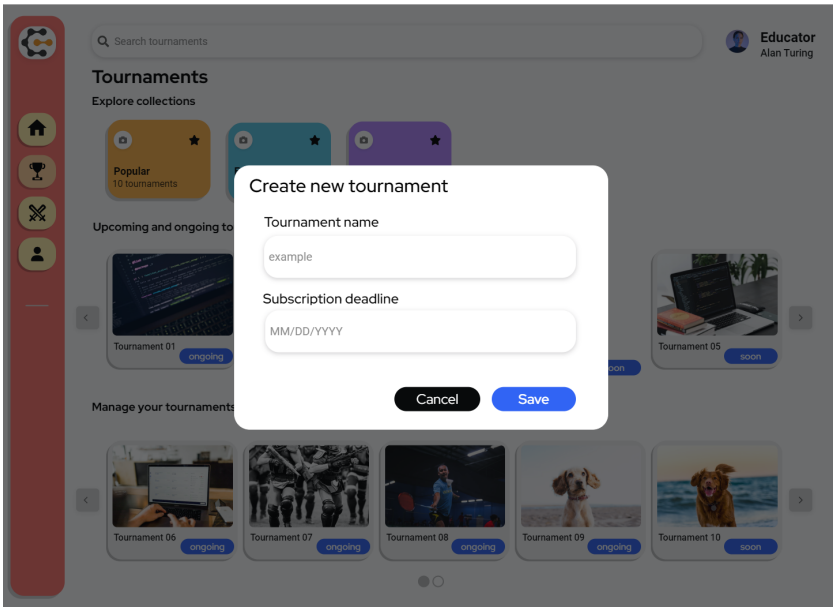


Figure 29: Page used from educators to create a new tournament

The tournament page offers to the educator owning the tournament the possibility to manage it. He will be able to click the "Add collaborator" button to add another educator to the tournament by email address. Moreover, the page offers the "collaborators" tab that can be used to check and manage the list of collaborators. The last tab called "leaderboard" allows the educator to check the ranking of the students enrolled in the tournament. The main tournament page contains the list of all the battles created within the tournament. By clicking on a specific battle, the educator will be redirected to the battle's page. Most importantly, the educator can create a battle by clicking on "Create battle" button.
The student's tournament page will share most of the content, but the options to manage the tournament. The same page, if the tournament is not started yet, offers the possibility to the student to subscribe to the tournament and by default shows a list of partipants.

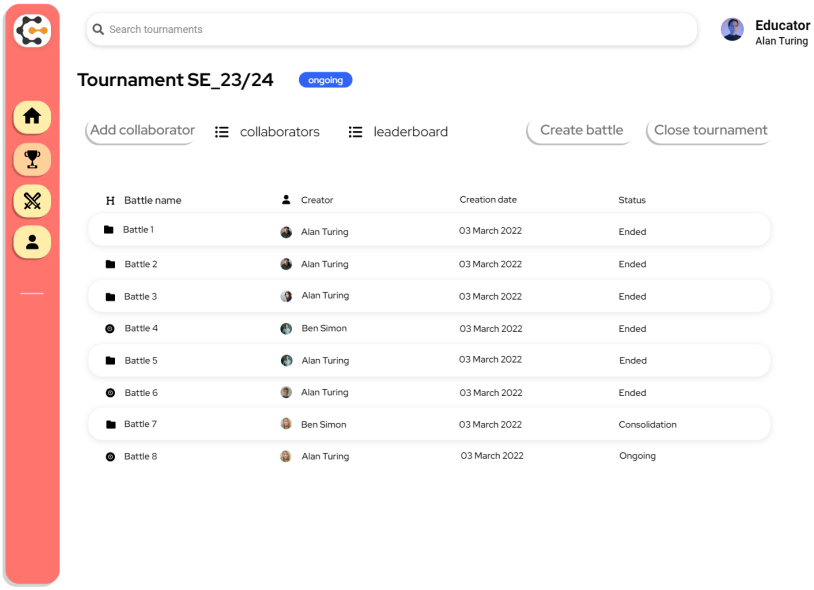


Figure 30: Page used from educators to manage the main settings of an ongoing tournament

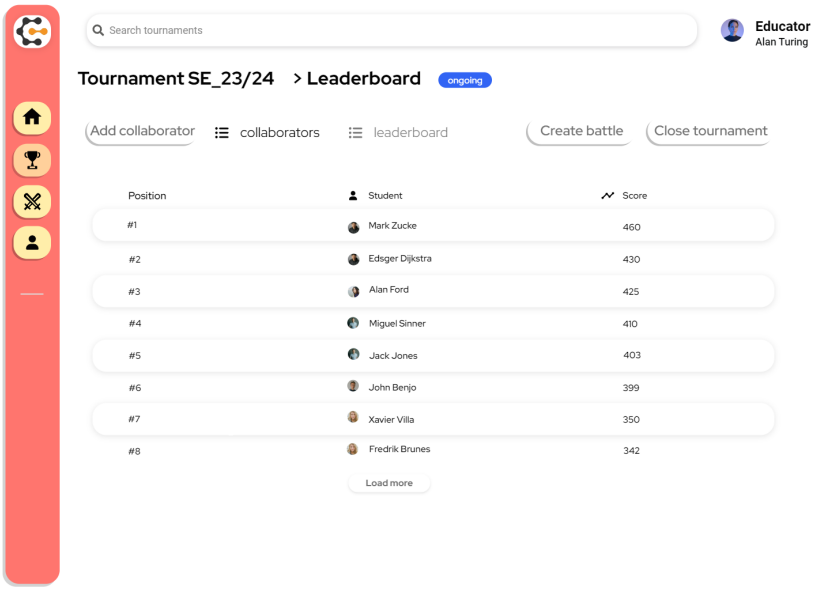


Figure 31: "Leaderboard" tab available to both educators and students to check the ranking in tournament

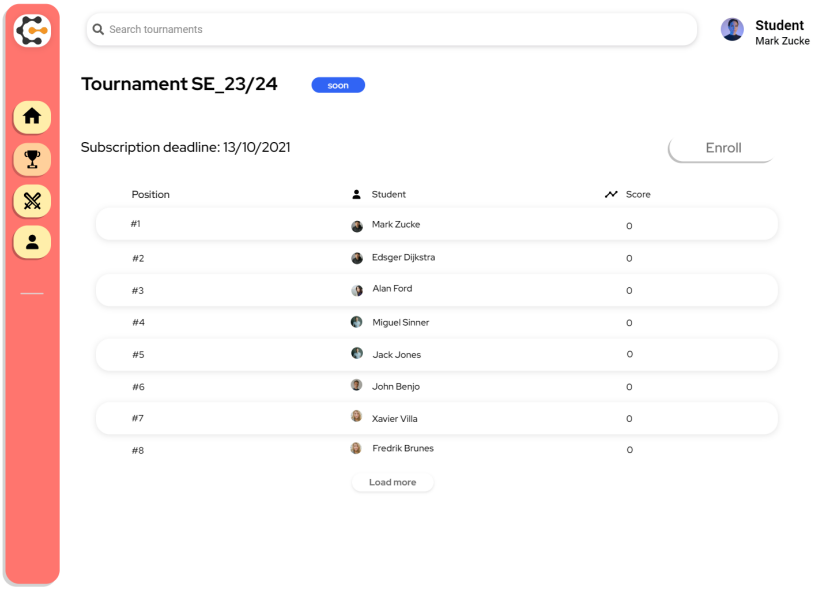


Figure 32: Page used from students to subscribe to a tournament

3.4 Battles

After selecting the option to create a battle, the educator will be redirected to the page used to create a new battle. The page will ask the educator to provide the name of the battle, the description, the deadline for the registration and the deadline for the submission. Moreover, the educator will be able to select the minimum and maximum number of students per group allowed for the battle. The educator will be able to set some information about the battle and to configure the battle settings for the evaluation. He will be required to upload the Code Kata files of the battle, which contain the test cases and the basic template of the project that students will have to complete with their solutions.

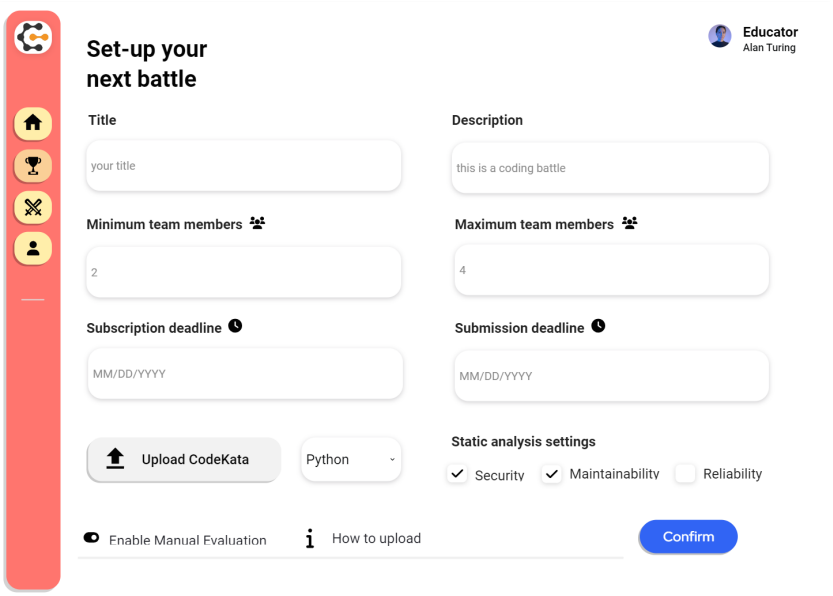


Figure 33: Page used from educators to create a new battle

When a battle is in subscription phase, the battle's page will offer to the student the options to subscribe to it by creating a new team or joining an already existing one. If the student decides to create the team, he will be asked to input the name of the team and its privacy setting. Instead, if the student want to join a team, the system will ask him to insert the name of the team (if public) or the invite code (if private). The page will also show the list of all the teams enrolled in the battle, to facilitate the team formation.

Teams can be clicked in the leaderboard and a small pop-up with the list of members will be shown.

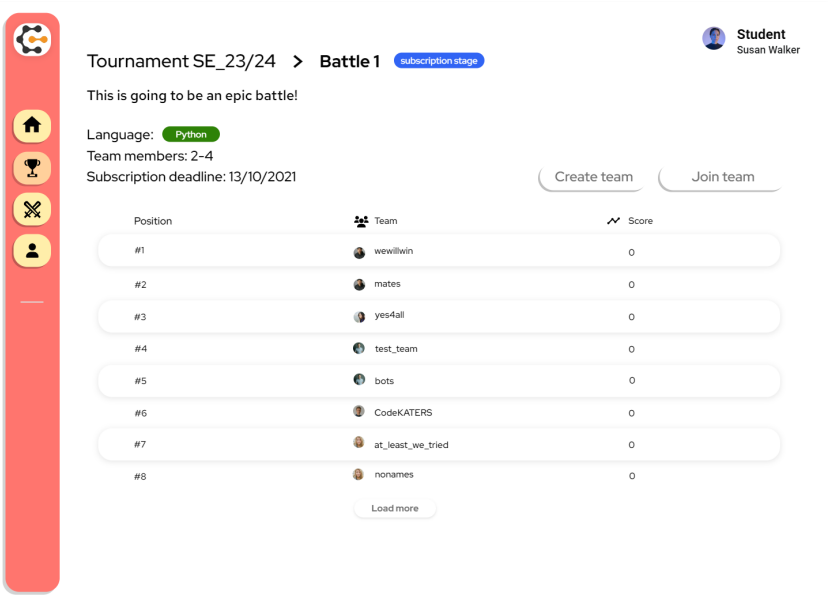


Figure 34: Page used from students to subscribe to a battle

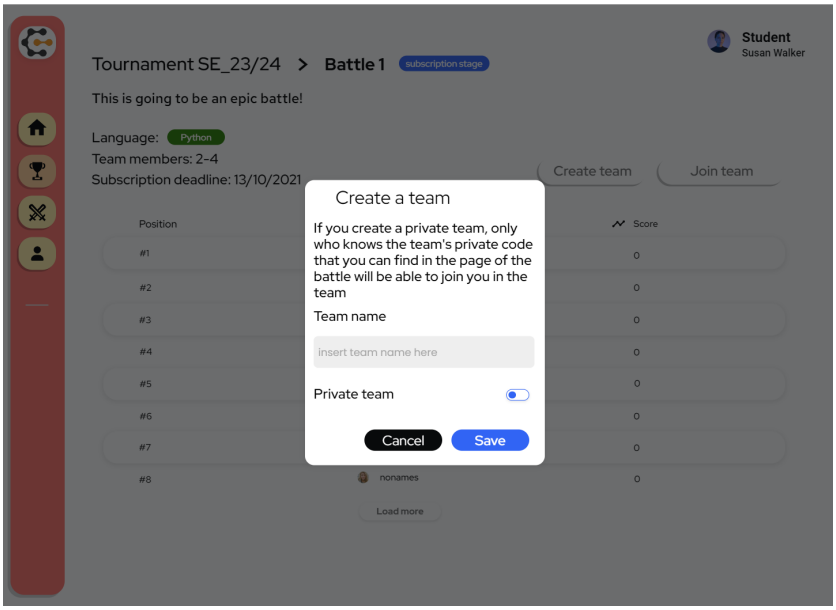


Figure 35: Page used from students to create a new team

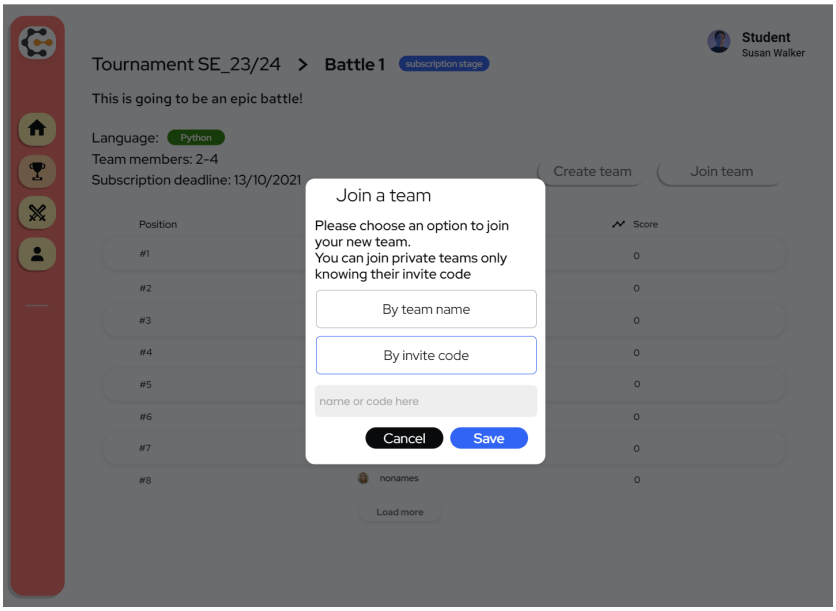


Figure 36: Page used from students to join a team

The page of a battle which is in the submission phase, will show by de-

fault the leadeboard of the teams. By clicking on "Your team" button, the student will be able access a section that contains important team settings and the information related to the submissions.
In case the manual evaluation is not enabled, this page will have similar content for both educators and students, but the team settings button.

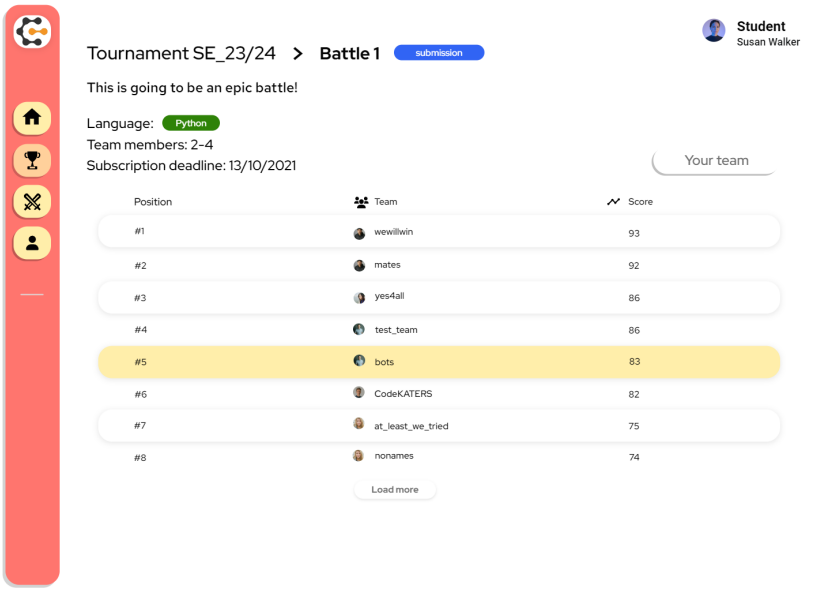


Figure 37: Page used from students when a battle is in submission phase

The team page, accessible only to students of team, will contain all important settings needed to properly setup the GitHub repository and evaluation information about the team's last submission. It will show also some statistics about past submissions to facilitate the tracking of the team's performances. By clicking on "Team members" box, the list of teammates will show up, with related commits count.

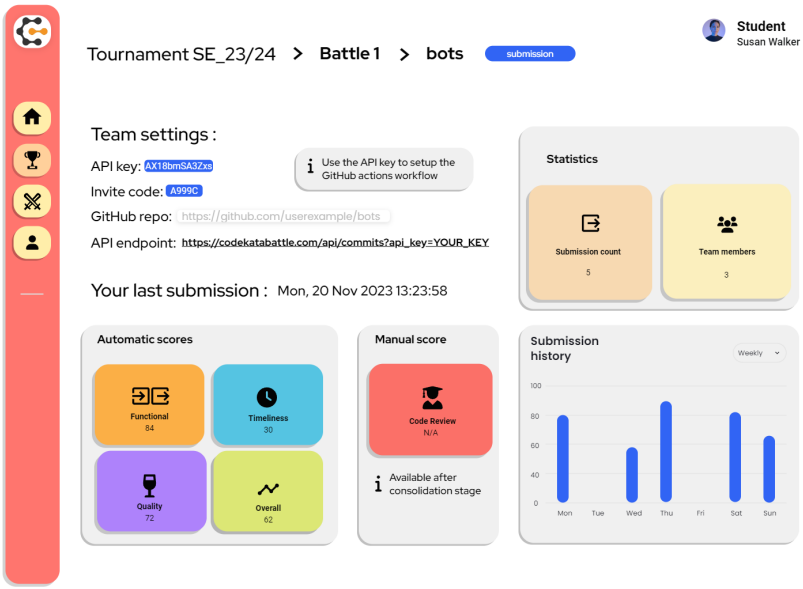


Figure 38: Page used from students to manage the settings of their team and to check submissions information

3.5 Evaluation

The battle page, during the optional consolidation stage, will offer a special team leaderboard, with an overview of both manual and automatic scores. By clicking on a specific team, the educator has access to the evaluation page of the team. The page also contains a button to close the consolidation phase once the educator has evaluated all the teams.
In the evaluation section, the educator is able to access the GitHub repository of the team to check the code of the last submission. Finally, he is provided with a field to input the manual score.

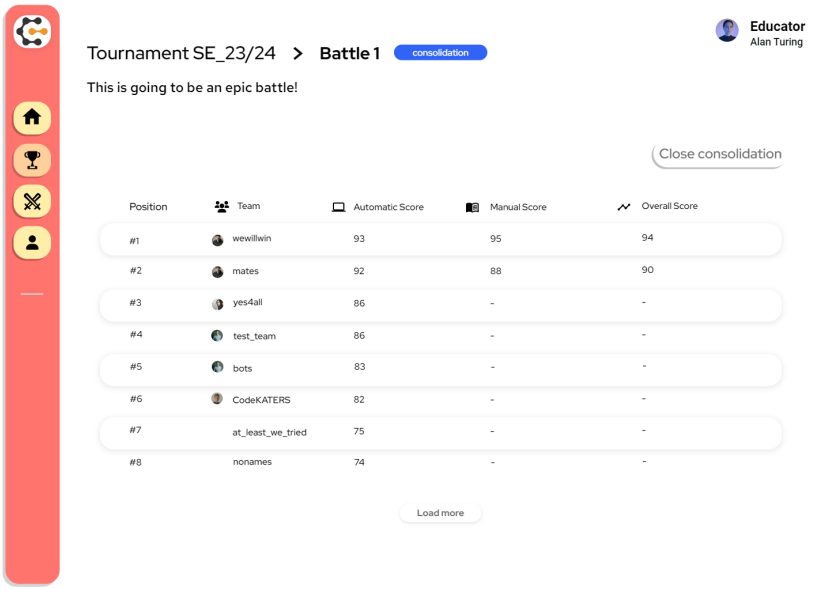


Figure 39: Page used from educators to manage the consolidation stage of a battle

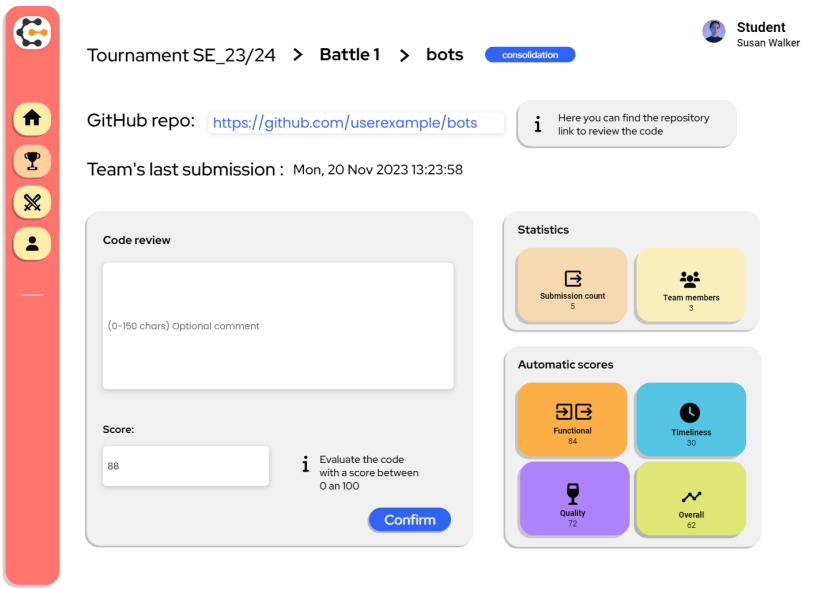


Figure 40: Page used from educators to manually evaluate a submission

4 Requirements traceability

5 Implementation, Integration, Test Plan

The implementation, integration and testing of the system will be perfomed following mainly a bottom-up approach, with some exceptions due to the complex system architecture. Given the microservices architecture, lower-level services will be implemented and tested first, and then gradually integrated with higher-level components. This approach is preferred because it allows to test the system as soon as possible, and to identify and fix bugs early in the development process, reducing the need of implementing complex stubs. External services do not need to be tested, as it is assumed that they are already tested and reliable. The testing will be performed in parallel with the implementation, and will be carried at different levels: unit testing, integration testing and system testing. Regression testing will be performed at each level, to ensure that new changes do not break existing functionalities. Also user acceptance testing is considered, to ensure that the final system meets the requirements of the customer.

5.1 Plan

In the first step, components that do not rely on others will be implemented and tested. These are the ServiceRegistry, EventQueue, NotificationQueue and SubmissionQueue components. Note that, even if the queues are not depending on other components, to be tested it is needed to simulate the services that will consume them. For this reason, stubs of the EventService, NotificationService and SubmissionService will be needed. Given the fact that ServiceRegistry covers a fundamental role in the synchronous communication between services, it will be implemented and tested before the other service components. However, in the following pictures, the connection between ServiceRegistry and other services is not shown, to avoid cluttering the diagrams.

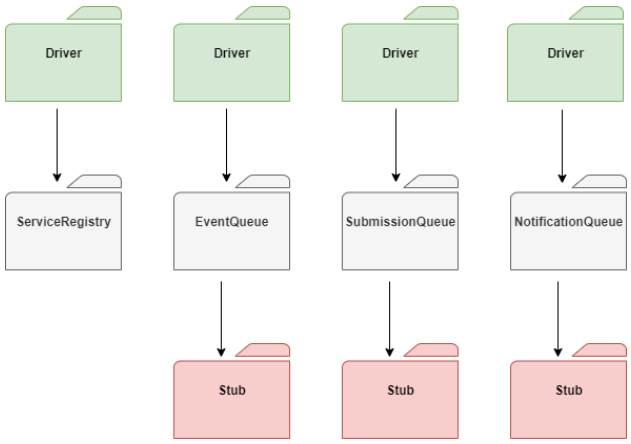


Figure 41: Bottom-up step 1

As second step, components that depends on the previous ones and on external services will be implemented, tested and integrated. These are the NotificationService and the GitHubIntegrationService. The NotificationService consumes the NotificationQueue, so it replaces the corresponding stub. For this reason, the previous driver introduced to test the NotificationQueue can be reused.

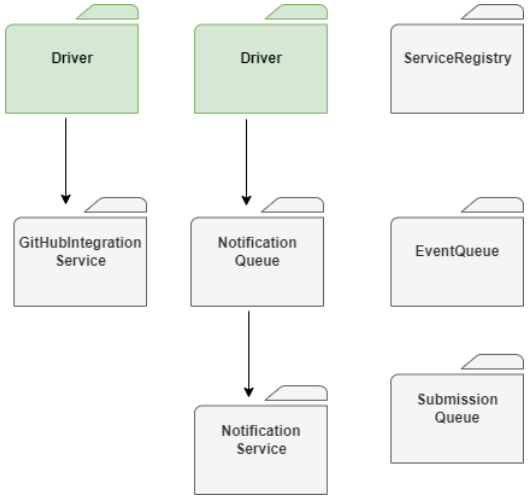


Figure 42: Bottom-up step 2

The third step will introduce in the systems two fundamental services: the TournamentService and the AuthService. Being both composed of two

subcomponents (managers and models), they also should be implemented and tested incrementally: starting the implementation from models, no stubs will be needed to test managers, and then the two subcomponents will be integrated together. However, since the TournamentService relies on the EventService as event consumer, a stub of the EventService will be needed to test the TournamentService.

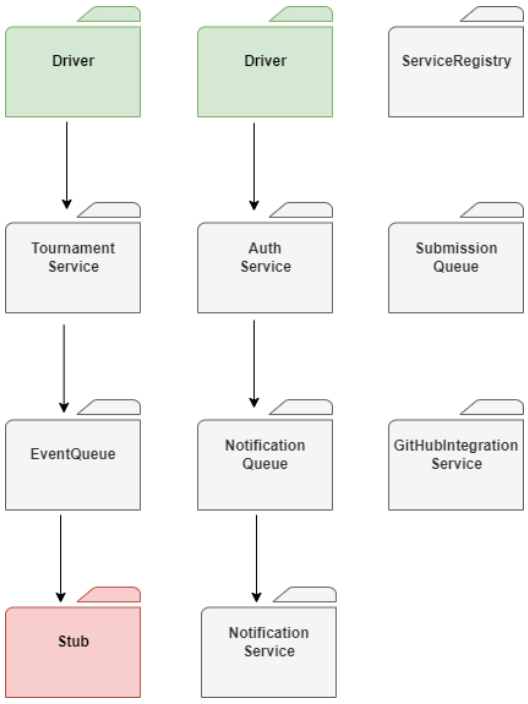


Figure 43: Bottom-up step 3

The fourth step includes the implementation of the EventService and EvaluationService.

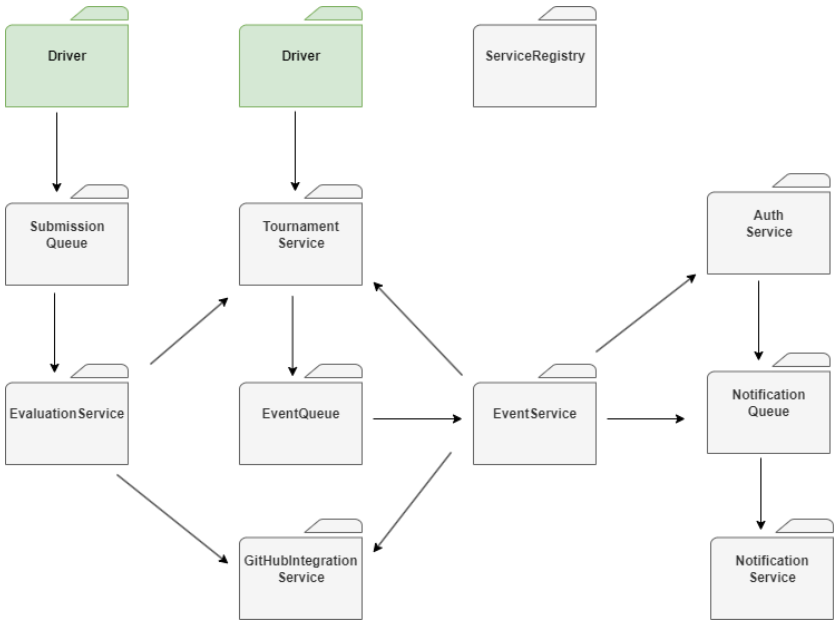


Figure 44: Bottom-up step 4

As fifth step, the APIGateway is implemented and integrated with the rest of the system. To test the APIGateway, a driver is needed. In particular, this driver, in addition to simulating the web calls related to the webapp, will also need to simulate the external calls of the GitHubActions service.

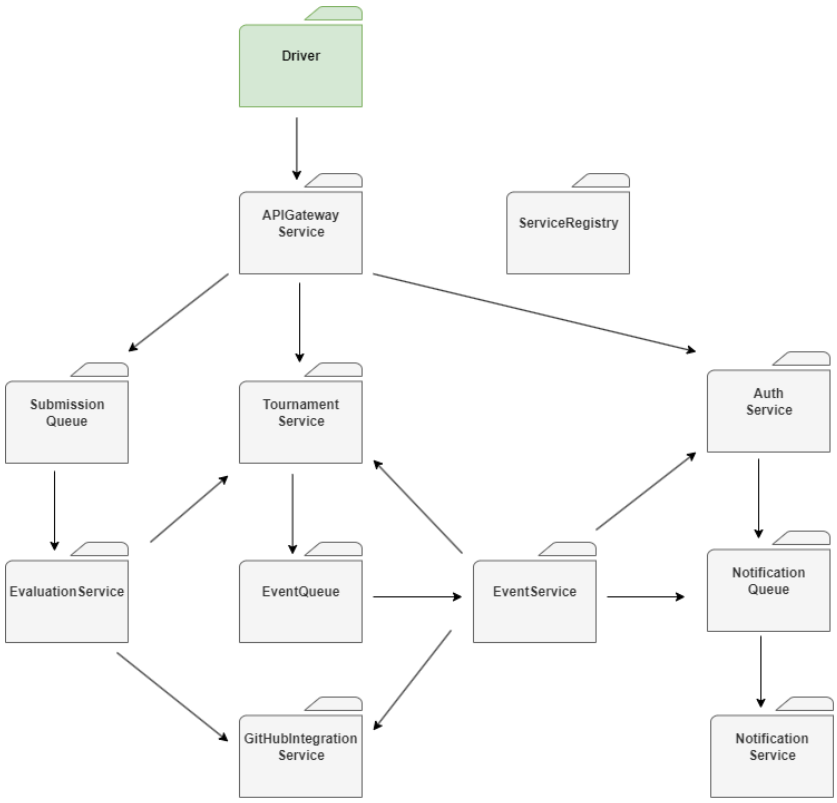


Figure 45: Bottom-up step 5

Finally, the WebApplication is implemented and tested, concluding the implementation and integration of the whole system.

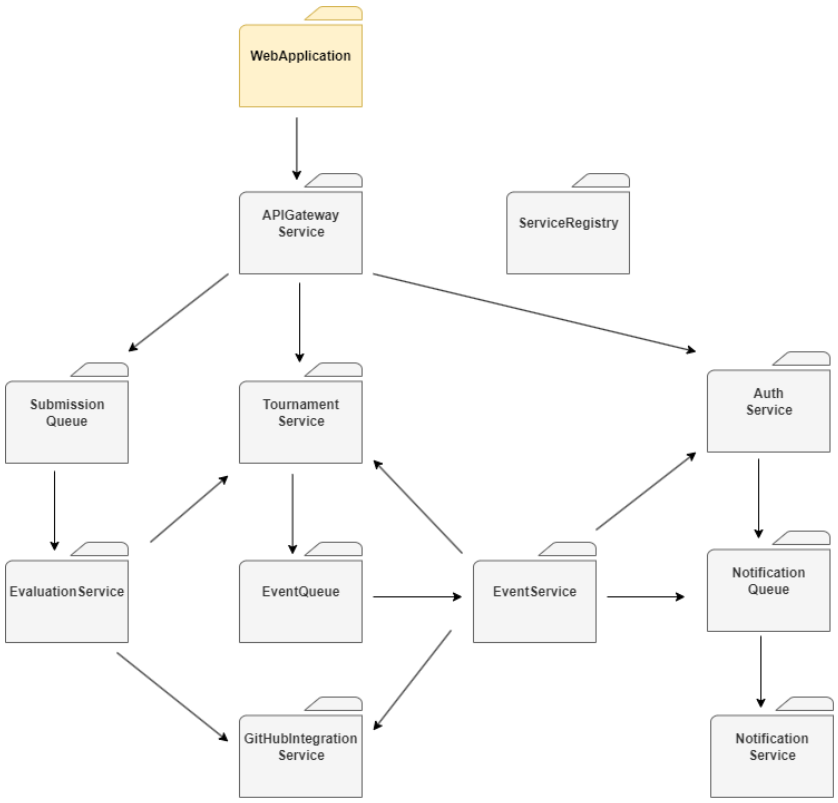


Figure 46: Bottom-up step 6

5.2 E2E Testing

Once the system is fully integrated, Selenium will be adopted as main framework for automating end-to-end testing of the web application, covering multiple browsers to ensure optimal compatibility. Tests will simulate real user workflows that have been identified in the RASD. Each workflow such as signup, creation of a tournament, battle enrollment, etc. will be scripted to replicate user actions, data entry, navigation between pages, and validation of responses. Assertions will be written to verify expected UI updates, error messages, database changes, and integration with external services. Tests will be data-driven to cover different use cases and scenarios.

6 Effort Spent

Name and Surname	Section 1	Section 2	Section 3	Section 4
Tommaso Capacci	10	10	10	10
Gabriele Ginestroni	10	10	10	10