POLITECNICO MILANO 1863

SOFTWARE ENGINEERING II

CKB – CodeKataBattle

Design Document

Version 1.0

Feraboli Alessandro, Filippini Marco, Lucca Simone

January 07, 2024

CONTENTS

1. Introduction

1.1 Purpose

The aim of the CodeKataBattle (CKB) platform is to facilitate an interactive and educational environment for students, enabling them to refine their software development skills through practical coding exercises. The platform also helps students to improve their collaboration skills, by allowing them to form teams and engage in competitive coding battles.

1.2 Scope

To meet scalability and availability requirements, a microservices architecture is recommended. Each service handles a specific part of the application independently. Opting for separate databases for each service eliminates coupling between components, enhancing resilience and scalability. The interfaces for each microservices are RESTful and they communicate with each other only using http. This choice is preferable to asynchronous queues because most communication that needs to happen is for data that is essential to complete the various operation. This also reduces the complexity of the system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Code KATA: A kata, or code kata, is defined as an exercise in programming which helps hone your skills through practice and repetition

1.3.2 Acronyms

- CKB: Code Kata Battle
- RASD: Requirement Analysis and Specification Document
- DD: Design Document
- API: Application Programming Interface

1.3.3 Abbreviations

- [Gn] → the n-th goal
- [WPn] → the n-th world phenomenon
- [SPn] → the n-th shared phenomenon
- [Dn] → the n-th domain assumption
- [Rn] → the n-th requirement
- [UCn] → the n-th use case

1.4 Revision history

- Version 1.0 (01/07/2024)

1.5 Reference documents

- Specification of the DD assignment of the Software Engineering II course, held by professors Matteo Camilli, Elisabetta Di Nitto and Matteo Rossi at Politecnico di Milano, A.Y. 2023/2024.
- Slides of the Software Engineering II course, held by professors Matteo Camilli, Elisabetta Di Nitto and Matteo Rossi at Politecnico di Milano, A.Y. 2023/2024.

1.6 Document structure

Here it is the document structure:

1. **Introduction** → Useful information for the reader.
2. **Architectural Design** → Detailed description of the architecture of the system and its components, including deployment and runtime views.
3. **User Interface Design** → Mockups of the user interfaces.
4. **Requirements Traceability** → Connections between requirements defined in the RASD and the components described in this document
5. **Implementation, Integration and Test Plan** → Description of implementation, integration and testing processes that lead to a correct implementation of the system.
6. **Effort Spent** → Time spent on this document (per person and per section).
7. **References** → References to any document or software used to realize this document.

2. Architectural Design

2.1 High Level View

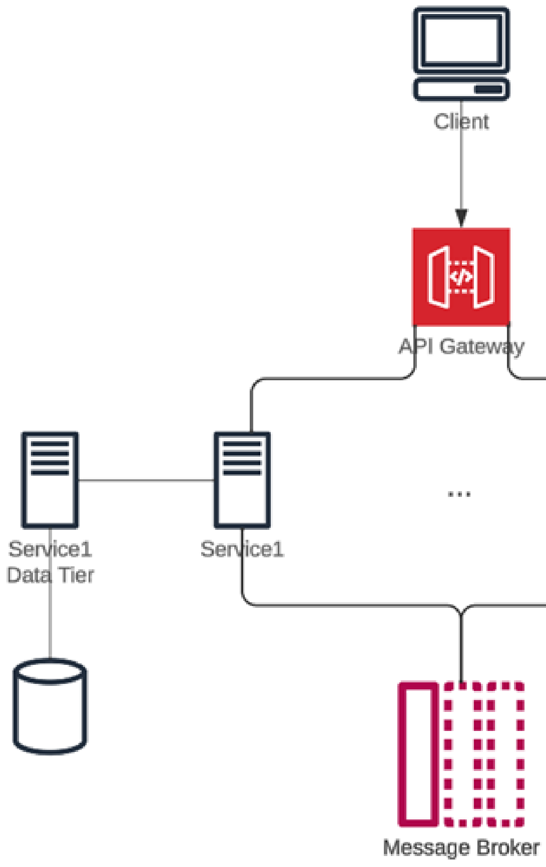


Figure 2.1

2.1.1 Client

The client is a web application running on the browser that directly communicates with the API gateway.

2.1.2 Api Gateway

Purpose:

- Aggregate Microservices: Act as a single-entry point for users to access multiple microservices seamlessly.
- Simplify Integration: Provide a simplified and consistent interface for interacting with the application.
- Enhance Security: Implement authentication and authorization mechanisms to secure access to microservices.

2.1.3 Services

Containing the business logic of the application, each one of them stores the data on separate databases for modularity and scalability purposes.

2.2 Component View

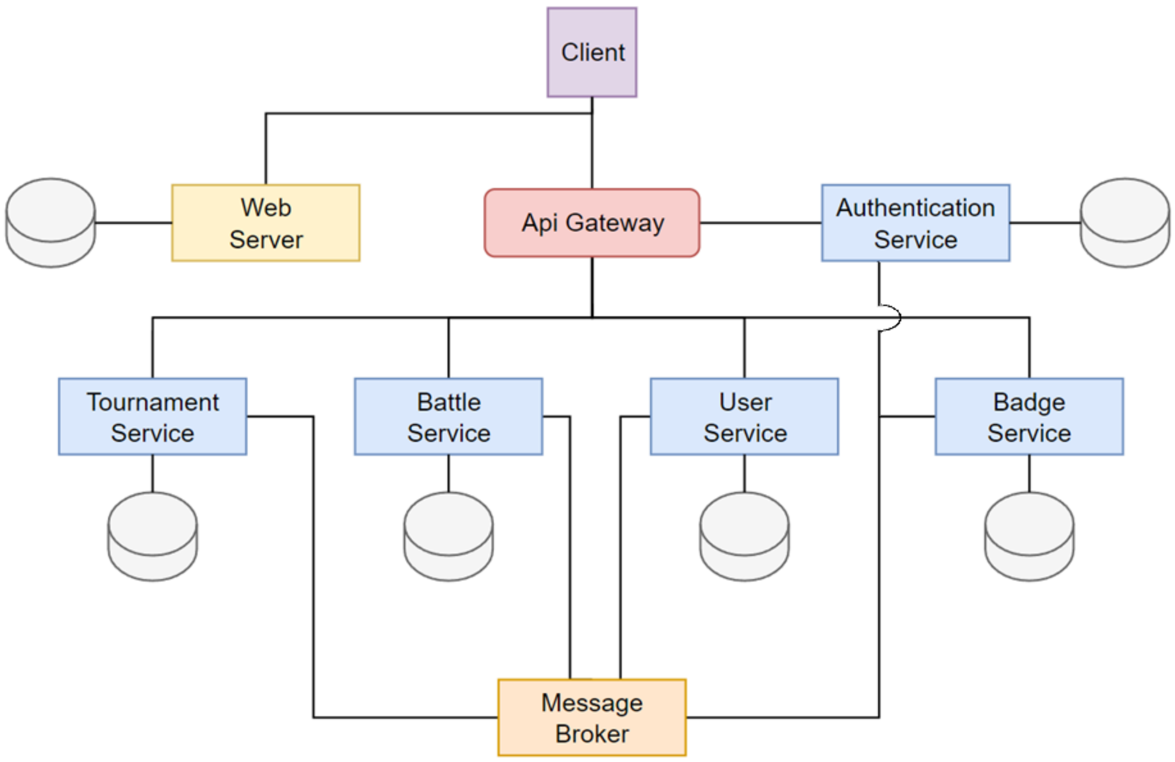


Figure 2.2

In our pursuit of an innovative and robust system architecture, we've embraced a RESTful, microservices-based approach complemented by an API gateway. This choice has brought forth a multitude of benefits across critical domains. Service communication is facilitated through well-defined APIs and protocols, fostering seamless interaction between our independent services. Our strategy for data management is that each service manages its own data, affording autonomy and reducing the risk of data issues.

Regarding scalability, our microservices can be scaled independently, ensuring optimal resource utilization and adaptability to varying workloads.

The distributed nature of our database design enhances fault tolerance. In the event of a database failure within a particular service, it's restricted within that service's scope, minimizing the impact on the entire system. Furthermore, we prioritize the use of database technologies that offer high availability and fault tolerance mechanisms, such as replication, sharding or clustering, depending on specific service requirements.

Testing and deployment phases are very simple: each microservice has its dedicated testing suite, allowing for comprehensive validation of individual functionalities. Deployment is managed adeptly, ensuring updates and changes are rolled out seamlessly without disrupting the entire system.

This architecture empowers us to navigate the complexities of modern software systems while harnessing the advantages of flexibility, scalability, security, and efficiency in service delivery.

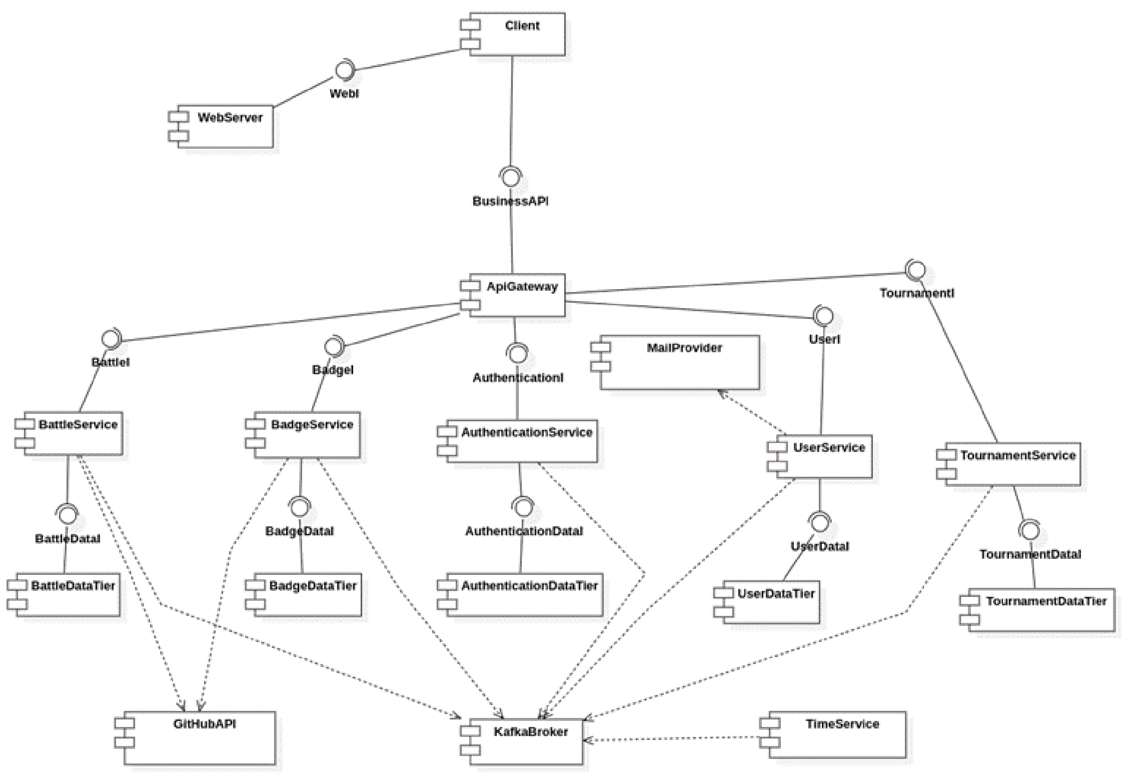


Figure 2.3

The following component diagrams were made to further clarify the structure of each component, they do not represent the software structure but only a logical order.

2.2.1 Tournament Service

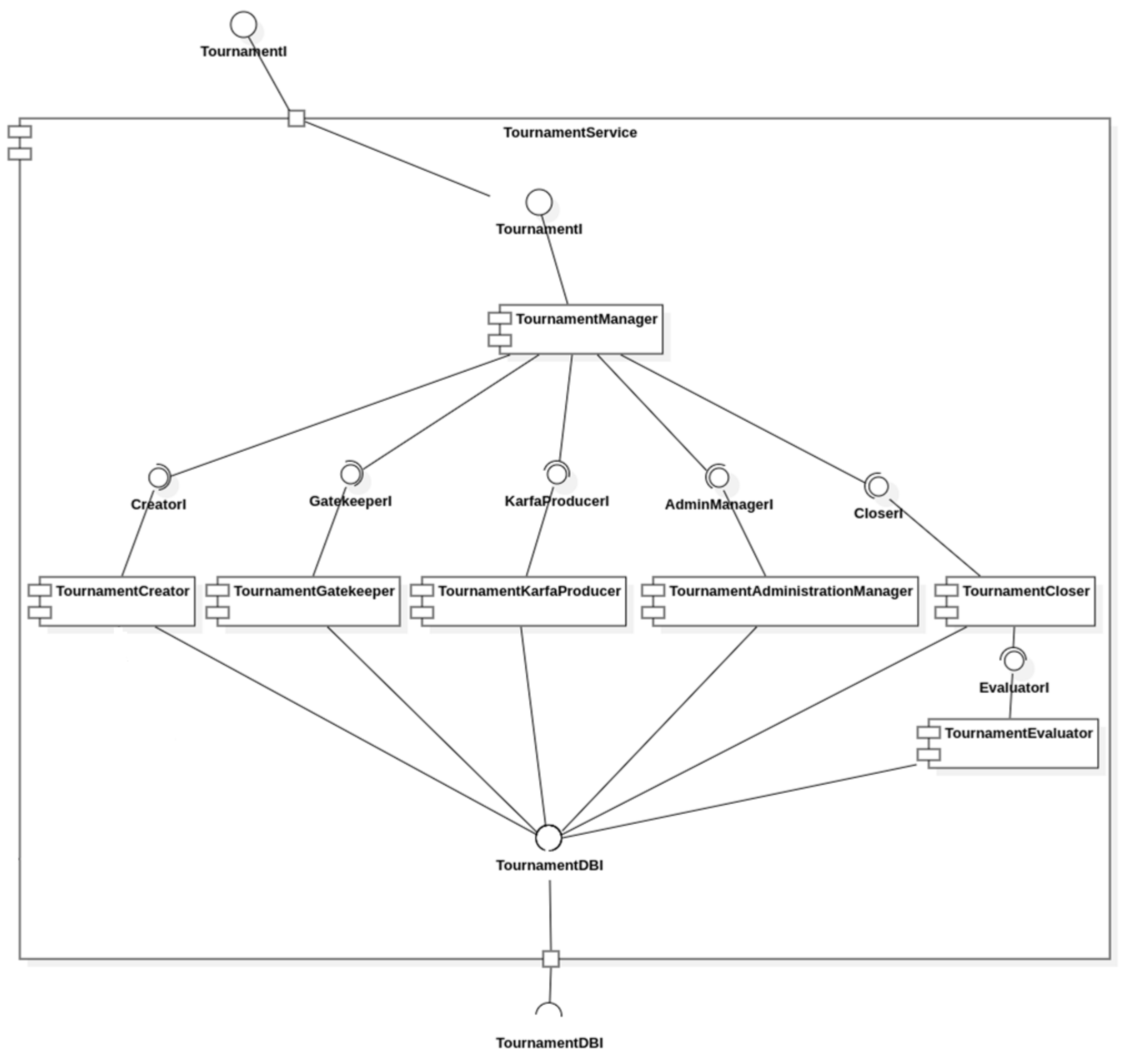


Figure 2.4

Tournament Service responsibilities:

1. Creation of the tournament;
2. Join tournament;
3. Quit tournament;
4. Add educator to the tournament;
5. Close tournament;
6. Calculate ranking and winner of the tournament;
7. Notify users about tournament creation through an external email service.

Components:

1. Tournament Manager
   - Acts as a dispatcher, redirecting the requests to the proper service
   - Checks the format of the request, ensuring that it was formulated correctly
2. Tournament Creator
   - Handles the requests for creating the tournament

3. Tournament Gatekeeper
    - Handles the requests for join and exit from the tournament
4. Tournament Administration Manager:
    - Handles the requests from a tournament leader for adding educators to the tournament
5. Tournament Closer
    - Handles the close tournament requests
6. Tournament Evaluator
    - Triggered internally, handles the ranking of the tournament
7. Tournament Kafka Producer
    - Responsible for publishing to the Kafka Broker new event about tournaments (e.g. tournament creation, closure, student registration,...)

Tournament Data Tier:

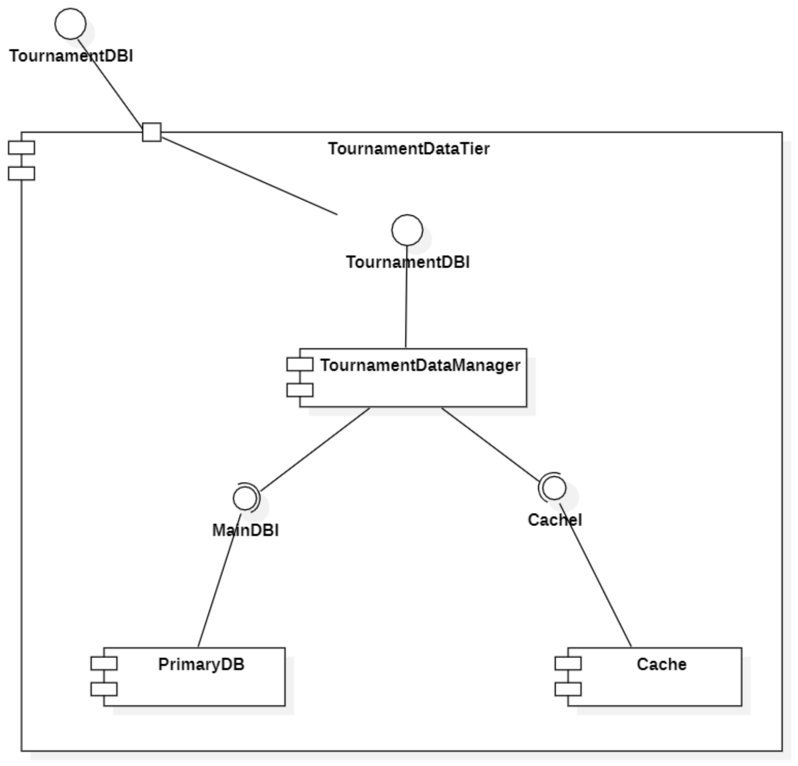


Figure 2.5

Components:

1. TournamentDataManager:
   Handles the accesses to the cache and the primary database to maximize performance
2. PrimaryDB:
   Relational database (or cluster) able to store large amounts of data
3. Cache:
   Fast and small database to access frequently accessed data

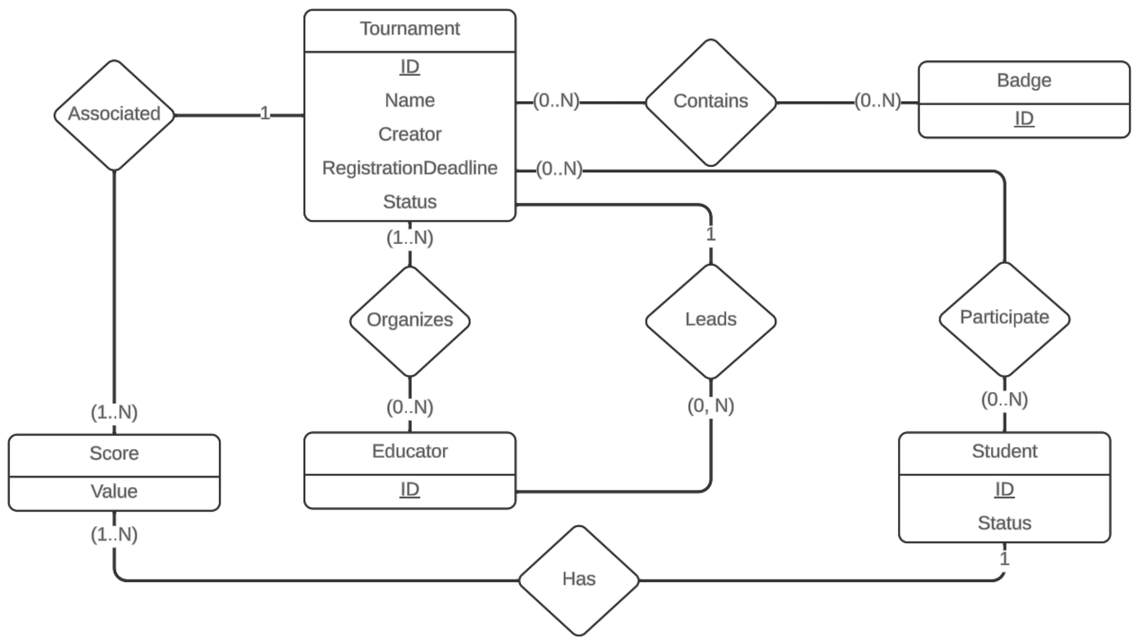
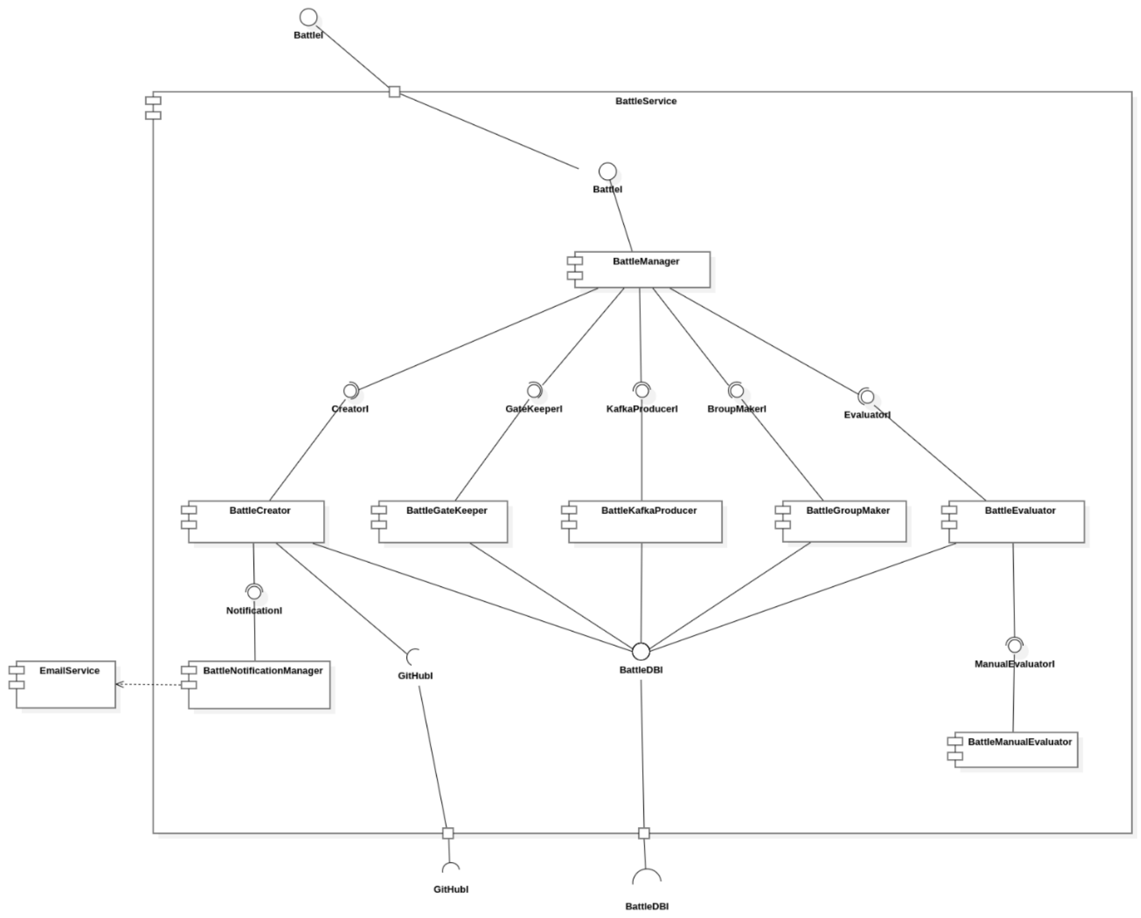


Figure 2.6

2.2.2 Battle Service



.

Figure 2.7

Battle service responsibilities:

1. Creation of battles;
2. Join battles;
3. Quit battles;
4. Form groups;
5. Pull the code of each group from GitHub;
6. Automatically evaluate students;
7. Manually evaluate students;
8. Notify students about battle creation through an external email service;
9. Create a GitHub repository for each battle through an external GitHub service.

Components:

1. Battle Manager
   - Acts as a dispatcher, redirecting the requests to the proper service
   - Checks the format of the request, ensuring that it was formulated correctly
2. Battle Creator
   - Handles the requests for creating a battle
3. Battle Gate Keeper
   - Handles the requests for join and exit from a battle
4. Battle Group Maker
   - Manages the groups creation process w.r.t. limits on group members imposed in the battle's creation phase
5. Battle Evaluator
   - Performs the automatic evaluation of code produced by each group of students. This is done every time they push the code on the main branch of their repository on GitHub
6. Battle Manual Evaluator
   - Allows the creator of a battle to manually evaluate each group of students (only when the submission deadline has expired)
7. Battle Kafka Producer
   - Responsible for publishing to the Kafka Broker new event about tournaments (e.g. tournament creation, closure, student registration,...)

Battle Data Tier

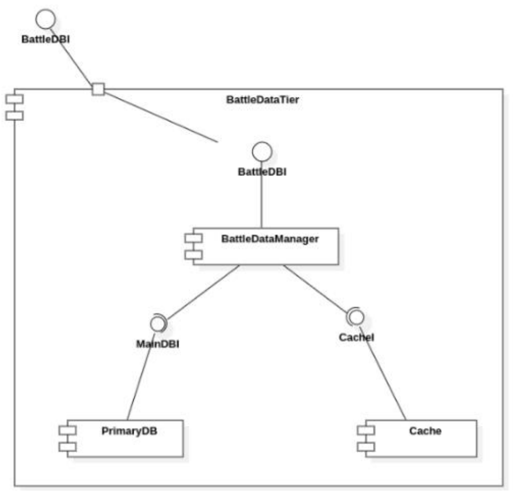


Figure 2.8

Components:

1. BattleDataManager:
    - Handles the accesses to the cache and the primary database to maximize performance
2. PrimaryDB:
    - Relational database (or cluster) able to store large amounts of data
3. Cache:
    - Fast and small database to access frequently accessed data

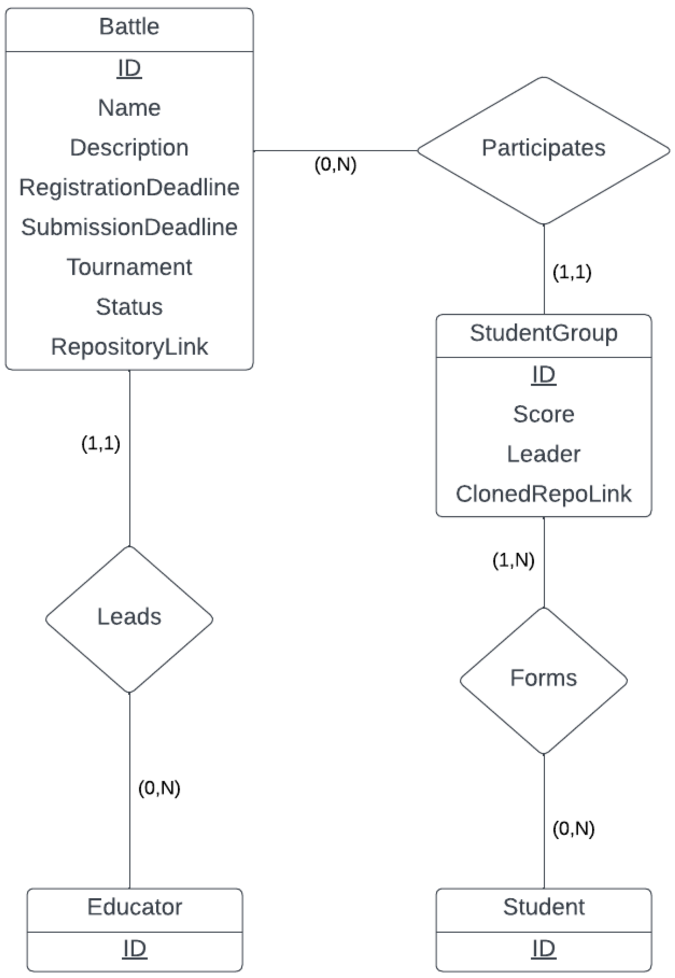


Figure 2.9

2.2.3 User Service

The “User Service” plays an important role in managing user personal information and contexts involving tournaments, battles, and badges. This service is responsible for maintaining and managing a variety of users’ personal data, including their personal details, email addresses, and passwords. Moreover, a significant aspect of the User Service is its capability to track users’ participation in various tournaments, as well as their performance in each battle within these tournaments. The user service can also keep track of the badge earned by each use. This is achieved by establishing relationships between the unique IDs of each battle and tournament and badges and the IDs of the users. These relationships enable the system to map the activities of each participant accurately and efficiently, ensuring effective data management.

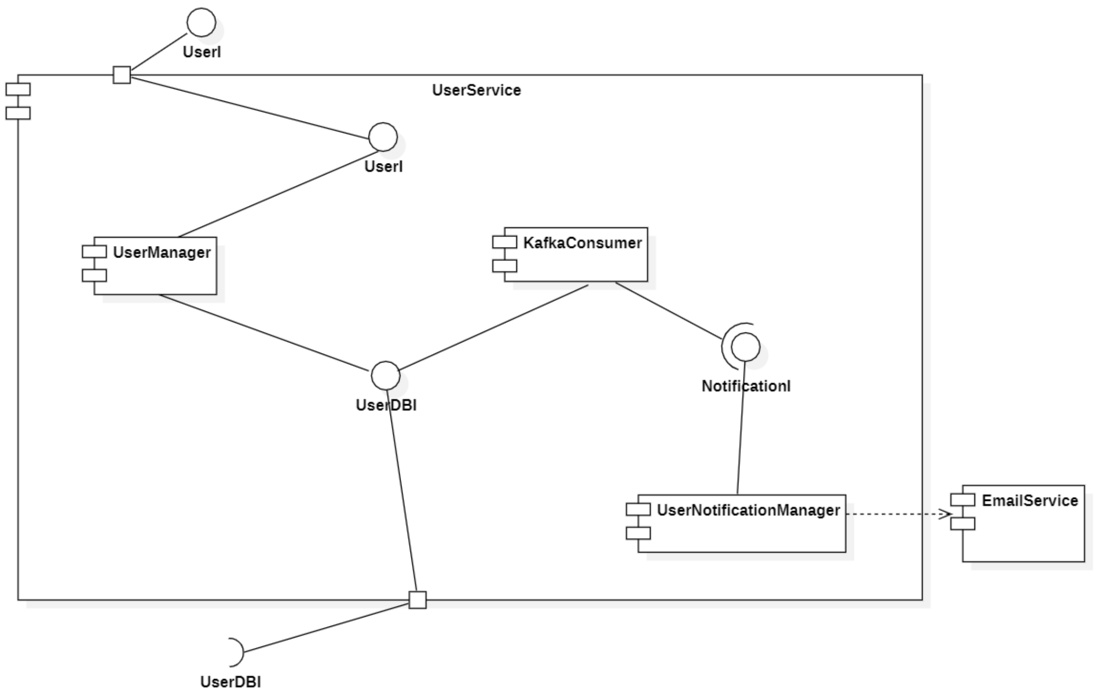


Figure 2.10

User service responsibilities:

1. Retrieve user personal data;
2. Retrieve tournaments which the user participates to;
3. Retrieve battles which the user participates to;
4. Retrieve badges earned by user;
5. Receive new data about tournaments, battles, and badges to upload next in the database;
6. Send email to the user through an email server.

Components:

1. User Manager
   - Checks the format of the request, ensuring that it was formulated correctly
   - Forward the request to the User data tier to retrieve data (specified in the request) from the database

2. Kafka Consumer
    - Receive the data published by the tournaments, battles and badges to forward this data to the database to upload it.
3. Tournament Notification Manager:
    - Responsible for sending notification

User Data Tier

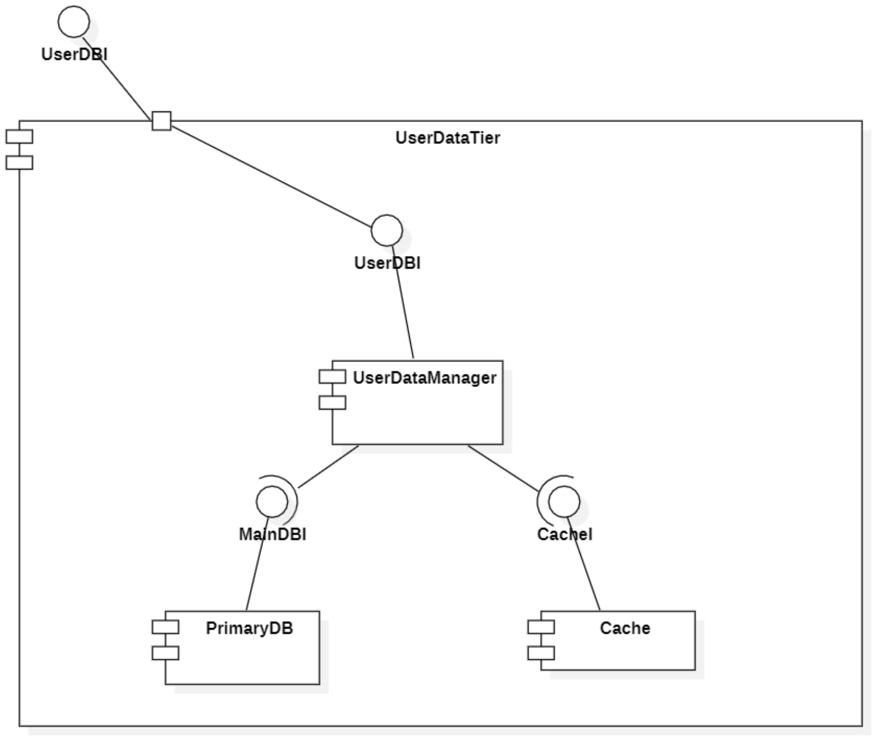


Figure 2.11

Components:

1. UserDataManager:
    - Handles the accesses to the cache and the primary database to maximize performance
2. PrimaryDB:
    - Relational database (or cluster) able to store large amounts of data
3. Cache:
    - Fast and small database to access frequently accessed data

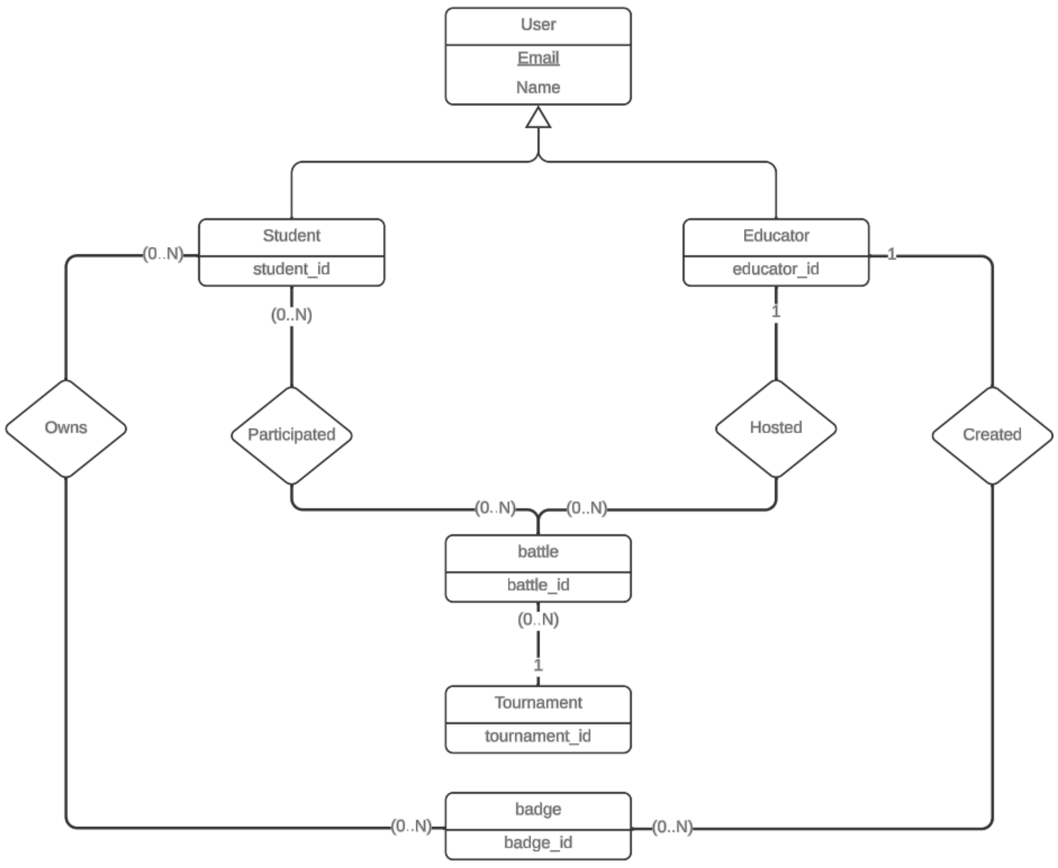


Figure 2.12

2.2.4 Badge Service

The badge service handles the creation, deletion, and evaluation of the badges. The main problem with this module is to provide an easy and safe language for the educator that allows him/her to create a large variety of predicates for the badges.

Educators must be able to create variables and predicates to express the badge predicate.

Language basic elements:
Types: int, bool

Functions: +, -, /, *, max, min, count

Comparator operators: <, >, >=, <=, ==, !=

Macros:
1. Repository Information
    1. pushed_at: Timestamp of the last push to the repository.
    2. size: Size of the repository in kilobytes.
    3. stargazers_count: Number of stars.
    4. watchers_count: Number of watchers.
    5. forks_count: Number of forks.
    6. open_issues_count: Number of open issues.
    7. default_branch: Default branch name.
    8. SUBSCRIBERS_COUNT: NUMBER OF SUBSCRIBERS.

2. User Information
    1. gravatar_id: Gravatar ID.
    2. url: URL of the user's GitHub profile.
    3. html_url: HTML URL of the user's GitHub profile.
    4. followers_url: URL of the user's followers.
    5. following_url: URL of the user's following list.
    6. gists_url: URL of the user's gists.
    7. repos_url: URL of the user's repositories.
    8. events_url: URL of the user's events.
    9. received_events_url: URL of the events received by the user.
    10. type: Type of user (User or Organization).
    11. site_admin: Boolean indicating if the user is a GitHub admin.

3. Issue Tracking
    1. id: Unique identifier for the issue.
    2. number: Issue number.
    3. title: Title of the issue.
    4. user: User who created the issue.
    5. labels: Labels attached to the issue.
    6. state: State of the issue (open, closed).
    7. locked: Boolean indicating if the issue is locked.
    8. assignee: User assigned to the issue.
    9. milestone: Milestone associated with the issue.
    10. comments: Number of comments on the issue.
    11. created_at: Timestamp when the issue was created.
    12. updated_at: Timestamp when the issue was last updated.
    13. closed_at: Timestamp when the issue was closed.
    14. author_association: Author's association with the repository.
    15. body: Body of the issue.

4. Branches and Commits
    1. node_id: Node ID of the commit.
    2. commit: Commit details including message, author, and committer.
    3. author: Author details of the commit.
    4. committer: Committer details of the commit.
    5. parents: Array of parent SHAs.
    6. branch_name: Name of the branch.
    7. protected: Boolean indicating if the branch is protected.
    4. Content and Files
    8. type: Type of the item (file, dir, symlink, submodule).
    9. size: Size of the item.
    10. name: Name of the item.
    11. path: Path of the item.
    12. content: Content of the file.
    13. sha: SHA of the item.
    14. url: URL to access the item.
    15. git_url: Git URL to access the item.
    16. html_url: HTML URL to access the item.
    17. download_url: Download URL of the item.

5. Webhooks and Events

5. type: Type of the event.
6. public: Boolean indicating if the event is public.
7. repo: Repository information where the event occurred.
8. actor: User who triggered the event.
9. org: Organization information if the event is within an organization.
10. created_at: Timestamp when the event occurred.

The macros are used as a simplification on top of the github api, each macro is hiding a procedure that will retreive a certain value from an api call.

Language syntax:
The educator can create int, boolean and string constants as follow:

```
int name1 = <int expression>;
bool name2 = <boolean expression>;
string name3 = <string expression>;
```

It is not possible to create an uninitialized variable.

The code must end with one or more of the following lines:

```
winif <boolean expression>;
```

This line defines the predicate that evaluates if a student won the badge. The educator will write multiple winif lines in case the badge has more than one predicate to satisfy.

Rule memorization:
The rules are tested and saved in the database as form of a string.

Evaluation strategy:
Each time a badge needs to be assigned to students the code is taken from the database. The macros are evaluated through java native interface, this allows GitHub API calls and seamless interservice communication.

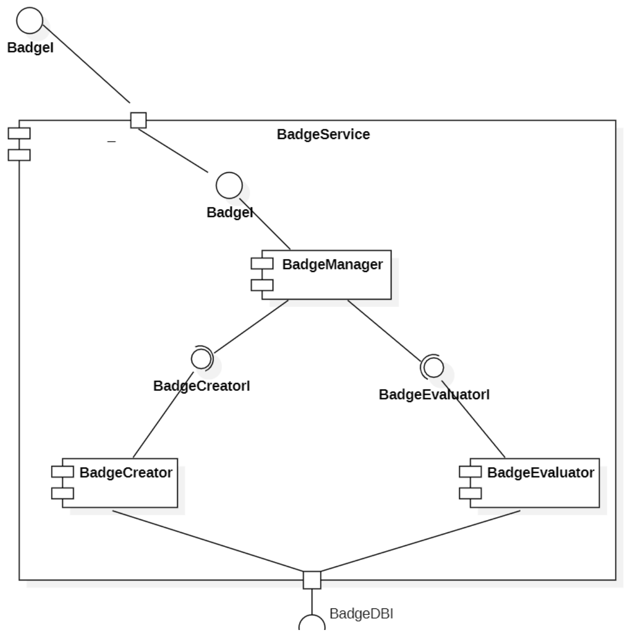


Figure 2.13

Badge data tier

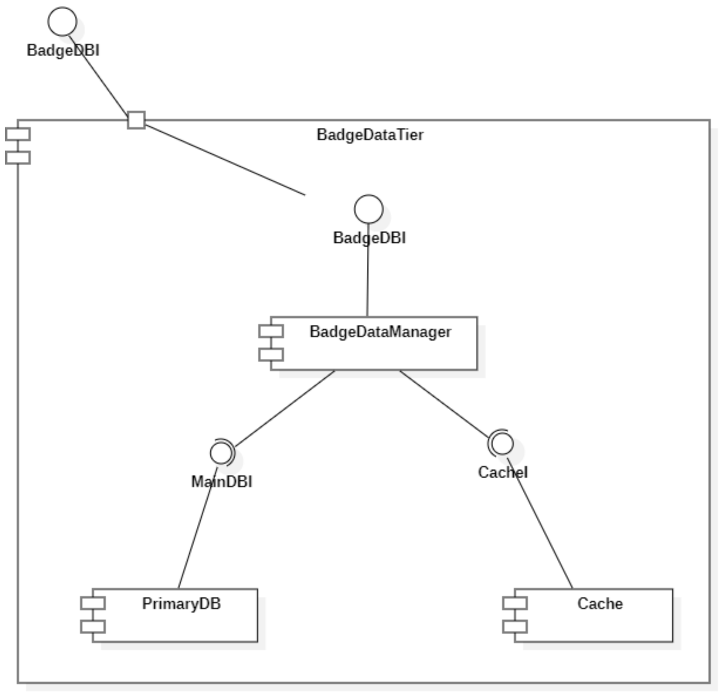


Figure 2.14

The components presented in this diagram are analogous to the one presented in previous services.

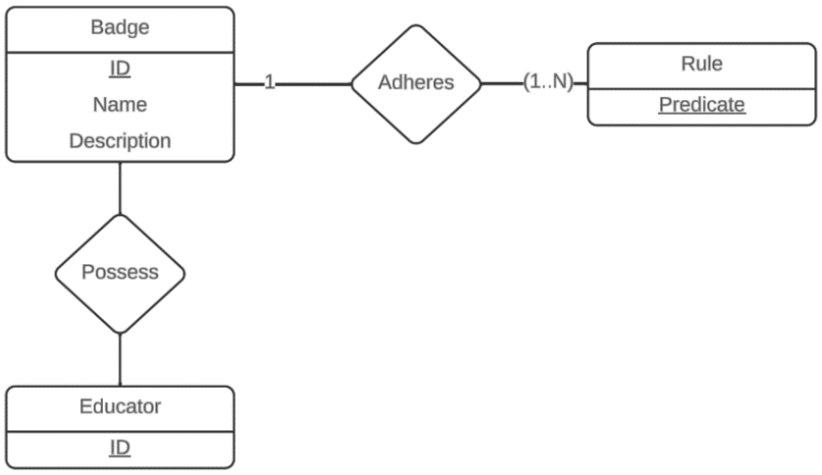


Figure 2.15

The decision of adding the educator table in the badge database was taken to speed up the deletion of a badge since there is the need to check if the user who deleted the badge was the one who possess it.

2.2.5 Authentication Service

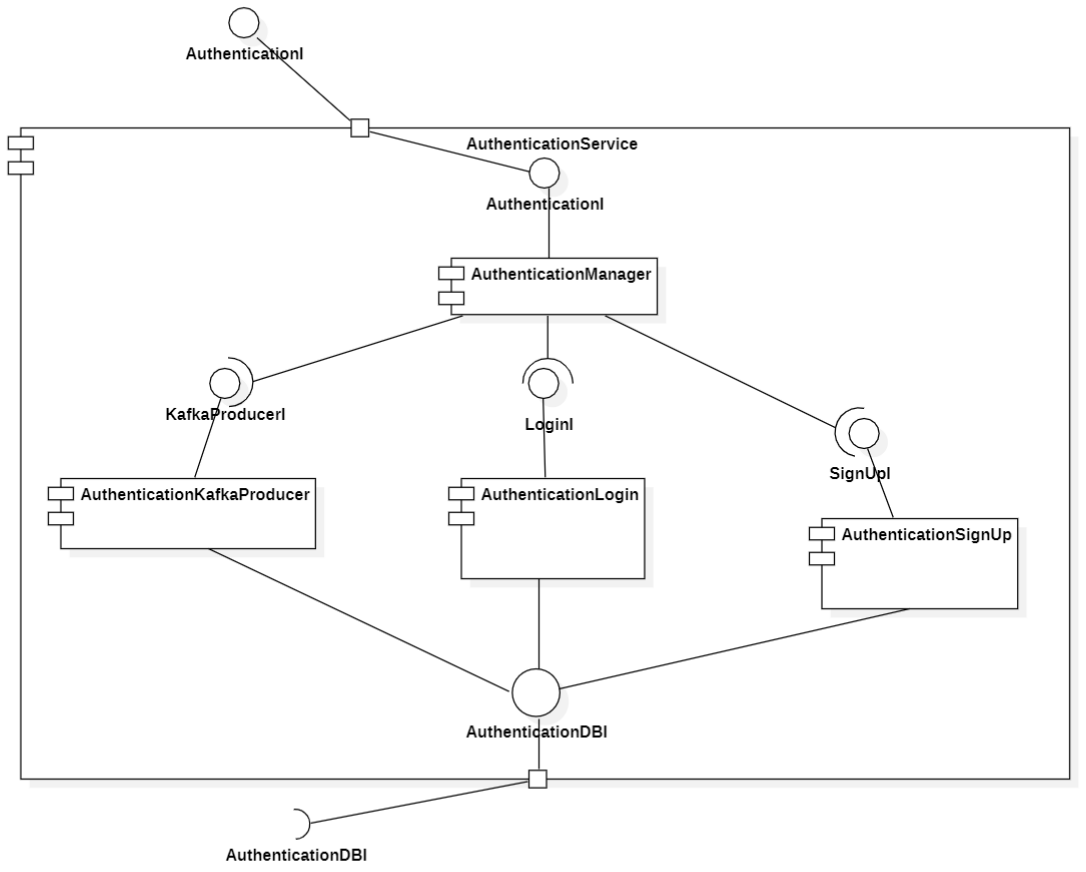


Figure 2.16

Authentication Service responsibilities:

1. User registration(Sign Up);
2. User Login.

Components:

1. Authentication Manager
    - Acts as a dispatcher, redirecting the request to the proper service (Authentication login or signup)
    - Checks the format of the request, ensuring that it was formulated correctly
2. Authentication Login
    - It checks user credentials ensuring that the person trying to log in is indeed who they claim to be
3. Authentication Sign Up
    - it’s very simple, it only ask for essential details like username, email and password
4. Authentication Manager
    - It has the task to send the confirmation of the registration through the mail provider to the user

Authentication data tier

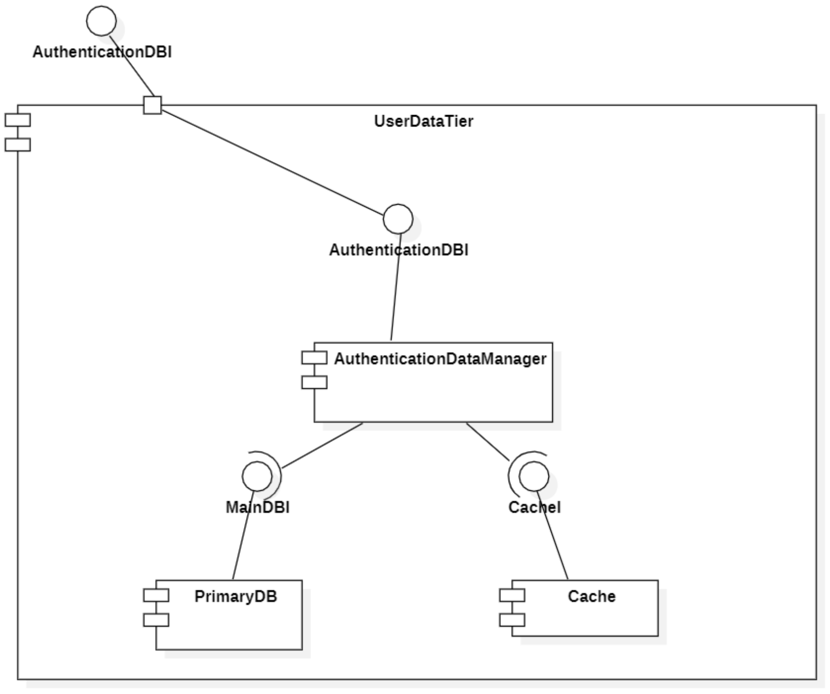


Figure 2.17

Components:

1. UserDataManager:
    - Handles the accesses to the cache and the primary database to maximize performance
2. PrimaryDB:
    - Relational database (or cluster) able to store large amounts of data
3. Cache:
    - Fast and small database to access frequently accessed data

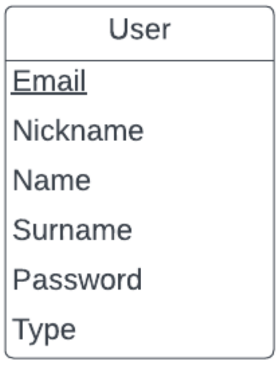
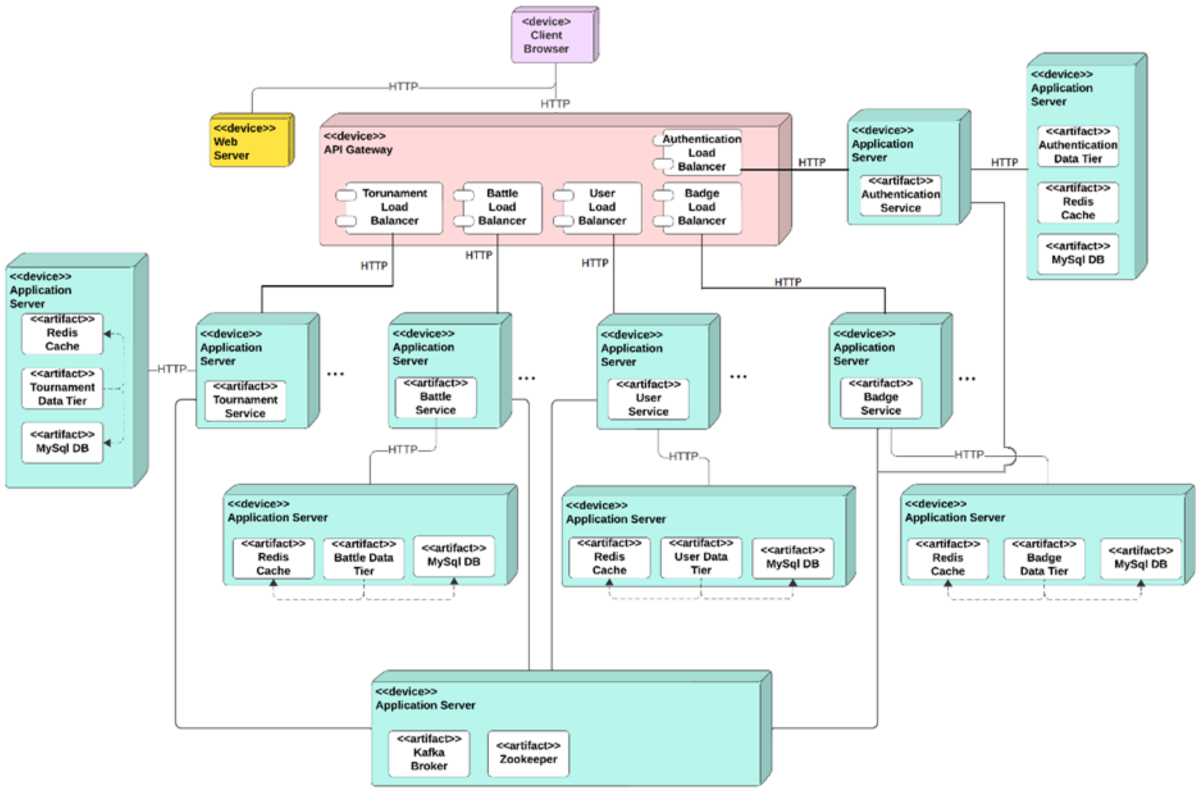


Figure 2.18

The "Email" is the key which uniquely identifies the user. The attribute "Type" identifies the user between Student and Educator. The "Nickname" identifies the name with which the user will appear in the system. The others are fields that identify personal data for each user

2.3 Deployment view



<device> Client Browser
HTTP
HTTP
<<device>> Web Server
<<device>> API Gateway
Authentication Load Balancer
Torunament Load Balancer
Battle Load Balancer
User Load Balancer
Badge Load Balancer
HTTP
<<device>> Application Server
<<artifact>> Authentication Service
HTTP
<<device>> Application Server
<<artifact>> Authentication Data Tier
<<artifact>> Redis Cache
<<artifact>> MySql DB
HTTP
HTTP
HTTP
HTTP
<<device>> Application Server
<<artifact>> Redis Cache
<<artifact>> Tournament Data Tier
<<artifact>> MySql DB
HTTP
<<device>> Application Server
<<artifact>> Tournament Service
<<device>> Application Server
<<artifact>> Battle Service
<<device>> Application Server
<<artifact>> User Service
<<device>> Application Server
<<artifact>> Badge Service
HTTP
HTTP
HTTP
<<device>> Application Server
<<artifact>> Redis Cache
<<artifact>> Battle Data Tier
<<artifact>> MySql DB
<<device>> Application Server
<<artifact>> Redis Cache
<<artifact>> User Data Tier
<<artifact>> MySql DB
<<device>> Application Server
<<artifact>> Redis Cache
<<artifact>> Badge Data Tier
<<artifact>> MySql DB
<<device>> Application Server
<<artifact>> Kafka Broker
<<artifact>> Zookeeper

Figure 2.19

System Deployment Overview
The deployment diagram provides a detailed representation of the physical deployment of software components (artifacts) on hardware or virtualized systems (nodes), illustrating the infrastructure setup for the CodeKataBattle platform.

Client Interaction:
At the highest level, we have the **Client**, a web browser through which users interact with the CKB platform. The client communicates with the backend systems via HTTP protocol, initiating interactions that traverse through the system's Web API Gateway.

API Gateway and Load Balancers:
Directly beneath the client interaction point is the **Web API Gateway**, which serves as the system's primary ingress point, funneling incoming HTTP requests to appropriate backend services. This gateway connects to a series of load balancers, each dedicated to a specific cluster of services:

- Tournament Load Balancer
- Battle Load Balancer
- User Load Balancer
- Badge Load Balancer
- Authentication Load Balancer

These load balancers are instrumental in distributing the load to ensure efficient processing and high availability.

Microservices and Data Tiers:
The core of the CKB's backend is composed of a set of microservices, each hosted on dedicated application servers. These microservices include:

- **Tournament Service**: Manages tournament-related operations, interfacing with a data tier that includes a Redis Cache for rapid data access and a MySQL database for persistent storage.
- **Battle Service**: Similar to the Tournament Service, it has its own Redis Cache and MySQL DB to manage battle-centric functionalities.
- **User Service**: Handles user-specific data and interactions, backed by a Redis Cache and MySQL DB for storage and caching needs.
- **Badge Service**: Oversees the allocation and management of gamification badges, supported by a Redis Cache and a MySQL DB.
- **Authentication Service**: Handles the authentication functionalities ensuring that users are authorized to do specific operations.

Each one of these services can be deployed on multiple machines, since the communication is restful, the processes are easily replicable. This is represented in the diagram by ellipses (...).

Messaging and Coordination:
At the lower end of the diagram, there is an application server dedicated to **Kafka** and **Zookeeper**. Kafka acts as a distributed streaming platform for handling real-time data feeds, while Zookeeper is used for Kafka's configuration management and coordination. To improve availability, it is best practise to deploy kafka nodes on multiple machines.

Artifacts:
Each service within the deployment is associated with artifacts that represent deployable units of the system, such as executables, libraries, or configuration files necessary for the operation of the services.

2.4 Runtime view

In visualizing the runtime view of our proposed architecture, the model encapsulates a dynamic ecosystem where multiple microservices collaborate within the overarching framework. At the core lies the API gateway, serving as the gateway to our system. It orchestrates the incoming requests from different clients and efficiently routes them to their respective destinations, which include the authentication service, tournament service, battle service, badge service, user service, and the web server.

As a client initiates an interaction, the API gateway acts as the central dispatcher, directing requests based on defined endpoints and service-specific APIs. The authentication service stands as the guardian, ensuring secure access and authentication for users engaging with our system. Meanwhile, the tournament service manages the intricacies of tournaments, handling the creation, modification, and retrieval of tournament-related data. The battle service governs the dynamics of in-system battles or challenges, while the badge service oversees the achievement framework, tracking user accomplishments and progress.

Integral to this runtime view is the user service, which serves as the core of user-related operations, managing user profiles, registration, and crucial user data. Finally, the web server stands as the interface between our system and the users, presenting a cohesive and interactive front-end experience while liaising with the API gateway to seamlessly retrieve and display information from the different array of services.

This runtime model symbolizes a dynamic and interconnected network of services, designed to cater to the multifaceted needs of our users. It embodies the essence of modularity, scalability, and efficiency.

2.4.1 Login

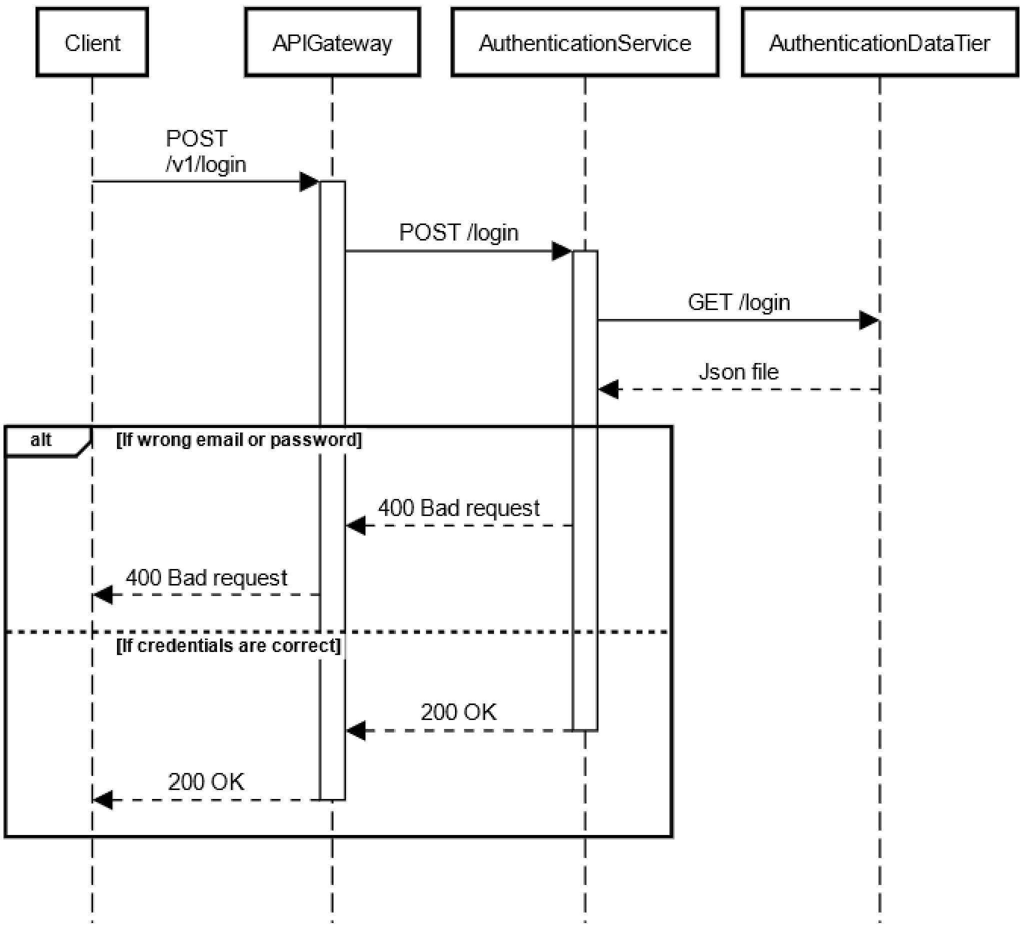


Figure 2.20

2.4.2 Signup

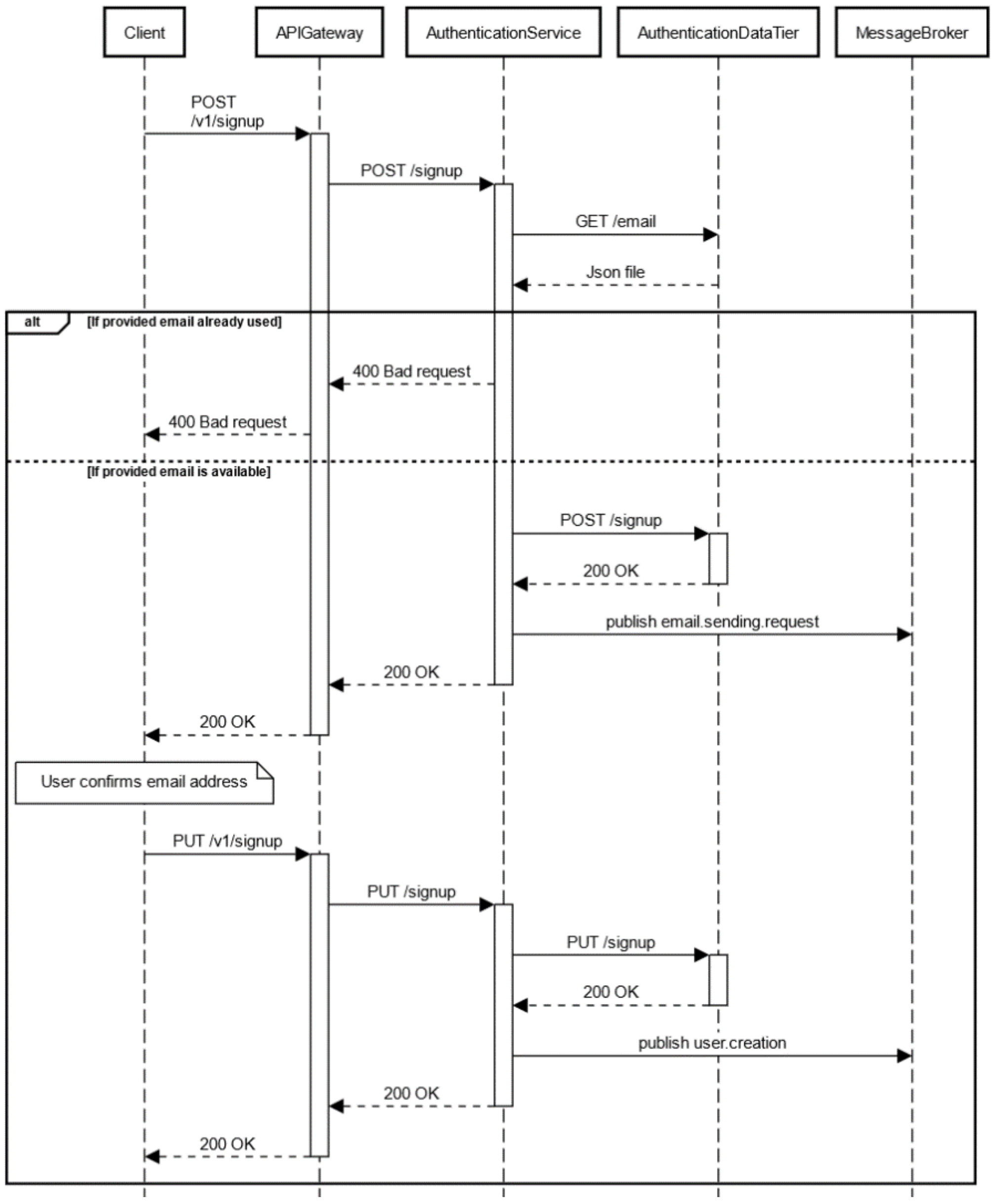


Figure 2.21

After ensuring the user has not already signed up, the system handles the registration request by sending a confirmation email. Until the email is not confirmed, the system does not store any password (so the POST /signup request generates a new entry in the authentication data base with email, first name, last name, nickname and account type). Once the system has received the confirmation, it stores also the new user password through the request PUT /signup.

2.4.3 View user profile

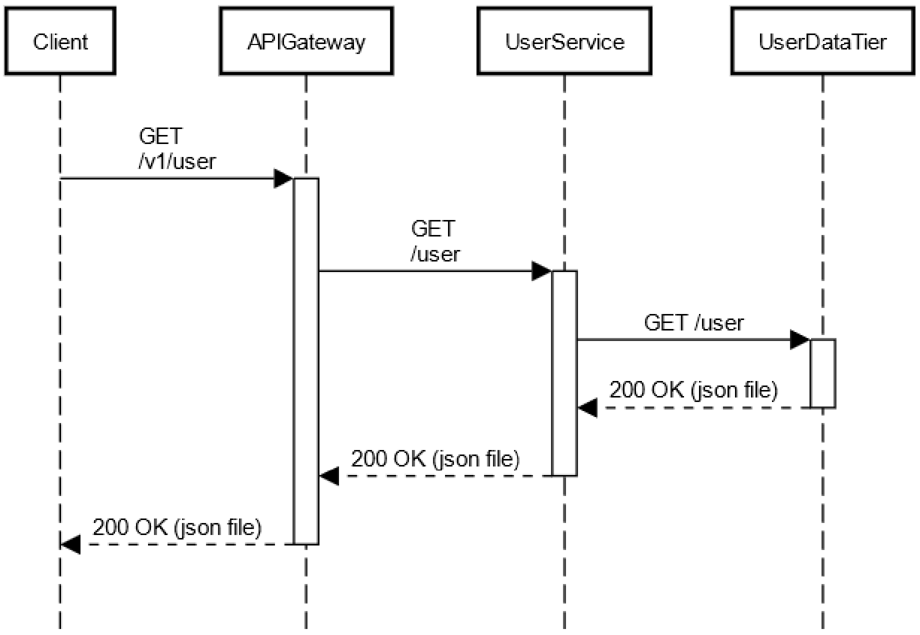


Figure 2.22

2.4.4 Create Tournament

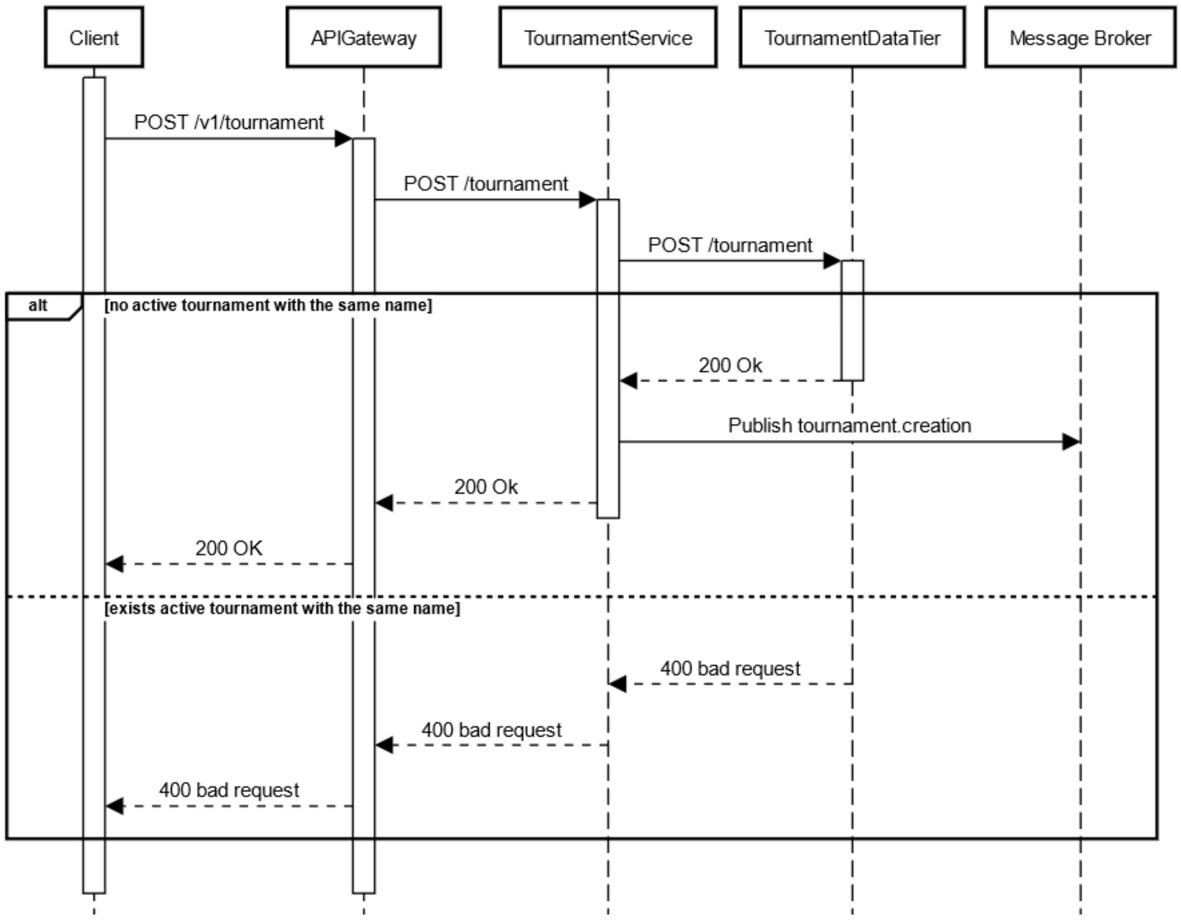


Figure 2.23

In this case, the “create tournament” call is not inherently idempotent. The case of multiple requests by the same user is handled by the data tier using a trigger that checks if there are no active tournaments with the same name.

There is also the need to start the tournament when the registration deadline expires. The application should maintain a state to understand when a specific deadline expires, this is not possible in stateless business logic servers. To solve this problem, a new time server was added. It will take the registration deadline and fire an event when it expires.

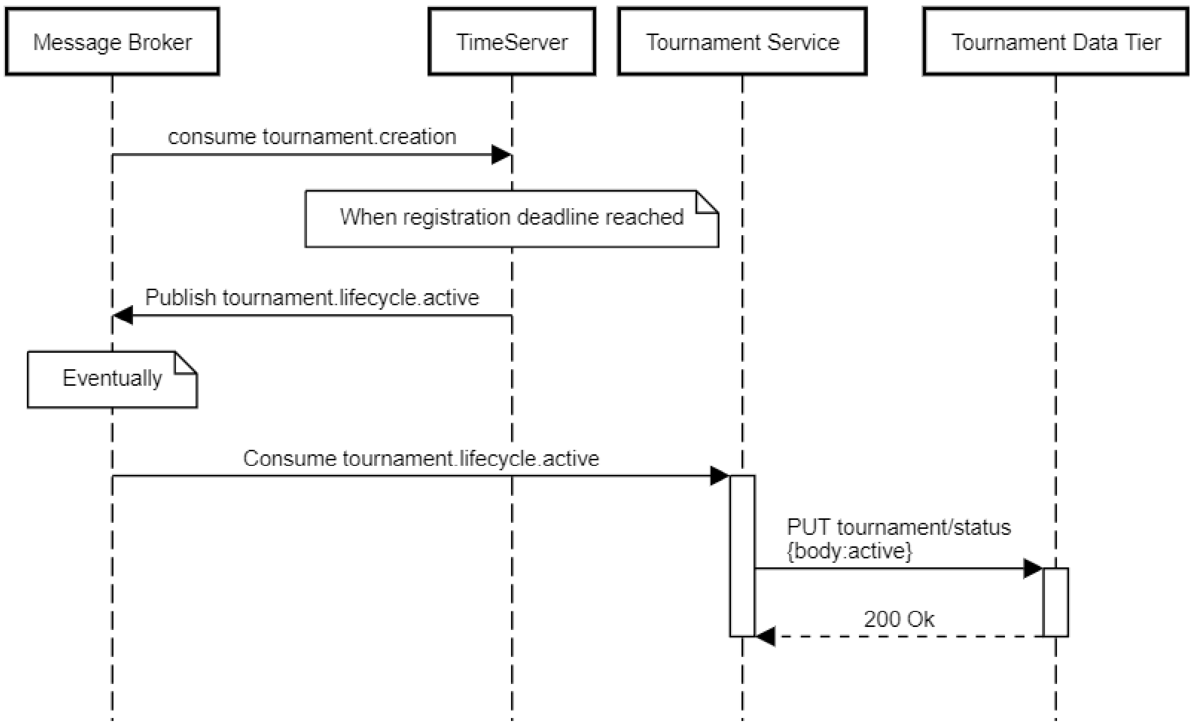

Message Broker
TimeServer
Tournament Service
Tournament Data Tier
consume tournament.creation
When registration deadline reached
Publish tournament.lifecycle.active
Eventually
Consume tournament.lifecycle.active
PUT tournament/status {body:active}
200 Ok

Figure 2.24

Simultaneously, the user service will consume the message on the tournament.creation topic and will send to all the users a mail to notify them about the creation of a new tournament.

2.4.5 Add educator to a tournament

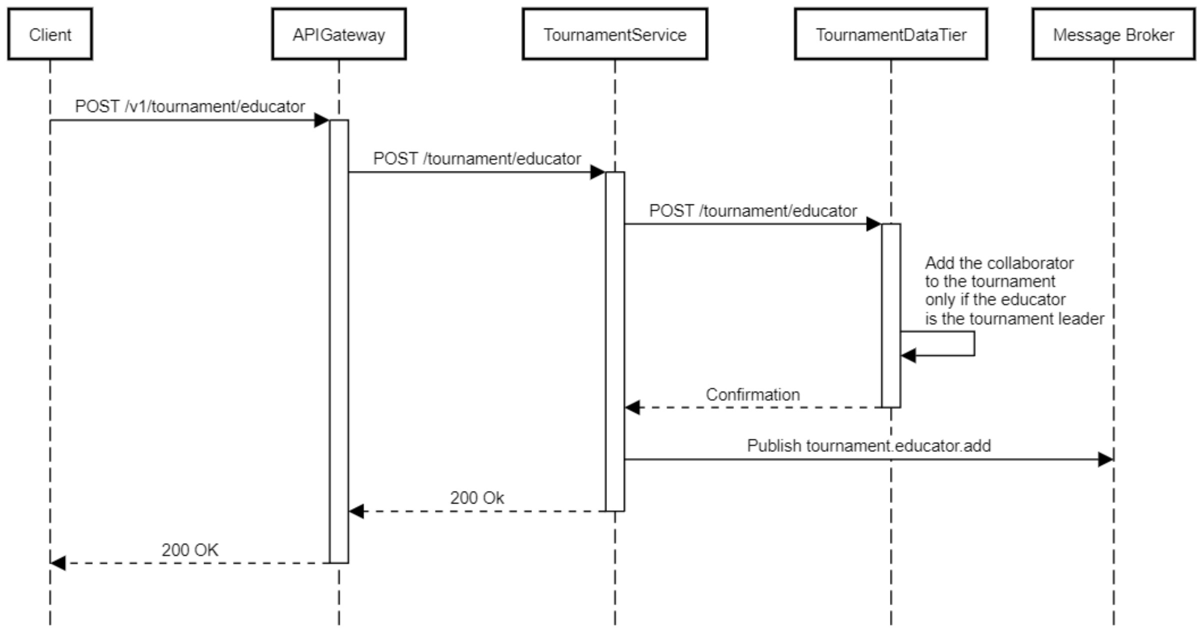
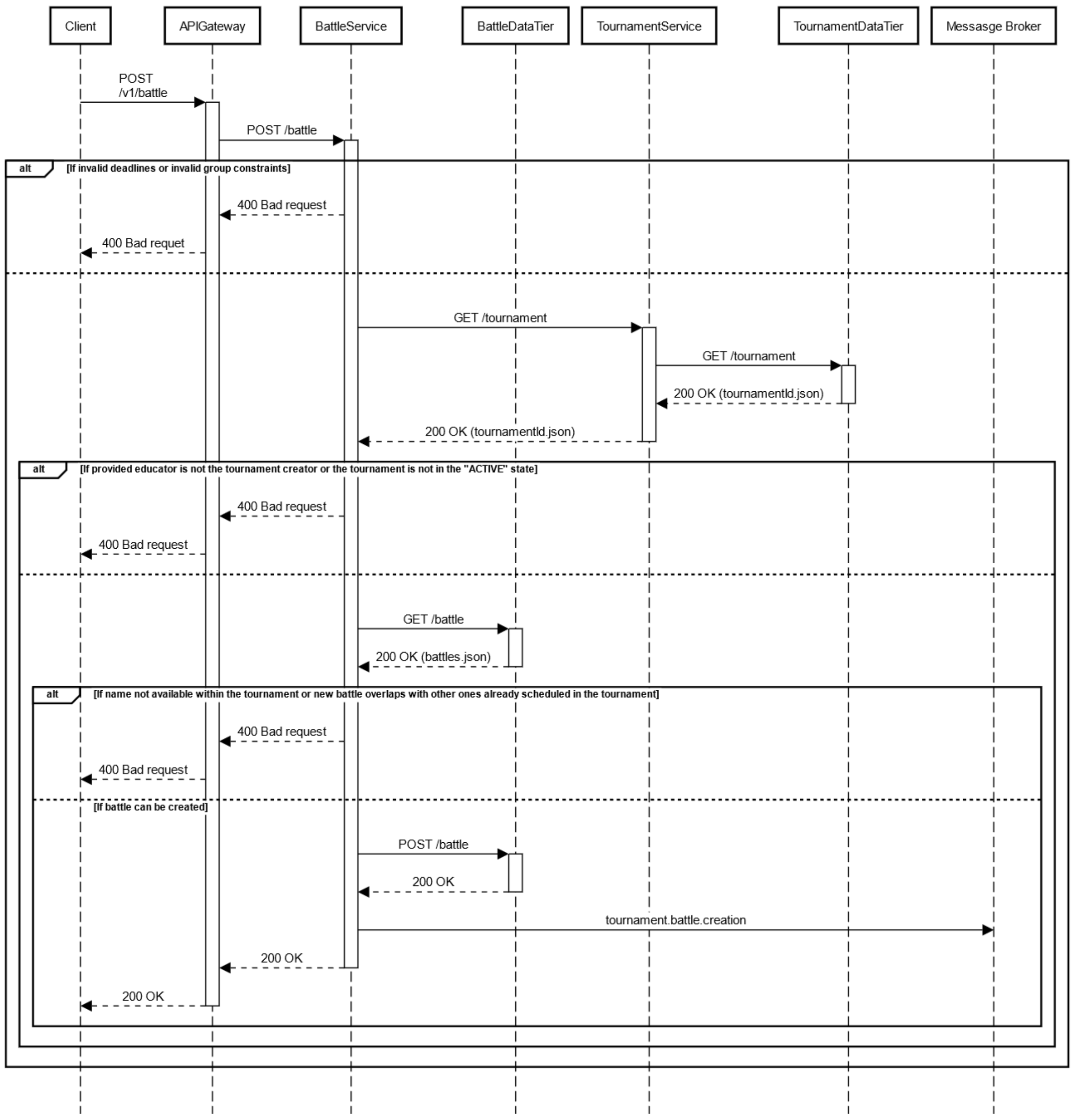

Client
APIGateway
TournamentService
TournamentDataTier
Message Broker
POST /v1/tournament/educator
POST /tournament/educator
POST /tournament/educator
Add the collaborator to the tournament only if the educator is the tournament leader
Confirmation
Publish tournament.educator.add
200 Ok
200 OK

Figure 2.25

The educator that wants to add a collaborator to a tournament that created is identified by its API key. There is no need to check if the request came from an educator by looking at the database, it is intrinsically specified by the key.

There is the necessity to check if the educator is the creator of the tournament since it is not possible to know it a priori. This is done in the tournament data tier by using a trigger.

2.4.6 Create battle



Client
APIGateway
BattleService
BattleDataTier
TournamentService
TournamentDataTier
Messasge Broker
POST /v1/battle
POST /battle
alt
[If invalid deadlines or invalid group constraints]
400 Bad request
400 Bad requet
GET /tournament
GET /tournament
200 OK (tournamentId.json)
200 OK (tournamentId.json)
alt
[If provided educator is not the tournament creator or the tournament is not in the "ACTIVE" state]
400 Bad request
400 Bad request
GET /battle
200 OK (battles.json)
alt
[If name not available within the tournament or new battle overlaps with other ones already scheduled in the tournament]
400 Bad request
400 Bad request
[If battle can be created]
POST /battle
200 OK
tournament.battle.creation
200 OK
200 OK

Figure 2.26

When creating a new battle, the system must check some aspect on the data given by the educator. First, the following two properties must hold: *registrationDeadline* < *submissionDeadline* and *minGroupMembers* < *maxGroupMembers*. Then, the system must ensure that the name of the new battle is unique within the tournament and that it does not overlaps with other battles already scheduled within the tournament context. Moreover, battle cannot be created if the given tournament is not in the "ACTIVE" state or if the given educator (the one who wants to create the battle) is not the creator of the tournament. In order to check so, the Battle Service performs a REST request to the Tournament Service to get the tournament's id and its creator id.

2.4.7 Evaluate group

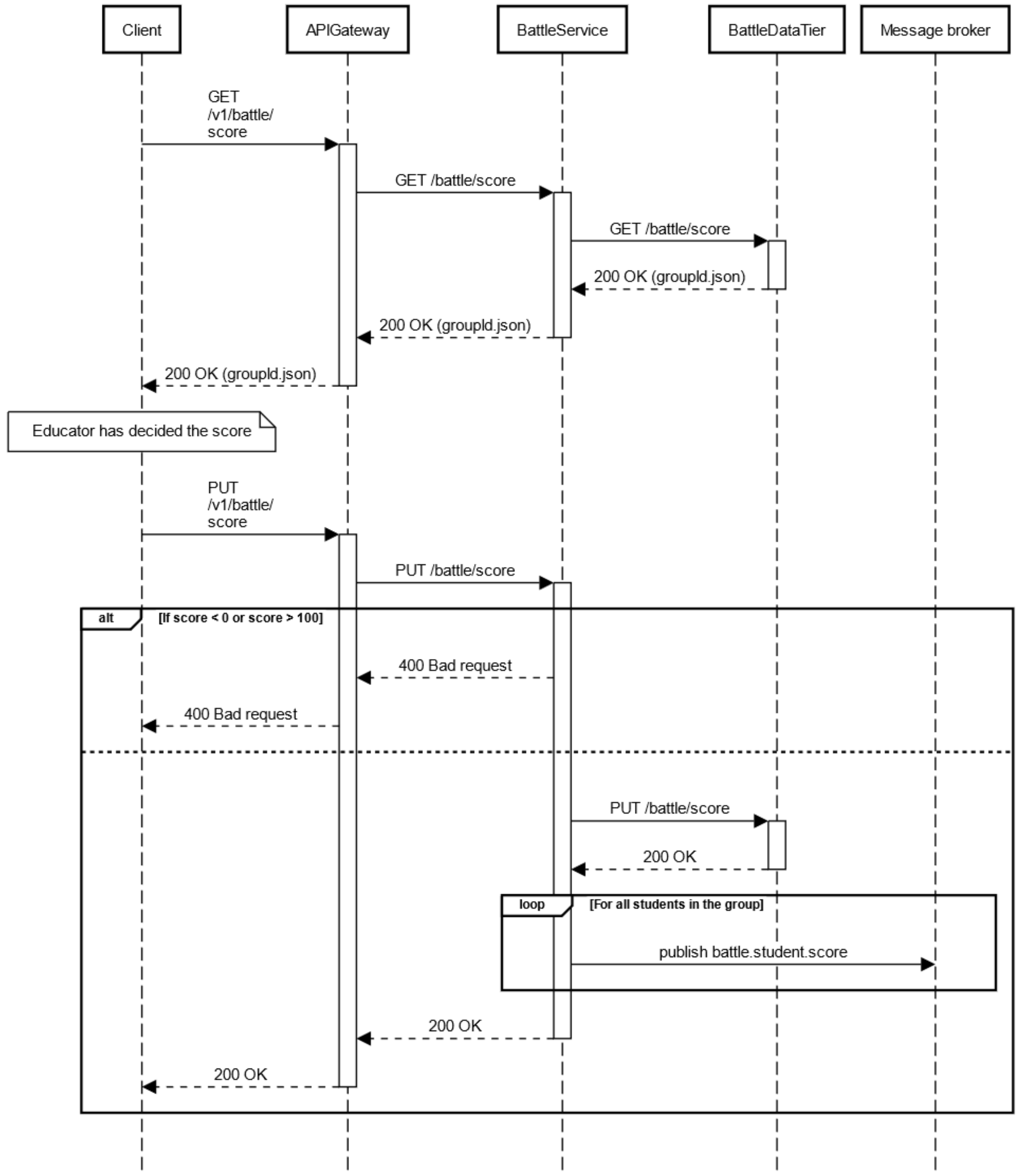


Figure 2.27

When an educator wants to manually evaluate a group, the first thing he/she has to do is to perform a RESTful request as following: GET /battle/score. In this way, the system responds by providing the temporary score automatically calculated (i.e. the score of the material provided by the group with the last push on the main branch of the repository) and with the link to the group cloned repository. The educator can now look through this material and then, finally, he/she upload a new score for the group by doing the request PUT /battle/group. The system will get this new score, average it with the original one already stored and publish the result.

2.4.8 Close tournament

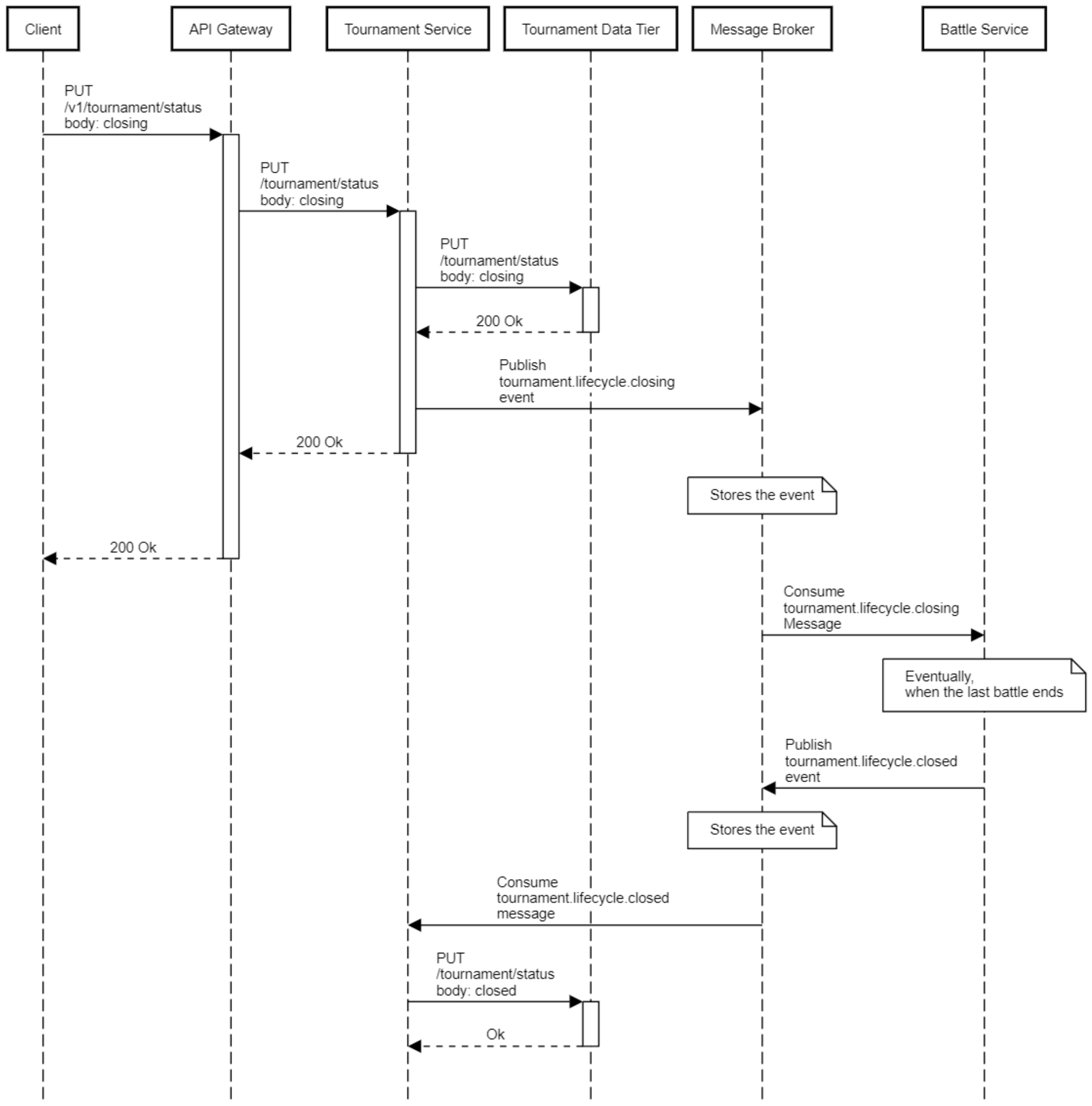

Client
API Gateway
Tournament Service
Tournament Data Tier
Message Broker
Battle Service
PUT /v1/tournament/status body: closing
PUT /tournament/status body: closing
PUT /tournament/status body: closing
200 Ok
Publish tournament.lifecycle.closing event
200 Ok
Stores the event
200 Ok
Consume tournament.lifecycle.closing Message
Eventually, when the last battle ends
Publish tournament.lifecycle.closed event
Stores the event
Consume tournament.lifecycle.closed message
PUT /tournament/status body: closed
Ok

Figure 2.28

When setting the tournament status to “closing”, the battle service will no longer be able to create battles inside the tournament, but the tournament will still be active for users until the remaining battle ends. Then it will be possible to proceed with the ranking and end procedure.

Ensured by the battle service:

- Battle can be created ⇔Tournament status is active
- Every battle that is currently scheduled for the future will be eliminated at the receiving of the closing notification

After the closure of the tournament, the tournament manager will trigger the beginning of the badge evaluation:

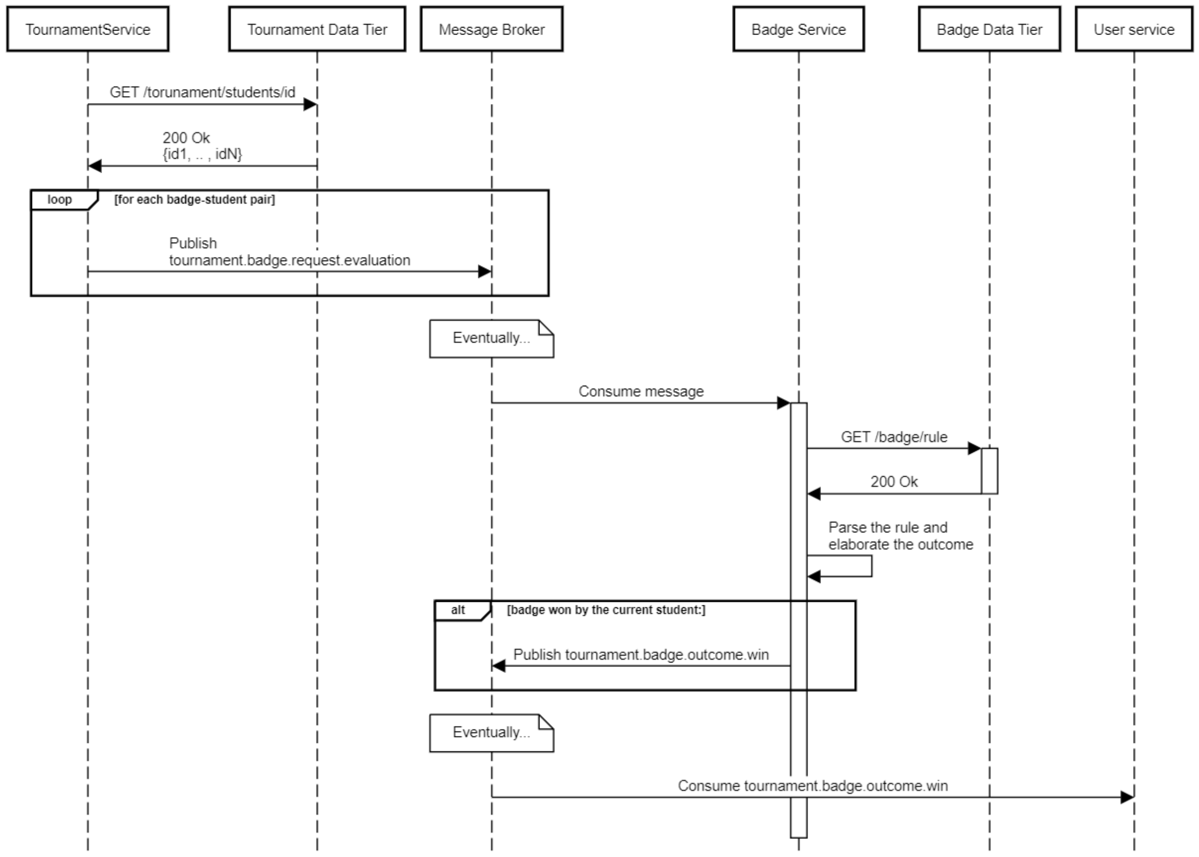


Figure 2.29

The goal of this solution is to maximize parallelism for the badge evaluation, the messages on topic tournament.badge.request.evaluation are sent for each student-badge couple. The badge service is meant to have many instances so that the evaluation can take place in parallel. When a win is found, the badge service notifies the user service that shows the badge in the profile of the winner.

2.4.9 Calculate tournament ranking

Every time a battle ends, the tournament receives the battle results from the battle service and stores the sum of all the previous results for each student in the tournament data tier. This way, it is possible for every user to see the tournament ranking even when there are still battle running. The drawback of this approach is that there might be inconsistencies with respect to the last battle score since the data is elaborated one by one for each student. This is not a problem because the data will eventually become consistent, the system has been designed to be fast in this regard.

(See "notify about battle score")

2.4.10 Create badge

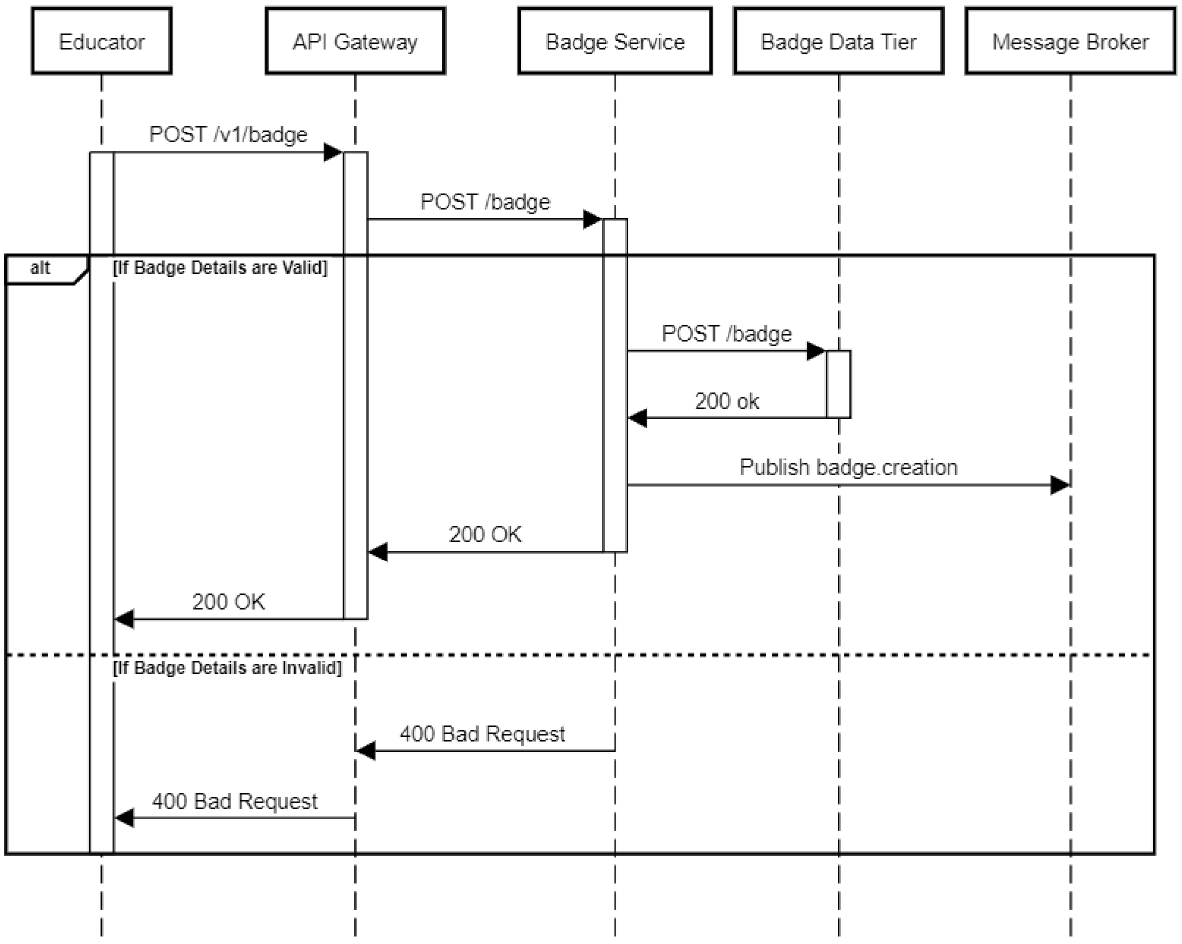


Figure 2.30

The API gateway verifies if the request comes from an educator. The badge service evaluates the predicate given in the request, if it finds it valid it will proceed to request to the data tier to save it persistently in the database.

2.4.11 Delete badge

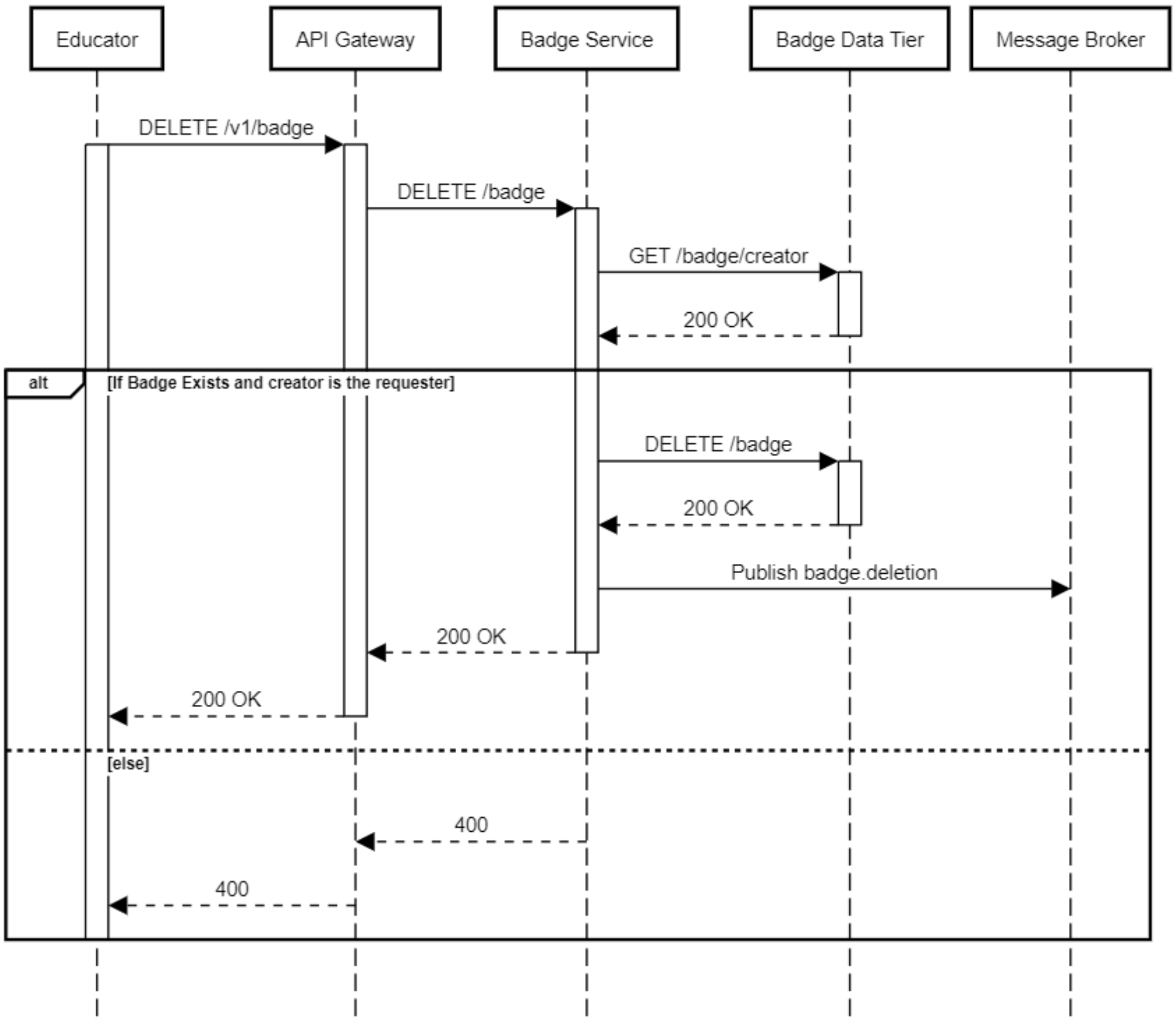


Figure 2.31

The API gateway verifies if the request comes from an educator. The badge service deletes the badge only if it belongs to the educator that is trying to eliminate it.

2.4.12 Join tournament

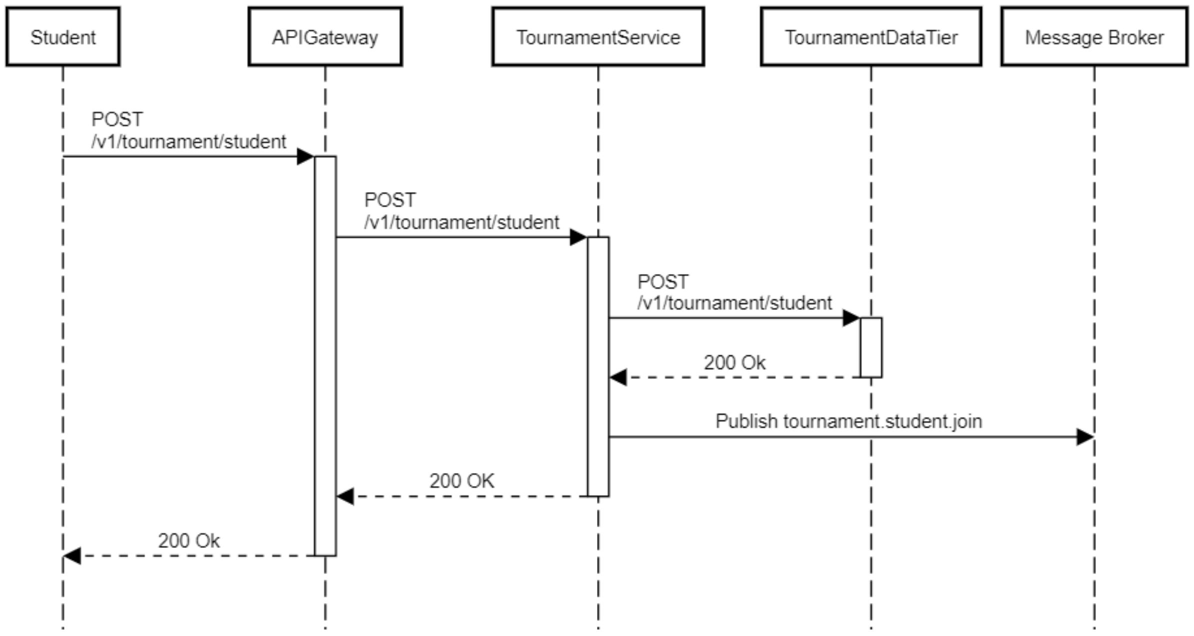

Student
APIGateway
TournamentService
TournamentDataTier
Message Broker
POST /v1/tournament/student
POST /v1/tournament/student
POST /v1/tournament/student
200 Ok
Publish tournament.student.join
200 OK
200 Ok

Figure 2.32

The tournament data tier will ensure that the student is unique inside the tournament.

2.4.13 Join battle

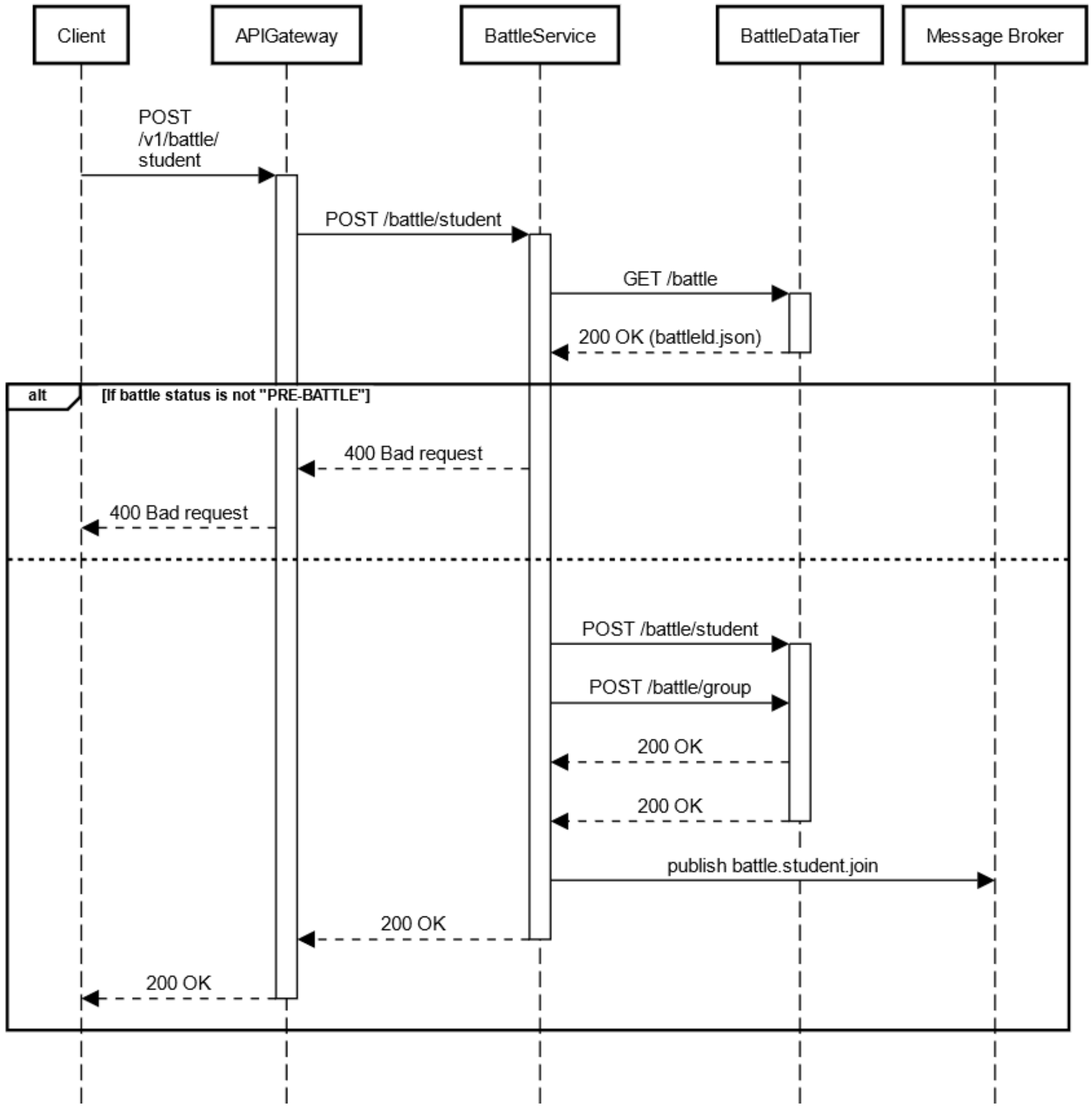


Figure 2.33

As mentioned in the RASD, when a student joins a battle, he/she does so by register as a singleton. When the registration is completed, then he/she can invite other students in his/her group or accept group invitation (if any has been received before the submission deadline). Obviously, this must be done according to the constraints on the number of the member of a group chosen by the creator of the battle. What just stated, can be done only when the battle state is “PRE-BATTLE”.

2.4.14 Invite student in the group

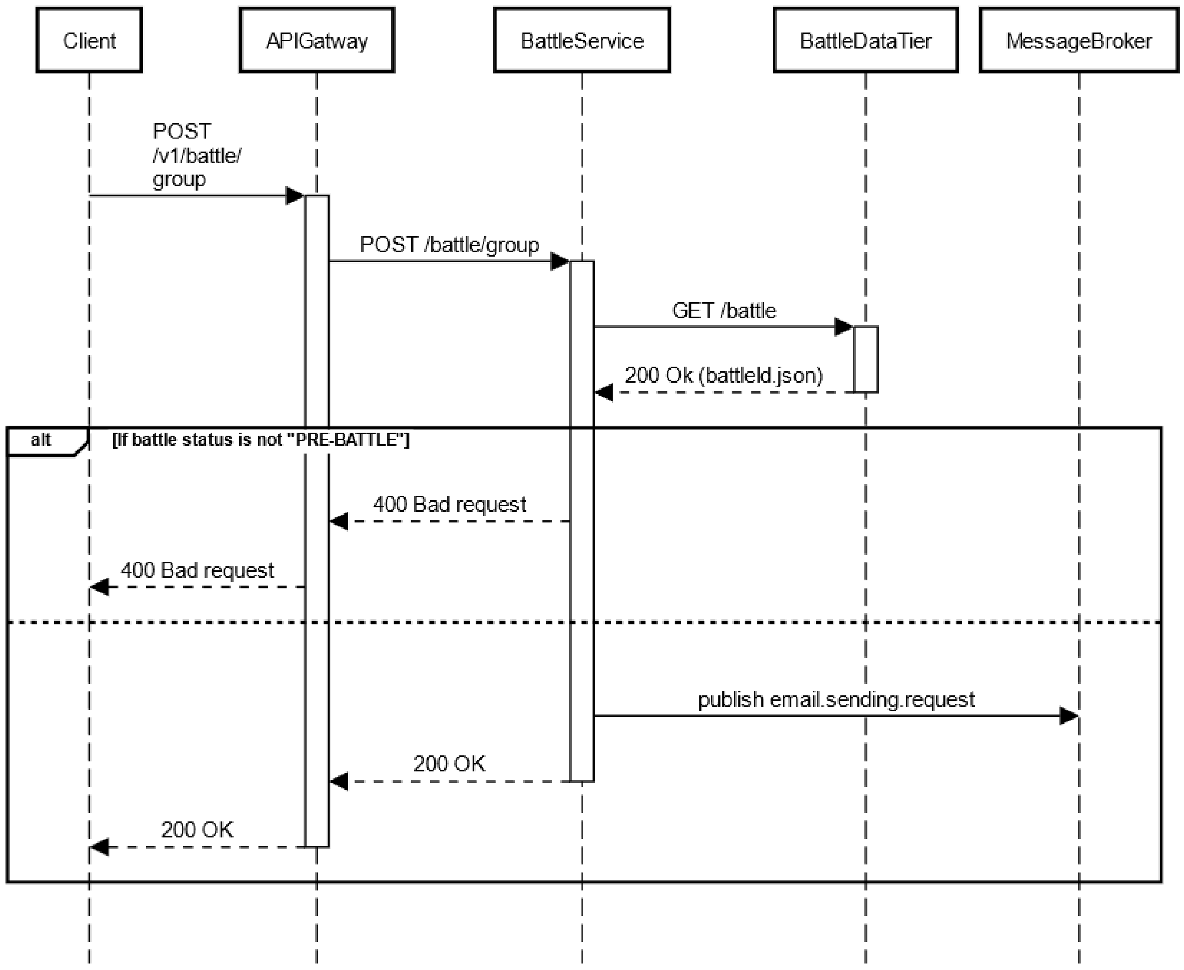


Figure 2.34

After checking that the battle state is "PRE-BATTLE", the battle-service publishes a message on the Kafka topic email.sending.request. The user-service consumes event from this topic and handles the email sending process, using all the information provided in the Kafka message.

2.4.15 Respond to group invitation

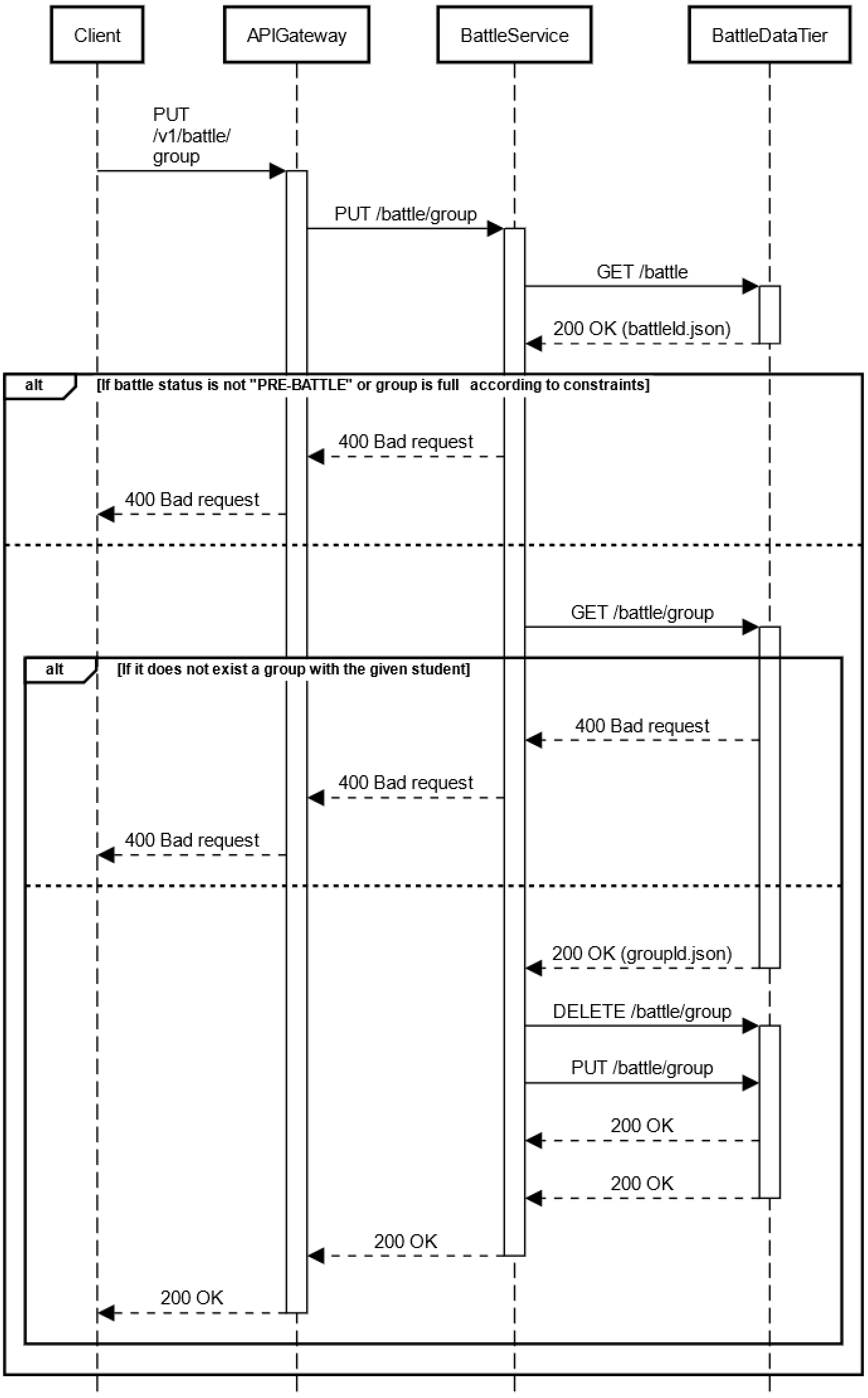


Figure 2.35

The system prevents students from joining groups when full or when the battle state is not “PRE-BATTLE”. In order to check that the student from whom the request is coming is indeed registered to the battle, the system checks if it is already in any group by doing the request GET /battle/group. This can be done since, when a student joins a battle, he/she does so by registering as a singleton (i.e. a group formed only by him/her). Either a student is accepting the invitation while still being a singleton or he/she has previously joined another group, the systems ensures that he/she leave the previous group before joining the new one.

2.4.16 Participate to battle

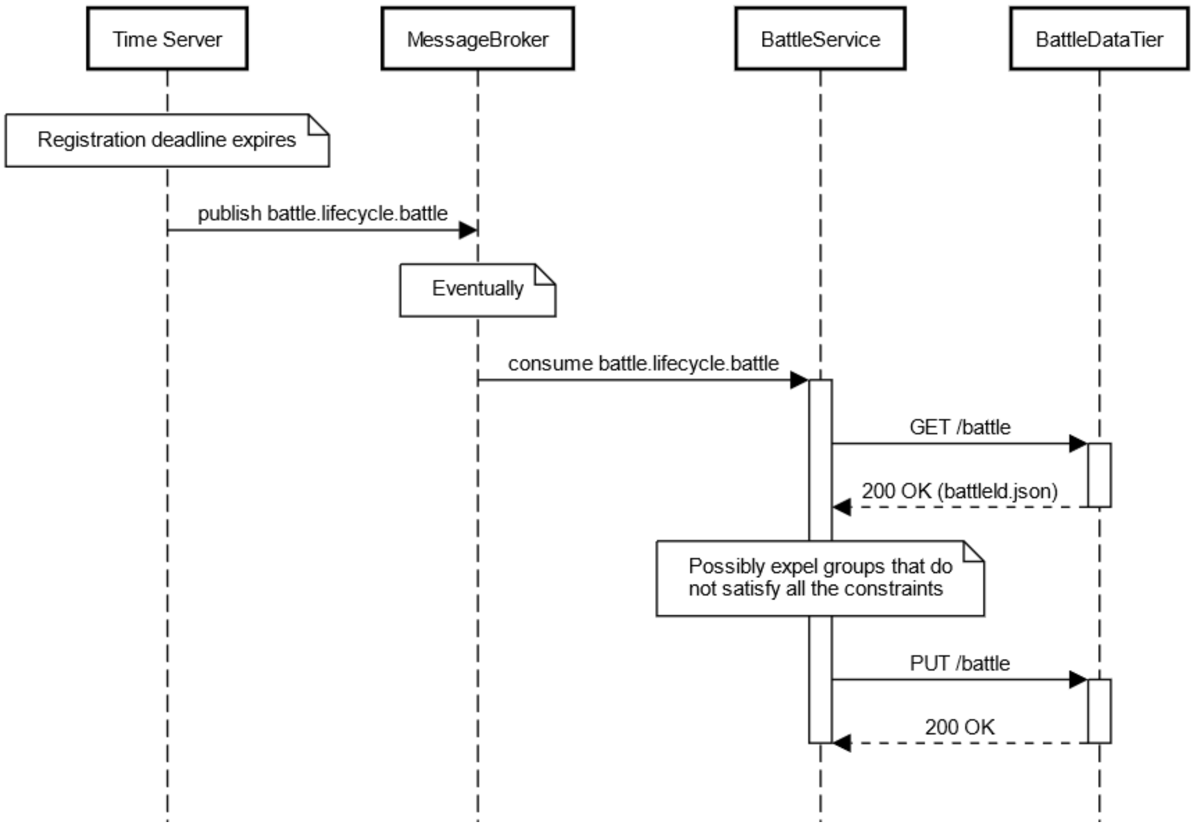

Time Server
MessageBroker
BattleService
BattleDataTier
Registration deadline expires
publish battle.lifecycle.battle
Eventually
consume battle.lifecycle.battle
GET /battle
200 OK (battleId.json)
Possibly expel groups that do not satisfy all the constraints
PUT /battle
200 OK

Figure 2.36

2.4.17 Quit battle

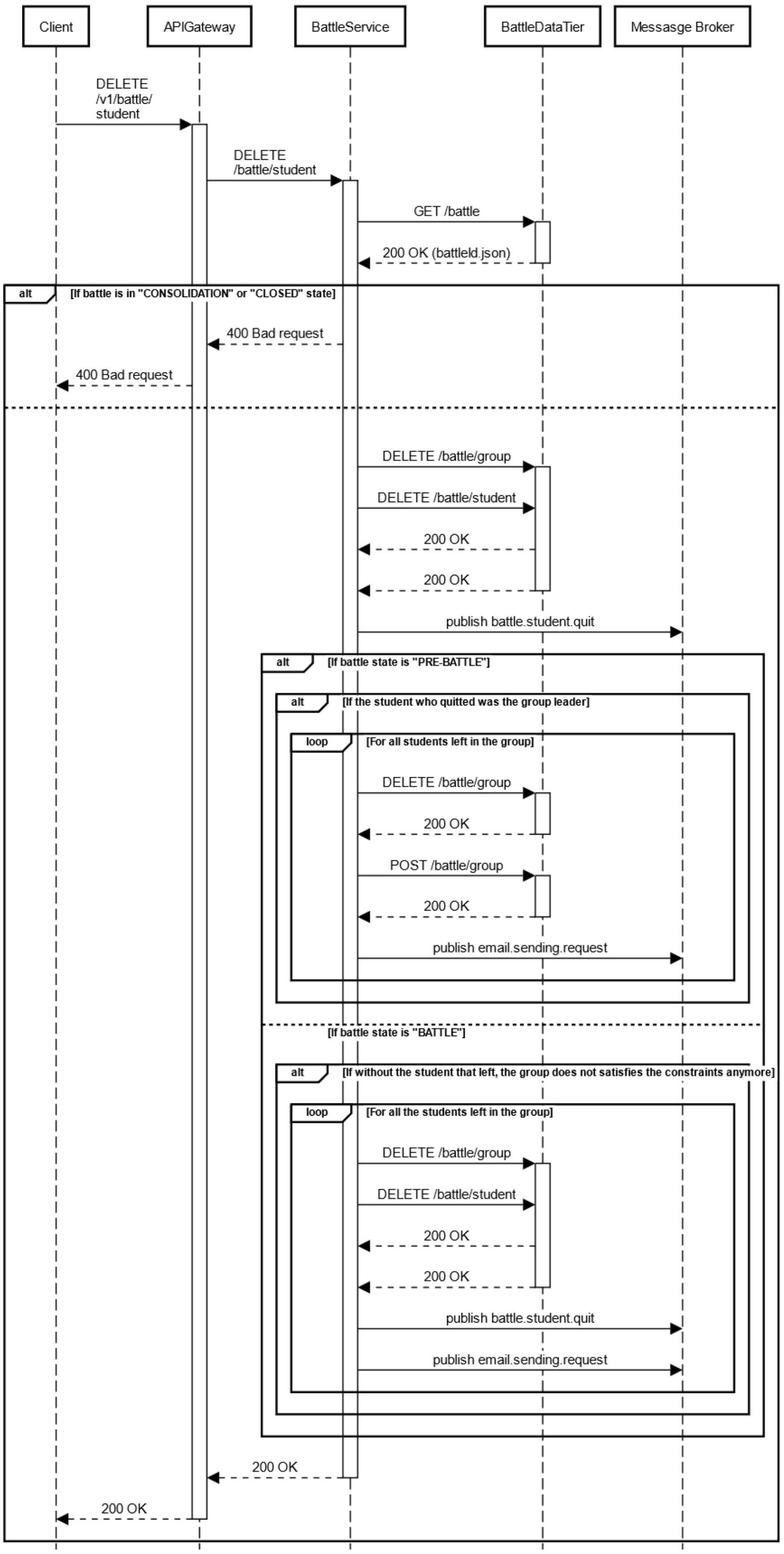

Client
APIGateway
BattleService
BattleDataTier
Messasge Broker
DELETE /v1/battle/ student
DELETE /battle/student
GET /battle
200 OK (battleId.json)
alt [If battle is in "CONSOLIDATION" or "CLOSED" state]
400 Bad request
400 Bad request
DELETE /battle/group
DELETE /battle/student
200 OK
200 OK
publish battle.student.quit
alt [If battle state is "PRE-BATTLE"]
alt [If the student who quitted was the group leader]
loop [For all students left in the group]
DELETE /battle/group
200 OK
POST /battle/group
200 OK
publish email.sending.request
[If battle state is "BATTLE"]
alt [If without the student that left, the group does not satisfies the constraints anymore]
loop [For all the students left in the group]
DELETE /battle/group
DELETE /battle/student
200 OK
200 OK
publish battle.student.quit
publish email.sending.request
200 OK
200 OK

Figure 2.37

Students can quit a battle they previously registered to only if its state is either "PRE-BATTLE" (i.e. registration-deadline not expired yet) or "BATTLE" (i.e. submission-deadline not expired yet). The system behavior depends on factor such as the battle's status and on whether the student who quits is the group leader or not:

- Battle in "PRE-BATTLE" state
    - Student is the group leader: the group is disbanded, he/she is no longer register to the battle and every group member left (if any) in now register as a singleton
    - Student is not the group leader: he/she is no longer register to the battle (if the group does not satisfy anymore the constraints on the number of members, it will be excluded by the registration deadline in case a solution will not be found)
- Battle in "BATTLE" state (either the student who left was the group leader or not, the system behavior depends on the number of group members that are still in the group)
    - Constraints still satisfied: the system only registers that the student leaves the group
    - Constraints no longer satisfied: the group is disjointed and all its members are kicked from the battle

2.4.18 Quit tournament

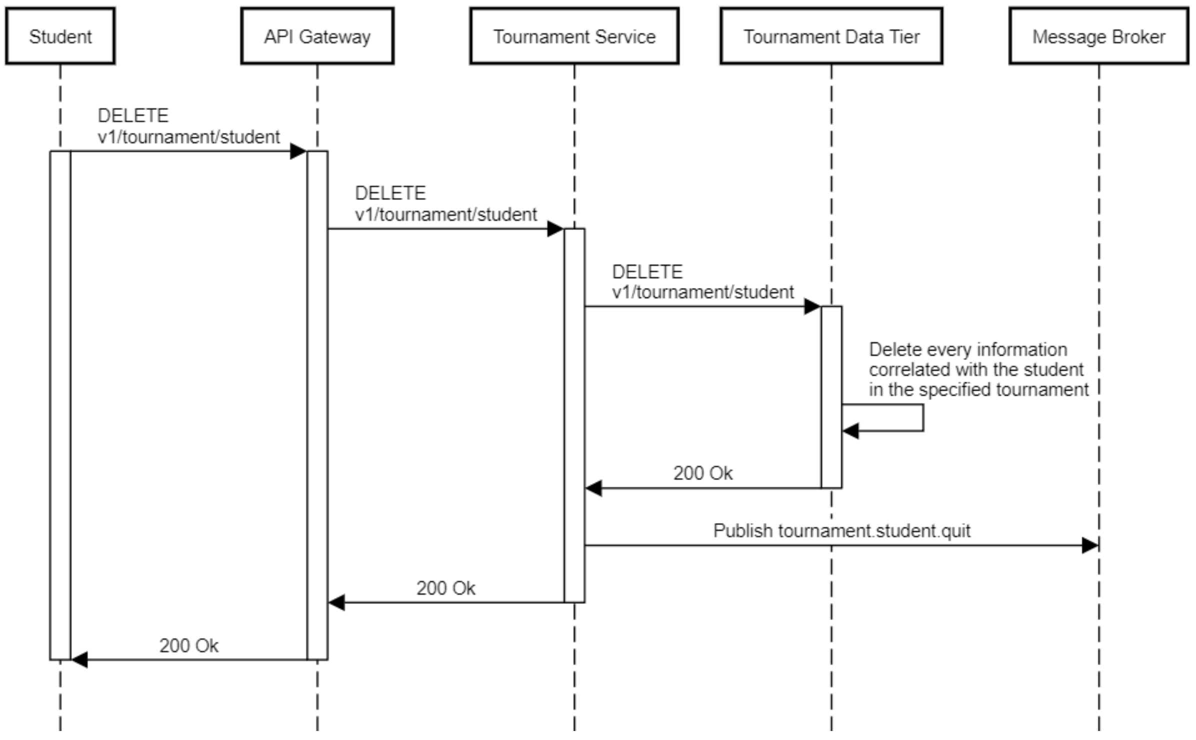

Student
API Gateway
Tournament Service
Tournament Data Tier
Message Broker
DELETE v1/tournament/student
DELETE v1/tournament/student
DELETE v1/tournament/student
Delete every information correlated with the student in the specified tournament
200 Ok
Publish tournament.student.quit
200 Ok
200 Ok

Figure 2.38

When a student decides to quit a tournament, the tournament tier will remove all the information about the student in the tournament.

2.4.19 Calculate submission score

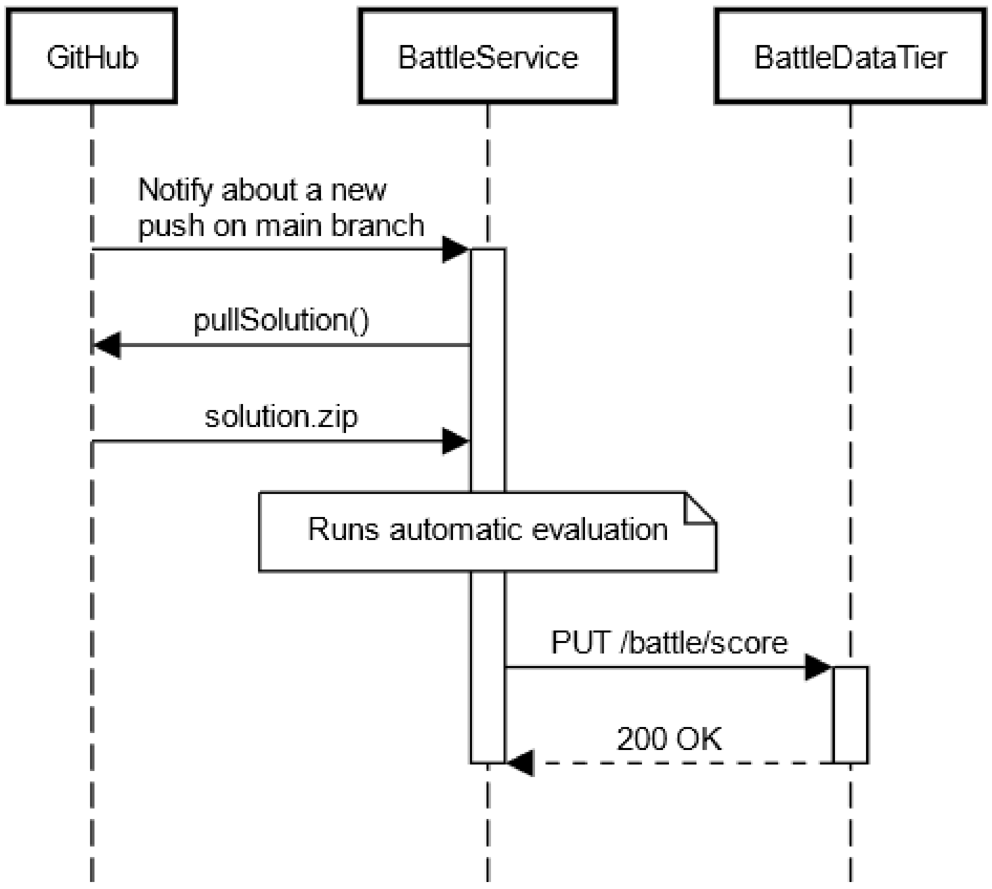


Figure 2.39

Every time a new push on the main branch of the GitHub repository has been performed by some group member, the system is informed by the GitHub Actions. So, it can pull the latest version of the code, evaluate it according to the relevant aspect chosen during the battle creation and register a temporary score. This score is registered in the battle database but is not publish to the Kafka broker. Only definitive scores are propagated with Kafka.

2.5 Components interfaces

2.5.1 TournamentI

1. Creation of the tournament

HEADER	
METHOD	POST
ENDPOINT	tournament/

BODY	{
	name: String,
	registrationDeadline: String [ISO 8601 format],
	description: String,
	educatorID: Long, //ID of tournament's creator
	badges: List of Long //badge IDs
	}

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{outcome: "Ok" }

STATUS CODE	400
SERIALIZED DATA	{outcome:" Bad request", reason: "Name not available"}

2. Join tournament

HEADER	
METHOD	POST
ENDPOINT	tournament/student

BODY	{
	tournamentID: Long,
	studentID: Long
	}

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{outcome: "Ok" }

3. Quit tournament

HEADER	
METHOD	DELETE
ENDPOINT	tournament/student

BODY	{
	tournamentID: Long,
	studentID: Long
	}

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

4. Add educator to the tournament

HEADER	
METHOD	POST
ENDPOINT	tournament/educator

BODY	{
	requesterID: String,
	newEducatorID: String,
	tournamentID: String
	}

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{outcome: “Ok” }

5. Change tournament status

HEADER	
METHOD	PUT
ENDPOINT	tournament/status

BODY	{
	educatorID: Long,
	tournamentID: Long,
	status: String
	}

Possible tournament statuses are: “PREPARATION”, “ACTIVE”, “CLOSING”, “CLOSED”.

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{outcome: “Ok”}

6. Calculate ranking and winner of the tournament

HEADER	
METHOD	GET
ENDPOINT	tournament/ranking
BODY	{tournamentID: Long, firstIndex: Integer, lastIndex: Integer }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{outcome: “Ok”, ranking: [

	{“studentID”: “001”, “score”: 1000}, {“studentID”: “002”, “score”: 950}, {“studentID”: “003”, “score”: 920}, //... other ranking entries]}

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request”, reason: “Invalid indexes” }

2.5.2 TournamentDataTierI

The tournament data tier has all the interfaces of the tournament service, plus the following:

1. Get the id of the students participating in a tournament

HEADER	
METHOD	GET
ENDPOINT	tournament/students/id

BODY	{
	tournamentID: String
	}

<u>Possible responses:</u>

STATUS CODE	200

SERIALIZED DATA	{
	studentIds: [Long]
	}

2. **Update the score of the student in a tournament**

HEADER	
METHOD	PUT
ENDPOINT	/student/score

BODY	{
	studentId: Long
	tournamentId: Long
	score: Int
	}

<u>Possible responses:</u>

STATUS CODE	200
SERIALIZED DATA	

2.5.3 BattleI

1. Creation of the battle

HEADER	
METHOD	POST
ENDPOINT	/battle
BODY	{ name: String, description: String, registrationDeadline: String [ISO 8601 format], submissionDeadline: String [IOS 8601 format], minGroupMember: Integer, maxGroupMember: Integer, educatorID: Long, //ID of battle's creator tournamentID: Long, //ID of the tournament in which battle takes place repositoryLink: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: "Ok" }

STATUS CODE	400
SERIALIZED DATA	{outcome: "Bad request", reason: "Invalid parameter(s)" }

2. Join battle

HEADER	
METHOD	POST
ENDPOINT	/battle/student
BODY	{ studentID: String, battleID: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: "Ok" }

3. Quit battle

HEADER	
METHOD	DELETE
ENDPOINT	/battle/student
BODY	{ battleID: Long, studentID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

4. invite student in a group

HEADER	
METHOD	POST
ENDPOINT	/battle/group
BODY	{ requesterID: Long, groupID: Long, battleID: Long, otherStudentID: LONG //ID of the student to invite }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

5. **Join group**

HEADER	
METHOD	PUT
ENDPOINT	/battle/group
BODY	{ acceptingStudentID: Long, battleID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

6. Quit group

HEADER	
METHOD	DELETE
ENDPOINT	/battle/group
BODY	{ battleID: Long, studentID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

7. Manual evaluate group

HEADER	
METHOD	PUT
ENDPOINT	/battle/score
BODY	{ Score: Integer, battleID: Long, groupID: Long, educatorID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

8. Get group partial score and repository link

HEADER	
METHOD	GET
ENDPOINT	battle/score
BODY	{ educatorID: Long, groupID: Long, battleID: Long }

Possible responses:

STATUS CODE	200

SERIALIZED DATA	{ outcome: “Ok”,
	Size: Integer,
	Groups: [
	        group1: {groupID: Long, score: Integer, repoLink: String},
	        group2: {groupID: Long, score: Integer, repoLink: String},
	        group3: {groupID: Long, score: Integer, repoLink: String},
	        //other groups
	] }

2.5.4 BattleDataTierI

1. Get battle with the given ID

HEADER	
METHOD	GET
ENDPOINT	/battle
BODY	{ battleID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, Size: Integer, Battles: [battle1: {json file}, battle2: {json file}, battle3: {json file}, //other battles] }

2. Get battles within the given tournament

HEADER	
METHOD	GET
ENDPOINT	/tournament
BODY	{ tournamentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, Size: Integer, Battles: [battle1: {json file}, battle2: {json file}, battle3: {json file}, //other battles] }

3. Create a battle

HEADER	
METHOD	POST
ENDPOINT	/battle
BODY	{ name: String, description: String, registrationDeadline: String [ISO 8601 format], submissionDeadline: String [ISO 8601 format], minGroupMember: Integer,

	maxGroupMember: Integer, tournamentID: Long, creatorID: Long, repositoryLink: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

4. Get group partial score and repository link

HEADER	
METHOD	GET
ENDPOINT	/battle/score
BODY	{ battleID: Long, groupID: Long }

Possible responses:

STATUS CODE	200

SERIALIZED DATA	{ outcome: “OK”,
	Size: Integer,
	Groups: [
	group1: {groupID: Long, score: Integer, repoLink: String},
	group2: {groupID: Long, score: Integer, repoLink: String},
	group3: {groupID: Long, score: Integer, repoLink: String},
	//other groups
	]
	}

5. Change battle status

HEADER	
METHOD	PUT
ENDPOINT	/battle
BODY	{ battleID: Long, newState: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

6. Get the group with the given student in the given battle

HEADER	
METHOD	GET
ENDPOINT	/battle/group
BODY	{ battleID: Long, studentID: Long }

Possible responses:

STATUS CODE	200

SERIALIZED DATA	{ outcome: “OK”,
	Size: Integer,
	students: [
	student1: {json file},
	student2: {json file},
	student3: {json file},
	//other students
	]
	}

7. Register the score for a group

HEADER	
METHOD	PUT
ENDPOINT	/battle/score
BODY	{ battleID: Long, groupID: Long, score: Integer }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

8. Register that a student has joined a battle

HEADER	
METHOD	POST
ENDPOINT	/battle/student
BODY	{ battleID: Long, studentID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

9. Join a group

HEADER	
METHOD	PUT
ENDPOINT	/battle/group
BODY	{ battleID: Long, groupID: Long, studentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: "OK" }

10. Quit battle

HEADER	
METHOD	DELETE
ENDPOINT	/battle/student
BODY	{ battleID: Long, studentID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: "OK" }

11. Quit group

HEADER	
METHOD	DELETE
ENDPOINT	/battle/group
BODY	{ battleID: Long, studentID: Long, groupID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: "Ok" }

2.5.5 UserI

1. Get user's main information

HEADER	
METHOD	GET
ENDPOINT	/user

BODY	{ userID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, user: { nickname: String, tournaments: tournamentID [ ], battles: battleID [ ], badges: badgeID [ ] }}

2.5.6 UserDataTierI

1. Get user’s main information

HEADER	
METHOD	GET
ENDPOINT	/user
BODY	{ userID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, user: { nickname: String, tournaments: tournamentID [ ], battles: battleID [ ], badges: badgeID [ ] }}

2. Post the new Tournament

HEADER	
METHOD	POST
ENDPOINT	/tournament
BODY	{ tournamentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

3. Post student joining a tournament

HEADER	
METHOD	POST
ENDPOINT	/tournament/student
BODY	{ tournamentID: Long,

	studentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

4. Delete a student quitting a tournament

HEADER	
METHOD	DELETE
ENDPOINT	/tournament/student
BODY	{ tournamentID: Long, studentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

5. Post the new Battle

HEADER	
METHOD	POST
ENDPOINT	/battle
BODY	{ battleID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

6. Post student joining a battle

HEADER	
METHOD	POST
ENDPOINT	/battle/student
BODY	{ battleID: Long, studentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

7. Delete a student quitting a battle

HEADER	
METHOD	DELETE
ENDPOINT	/battle/student
BODY	{ battleID: Long, studentID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

8. Assign a badge to the student which won it

HEADER	
METHOD	POST
ENDPOINT	/tournament/student/badge
BODY	{ tournamentID: Long, studentID: Long, badgeID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

9. Post the creation of a badge

HEADER	
METHOD	POST
ENDPOINT	/educator/badge
BODY	{ educatorID: Long, badgeID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

10. Delete a badge

HEADER	
METHOD	DELETE
ENDPOINT	/badge
BODY	{ badgeID: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

11. Post user information

HEADER	
METHOD	POST
ENDPOINT	/user
BODY	{ email: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok” }

2.5.7 BadgeI

1. Create new badge

HEADER	
METHOD	POST
ENDPOINT	/badge

BODY	{
	badgeId: String,
	rules: [String],
	educatorId: String
	}

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok”}

2. **Delete badge**

HEADER	
METHOD	DELETE
ENDPOINT	/badge
BODY	{badgeID: Long, educatorId: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “Ok”}

2.5.8 BadgeDataTierI

The badge data tier has all the interfaces of the badge service, plus the followings:

1. Get the creator of a badge

HEADER	
METHOD	GET
ENDPOINT	/badge/creator
BODY	{ badgeId: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ educatorId: Long }

2. Get all the rules of a specific badge

HEADER	
METHOD	GET
ENDPOINT	/badge/rule
BODY	{ badgeId: Long }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ rules: [String] }

2.5.9 AuthenticationI

1. Login

HEADER	
METHOD	POST
ENDPOINT	/login
BODY	{ email: String, password: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

2. Signup

HEADER	
METHOD	POST
ENDPOINT	/signup
BODY	{ email: String, password: String, name: String, surname: String, nickname: String, accountType: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

3. Confirm registration

HEADER	
METHOD	PUT
ENDPOINT	/signup
BODY	{ email: String, password: String, accountType: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK” }

STATUS CODE	400
SERIALIZED DATA	{ outcome: “Bad request” }

2.5.10 AuthenticationDataTierI

1. Get all the emails already used by CKB users

HEADER	
METHOD	GET
ENDPOINT	/email
BODY	

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, size: Integer, emails: [email1: String, email2: String, email3: String, ..., emailN: String] }

2. Get login information in order to check the validity of the account

HEADER	
METHOD	GET
ENDPOINT	/login
BODY	{ email: String }

Possible responses:

STATUS CODE	200
SERIALIZED DATA	{ outcome: “OK”, email: String, password: String }

2.6 Asynchronous notifications

2.6.1 Notify about the creation of a tournament

Topic name	tournament.creation

Pub	Sub
TournamentService	UserService, TimeServer

Message structure

Field	description	Type
tournamentId	Id of the new tournament	Long
name	Name of the new tournament	String
creatorId	Id of the creator	Long
registrationDeadline	Time of tournament beginning	String

The user service will consume the message and call POST /tournament in the user data tier interface.

2.6.2 Notify about student join a tournament

Topic name	tournament.student.join

Pub	Sub
TournamentService	UserService

Message structure

Field	description	Type
studentId	Id of the student	Long
tournamentId	Id of the tournament	Long

The user service, upon receiving of the message, will update its knowledge about the student by adding a tournament in which the user participated. It will use POST /tournament/student.

2.6.3 Notify about student quitting a tournament

Topic name	tournament.student.quit

Pub	Sub
TournamentService	UserService

Message structure

Field	description	Type
studentId	Id of the student	Long
tournamentId	Id of the tournament	Long

The user service, upon receiving of the message, will remove the student from the tournament. It will use DELETE /tournament/student

2.6.4 Notify about tournament registration deadline expiration

Topic name	tournament.lifecycle.active

Pub	Sub
TimeServer	TournamentServer

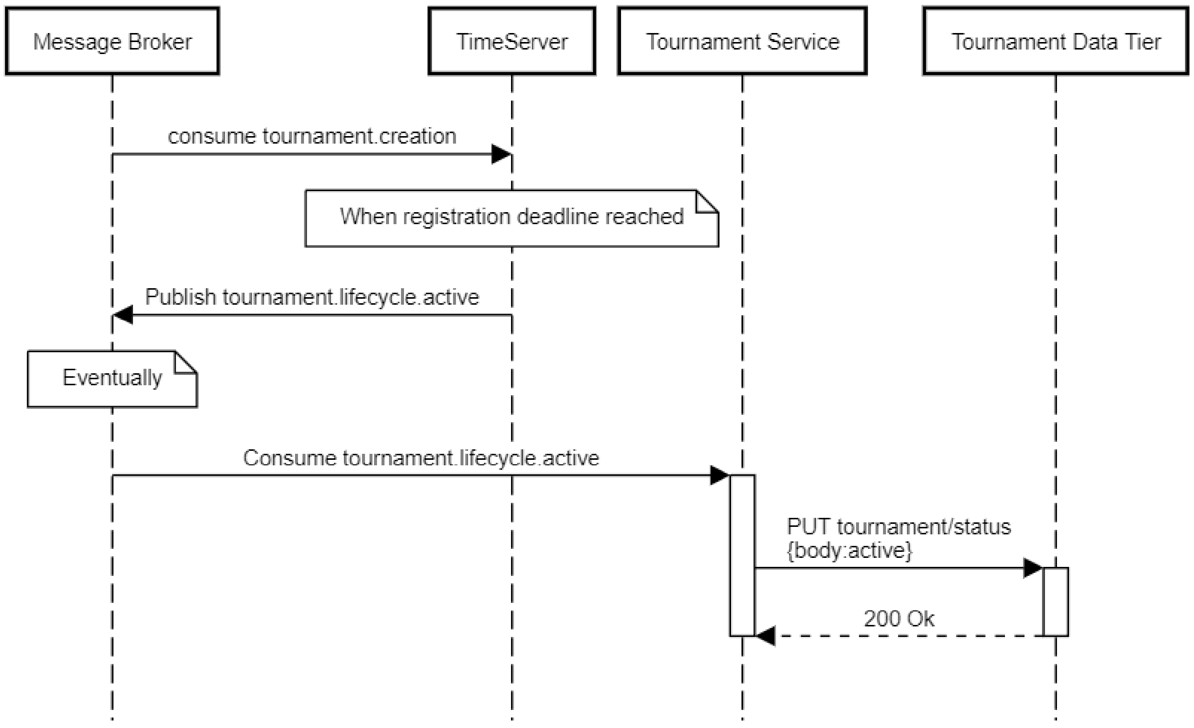

Message Broker
TimeServer
Tournament Service
Tournament Data Tier
consume tournament.creation
When registration deadline reached
Publish tournament.lifecycle.active
Eventually
Consume tournament.lifecycle.active
PUT tournament/status {body:active}
200 Ok

Figure 2.40

<u>Message structure</u>

Field	description	Type
tournamentId	Id of tournament	String

2.6.5 Notify about tournament closure

Topic name	tournament.lifecycle.closing

Pub	Sub
TournamentService	BattleService

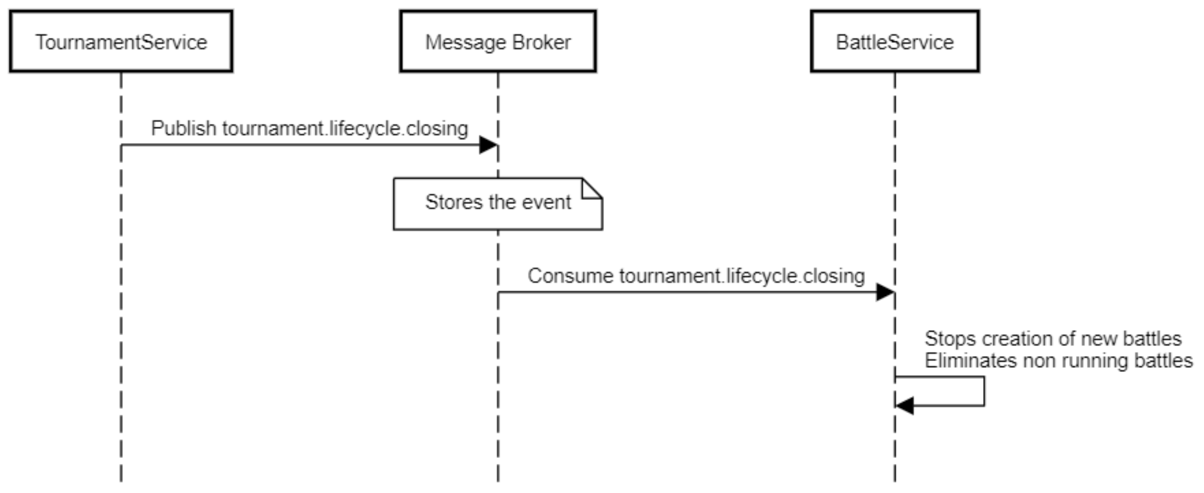

TournamentService
Message Broker
BattleService
Publish tournament.lifecycle.closing
Stores the event
Consume tournament.lifecycle.closing
Stops creation of new battles
Eliminates non running battles

Figure 2.41

<u>Message structure</u>

Field	description	Type
tournamentId	Id of closing tournament	String

2.6.6 Notify definitive closure of tournament

Topic name	tournament.lifecycle.closed

Pub	Sub
TournamentService	BattleService

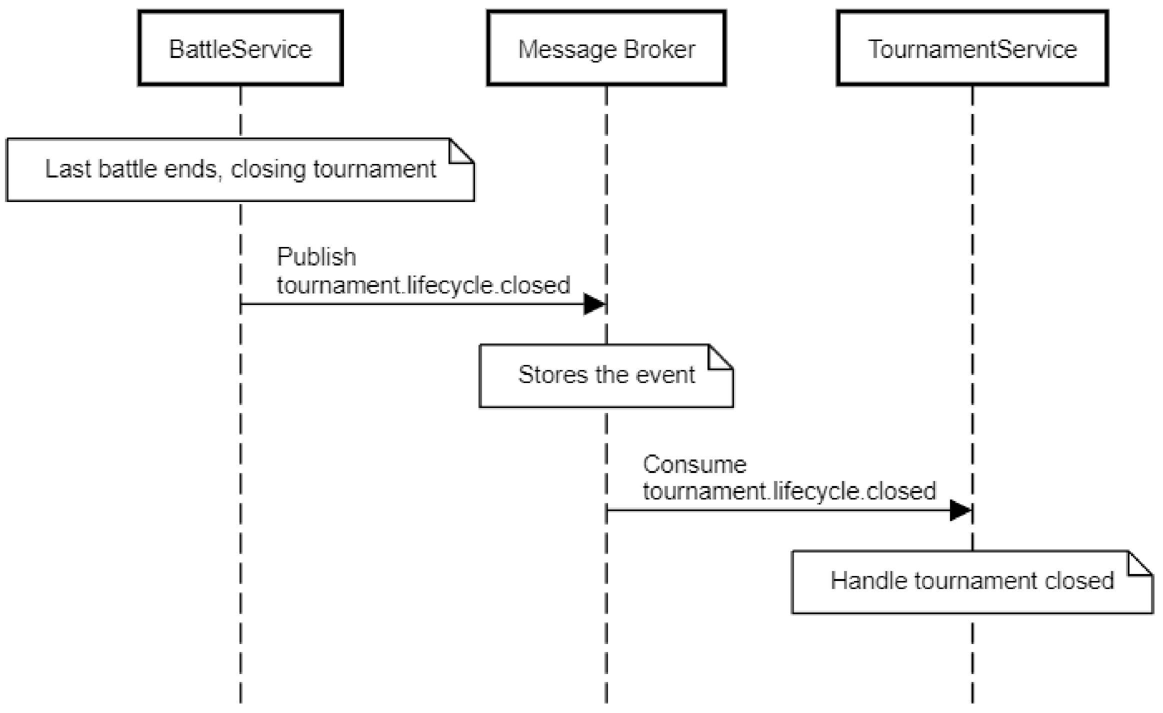


Figure 2.42

Message structure

Field	description	Type
tournamentId	Id of closed tournament	String

2.6.7 Notify about a badge evaluation request

Topic name	tournament.badge.request.evaluation

Pub	Sub
TournamentService	BadgeService

Message structure

Field	description	Type
badgeId	Id of the badge to evaluate	String
studentId	Id of the student	String
tournamentId	Id of the playing tournament	String

The badge service will consume the message and evaluate the badge, it will then publish on another topic (tournament.badge.outcome.win) if the badge was won

2.6.8 Notify about a badge win

Topic name	tournament.badge.outcome.win

Pub	Sub
BadgeService	UserService

Message structure

Field	description	Type
badgeId	Id of the badge to evaluate	String
studentId	Id of the student	String

The user service will call POST /tournament/student/badge from the UserDataTier interface to update the student badges adding a new one.

2.6.9 Notify about the creation of a badge

Topic name	badge.creation

Pub	Sub
BadgeService	UserService

Message structure

Field	description	Type
badgeId	Id of the new badge	String
educatorId	Id of the badge creator	String

The user service will consume the message and call the POST /educator/badge in the user data tier interface.

2.6.10 Notify about the deletion of a badge

Topic name	badge.deletion

Pub	Sub
BadgeService	UserService

Message structure

Field	description	Type
badgeId	Id of the deleted badge	String

The user service will consume the message and call the DELETE / badge in the user data tier interface.

2.6.11 Notify about the battle score

Topic name	battle.student.score

Pub	Sub
BattleService	TournamentService

Message structure

Field	description	Type
studentId	Id of the student	Long
score	Score of the student	Int
tournamentId	Tournament in which the battle was played	Long

The tournament service will consume the message and use the PUT /student/score API call to update the database about the score of the student in the tournament.

2.6.12 Notify about creation of a new user

Topic name	user.creation

Pub	Sub
AuthenticationService	UserService TournamentService BattleService

<u>Message structure</u>

Field	description	Type
email	Mail of the new user	String
type	Type of user	String
name	Name of the user	String
surname	Surname of the user	String
nickname	Nickname of the user	String
userId	Id of the new user	Long

2.6.13 Notify about email send request:

Publish on this topic the mail that want to send to a user.

Topic name	email.sending.request

Pub	Sub
TournamentService BattleService AuthenticationService	UserService

<u>Message structure</u>

Field	description	Type
from	Id of the sender	Long
to	Id of the recipient	Long
subject	Title of the email	String
body	Email content	String

2.6.14 Notify about the creation of a new battle

Topic name	battle.creation

Pub	Sub
BattleService	UserService, TimeServer

<u>Message structure</u>

Field	description	Type
BattletId	Id of the new battle	Long
educatorId	Id of the creator	Long
registrationDeadline	Time of tournament beginning	String

The user service will consume the message and call POST /battle in the user data tier interface.

2.6.15 Notify about a student quitting a battle

Topic name	battle.student.join

Pub	Sub
BattleService	UserService

<u>Message structure</u>

Field	description	Type
studentId	Id of the student	Long
battleId	Id of the battle	Long

The user service, upon receiving of the message, will update its knowledge about the student by adding a battle in which the user participated. It will use POST /battle/student.

2.6.16 Notify about student quitting a battle

Topic name	battle.student.quit

Pub	Sub
BattleService	UserService

<u>Message structure</u>

Field	description	Type
studentId	Id of the student	Long
battleId	Id of the battle	Long

The user service, upon receiving of the message, will remove the student from the battle. It will use DELETE /battle/student

2.6.17 Notify about battle registration deadline expiration

Topic name	battle.lifecycle.battle

Pub	Sub
TimeServer	BattleServer

<u>Message structure</u>

Field	description	Type
BattleId	Id of battle	Long

2.6.18 Notify about battle submission deadline expiration

Topic name	battle.lifecycle.consolidation

Pub	Sub
TimeServer	BattleService

<u>Message structure</u>

Field	description	Type
battleId	Id of battle	Long

2.6.19 Notify definitive closure of battle

Topic name	battle.lifecycle.closed

Pub	Sub
BattleService	TournamentService

<u>Message structure</u>

Field	description	Type
battleId	Id of battle	Long

3. User interface design

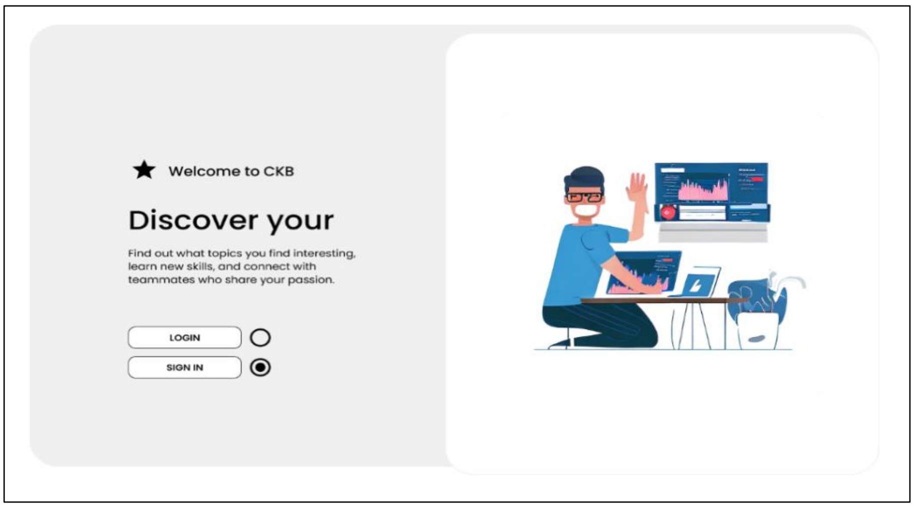

Welcome to CKB
Discover your
Find out what topics you find interesting, learn new skills, and connect with teammates who share your passion.
LOGIN
SIGN IN

Figure 3.1

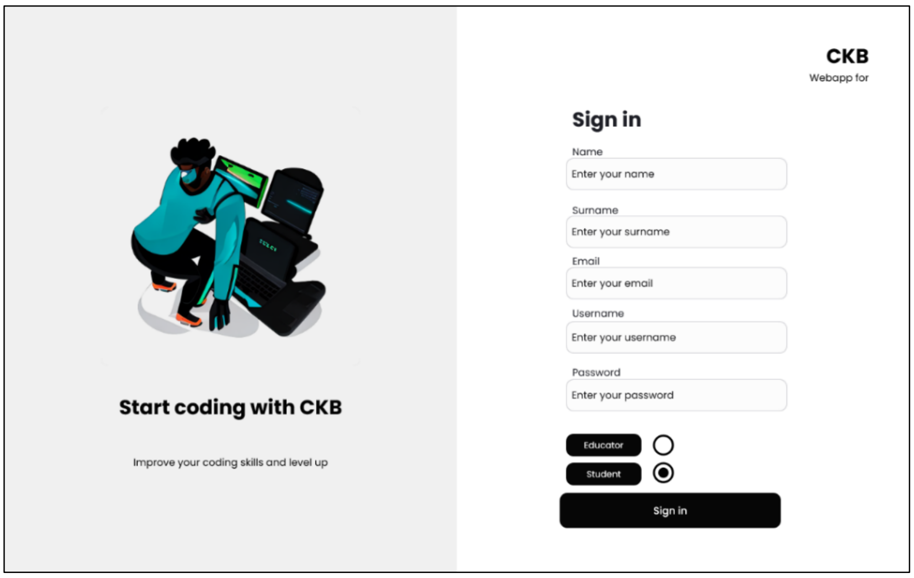


Figure 3.2

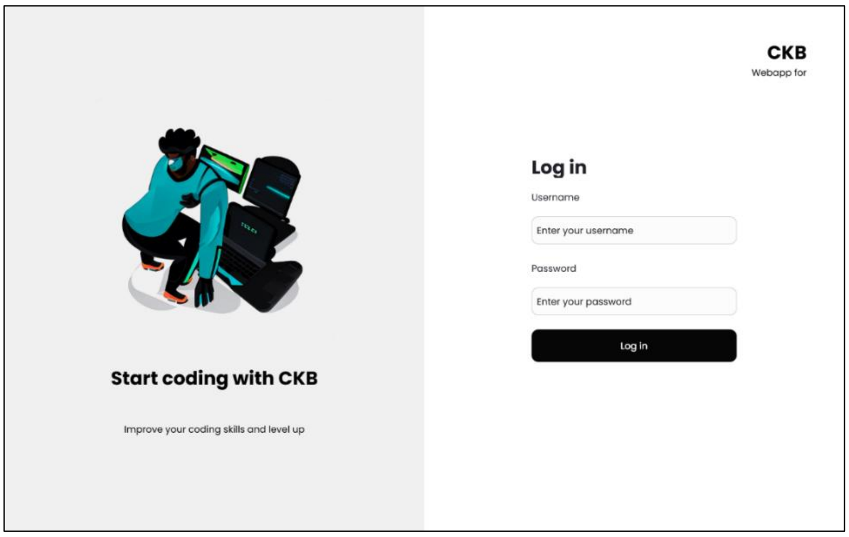


Figure 3.3

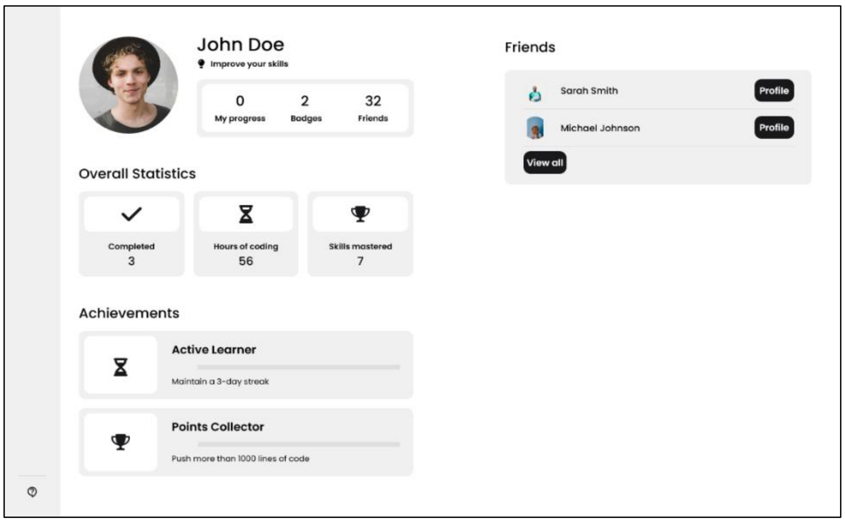


Figure 3.4

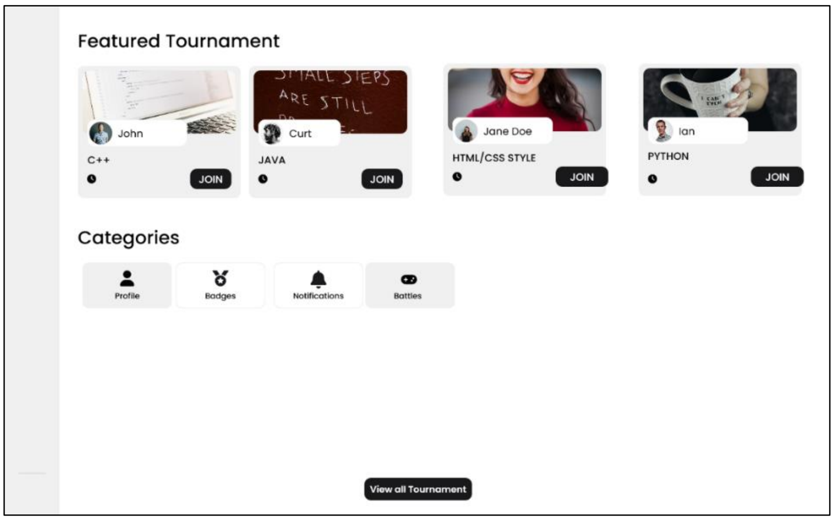


Figure 3.5

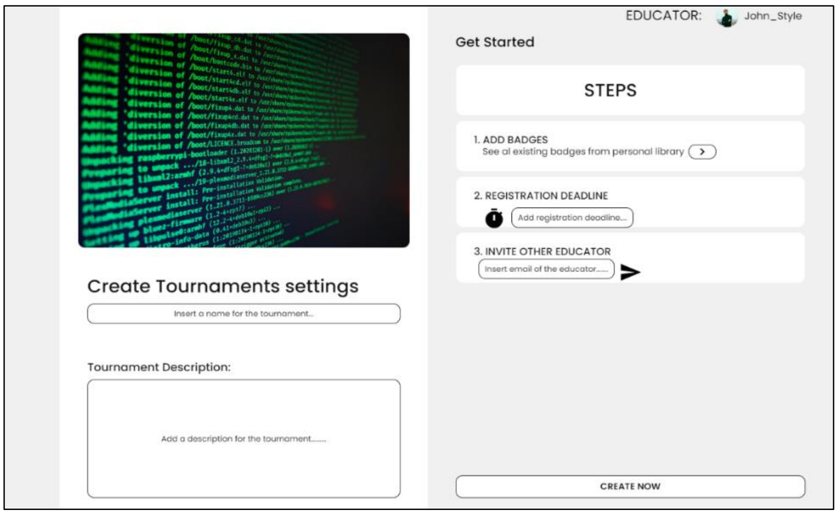

EDUCATOR: John_Style
Get Started
STEPS
1. ADD BADGES
See al existing badges from personal library
2. REGISTRATION DEADLINE
Add registration deadline...
3. INVITE OTHER EDUCATOR
Insert email of the educator......
Create Tournaments settings
Insert a name for the tournament...
Tournament Description:
Add a description for the tournament.......
CREATE NOW

Figure 3.6

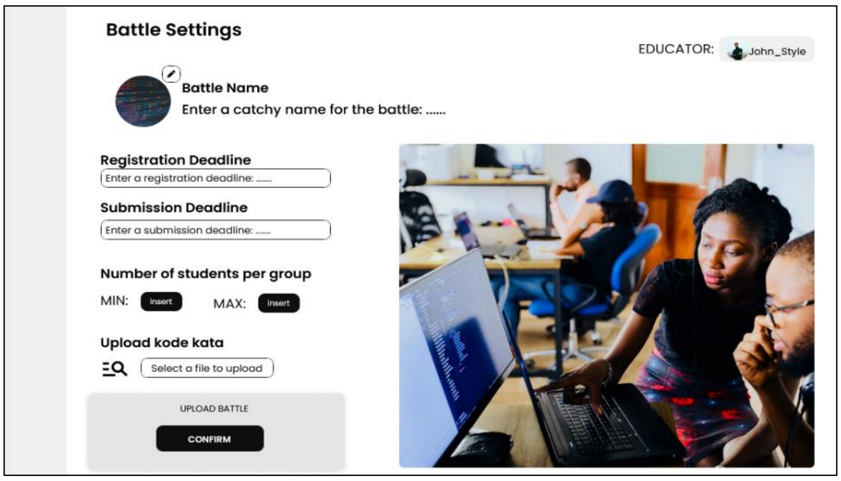
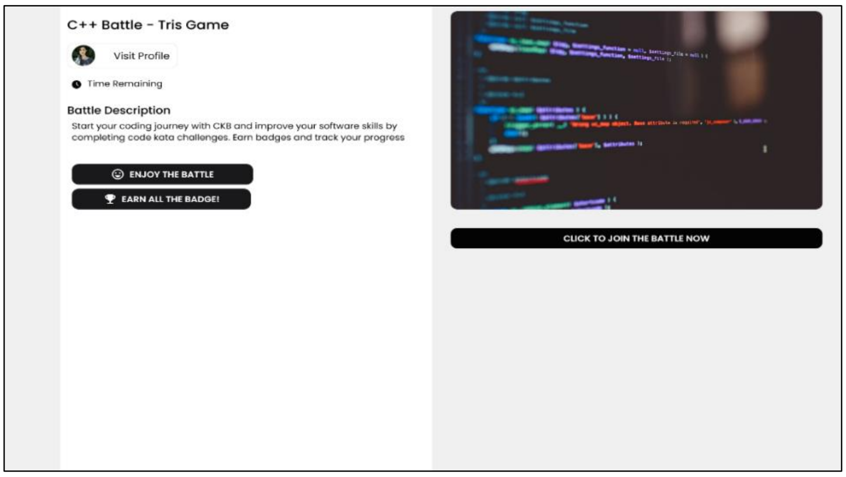


Figure 3.7



Start your coding journey with CKB and improve your software skills by completing code kata challenges. Earn badges and track your progress

Figure 3.8

4. Requirement traceability

	Client	API Gateway	Tournament Service	Battle Service	User Service	Badge Service	Authentication Service	Kafka	Time Server
[R1]	✖	✖					✖	✖	
[R2]							✖		
[R3]	✖	✖	✖						
[R4]			✖	✖				✖	✖
[R5]	✖	✖	✖			✖		✖	
[R6]	✖	✖	✖						
[R7]	✖	✖	✖					✖	
[R8]	✖	✖		✖				✖	
[R9]	✖	✖		✖				✖	
[R10]	✖	✖		✖					
[R11]				✖				✖	✖
[R12]				✖					
[R13]				✖					
[R14]				✖					
[R15]				✖					
[R16]				✖				✖	
[R17]			✖	✖	✖			✖	✖
[R18]	✖	✖		✖				✖	
[R19]				✖					
[R20]	✖	✖				✖		✖	
[R21]	✖	✖				✖		✖	
[R22]	✖	✖		✖				✖	
[R23]	✖	✖		✖					
[R24]			✖	✖	✖			✖	✖

5. Implementation, Integration, and test plan

5.1 Implementation Plan

The system should be implemented following a bottom-up strategy. Considering the architectural choices made, the following order of component development is suggested:

1. Microservices and their respective data tiers
2. Kafka Message Communication
3. API Gateway
4. Client and User Interface

Each microservice is responsible for a unique macro-function, which can be related to tournaments, battles, badges, users or authentication. It is natural, then, to start developing them as separated atoms of the system in order also to test them separately. Then Kafka Message Communication System can be implemented and integrated with the various services. Since microservices use some feature provided by external "software", during the integration and testing phase they need to be simulated by stubs. The third step regards the API Gateway which is responsible for handling the various requests coming from clients and forwarding them to the right service. The API Gateway also provides load balancing features that are performed separately for each microservice. The last component to be developed will be the client.

5.2 Component Integration and Testing

Each microservice must be first tested separately using ad hoc unit tests.

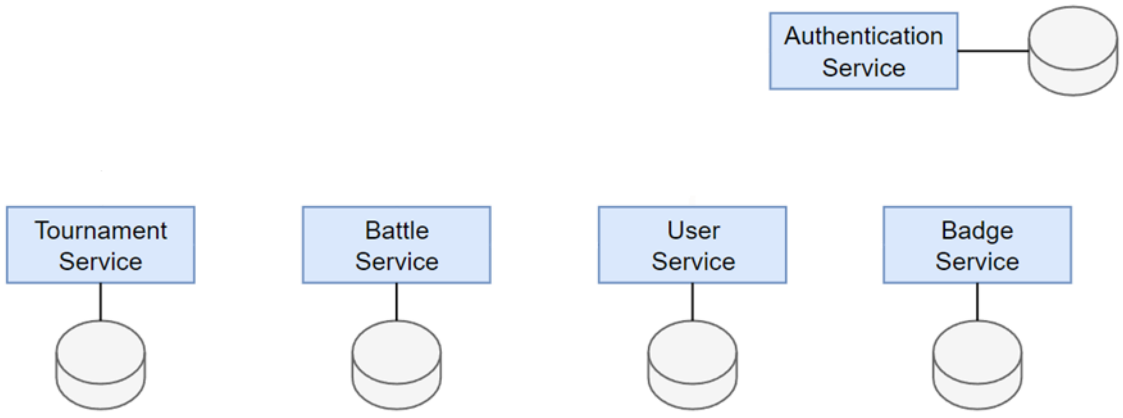


Figure 5.1

When microservices are ready and appropriately tested, Kafka Message Communication System can be integrated, providing producers and consumer development. It is then possible to test this new feature to make sure that changes in each service are correctly propagated to the other ones.

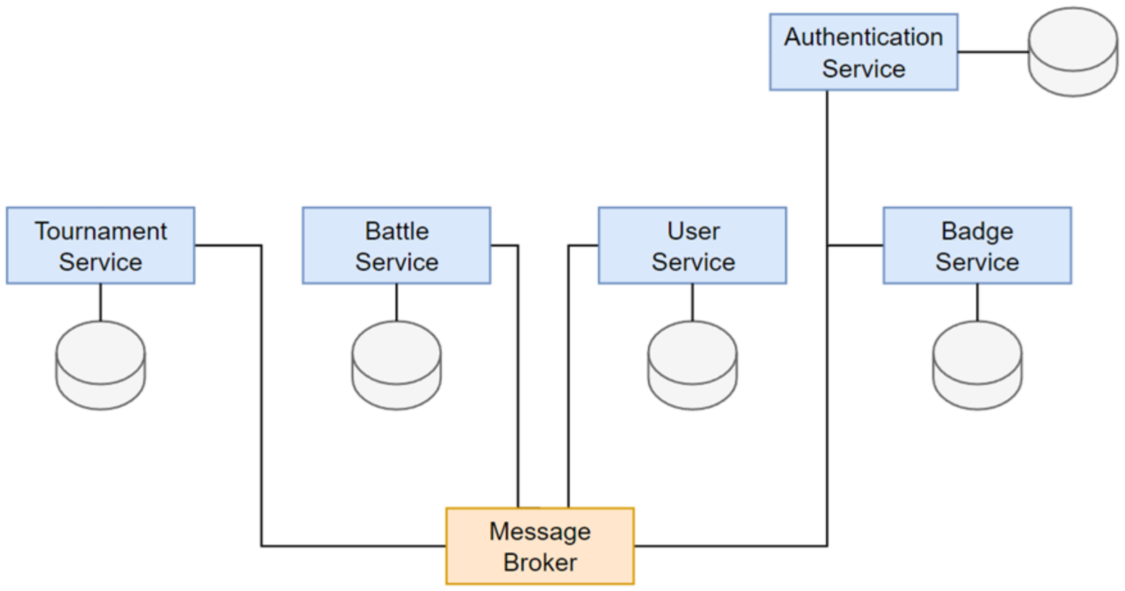


Figure 5.2

If everything is working in the right way, the API Gateway can be integrated and tested to make sure it handles each type of request managing the amount of traffic.

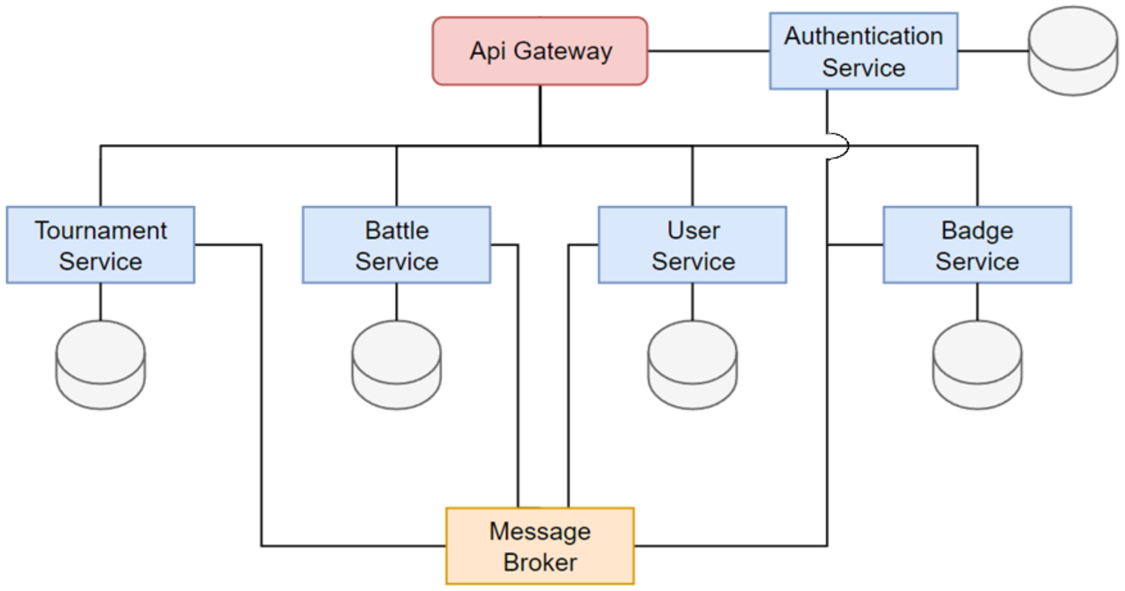


Figure 5.3

Finally, the client can be integrated with the other components and the entire system must be tested by functional, performance, load and stress tests.

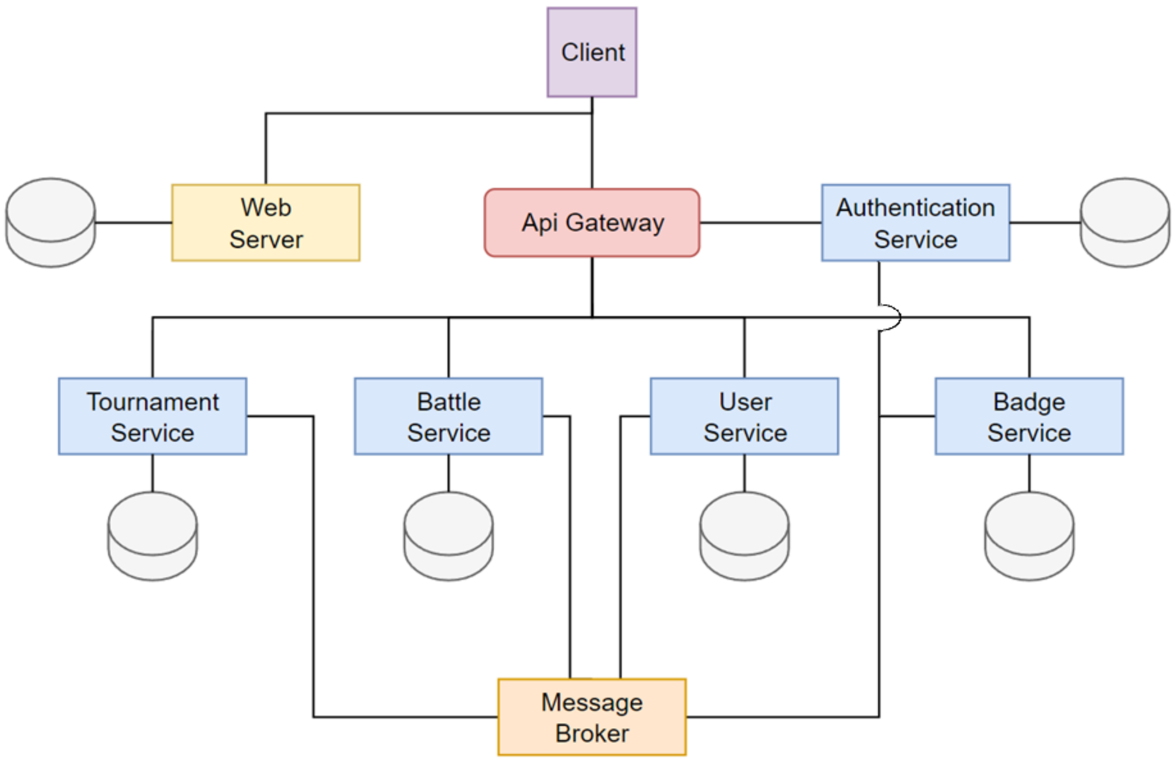

Client
Web Server
Api Gateway
Authentication Service
Tournament Service
Battle Service
User Service
Badge Service
Message Broker

Figure 5.4

6. Time Spent

Section	Feraboli	Filippini	Lucca
1	0	2	2.5
2	19,5	20	30.5
3	0	0	0
4	0,5	1.5	0
5	0	2	0

7. References

- Sequence diagrams made with sequencediagram.org
- Component diagrams made with StarUML
- Database's diagrams made with lucidcharts.com
- Interface mockups made with uizard.io