POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CodeKataBattle - Design Document

Software Engineering 2 Project
MSc Computer Science and Engineering

Authors: **Michele Bersani, Paolo Chiappini, Andrea Fraschini**
Reference Professor: **Prof. Matteo G. Rossi**

Student IDs: **10707192, 10743051, 10725610**
Academic Year: **2023-24**
Revision: **v1.0 07/01/24**
GitHub repository: **https://github.com/paolo-chiappini/BersaniChiappiniFraschini**

Contents

1. Introduction

1.1. Purpose

CodeKataBattle (CKB for short) is a web-based platform designed for students and educators. Its primary objective is to improve students' coding abilities by offering an environment for collaborative training on coding challenges.

Educators can use the CKB platform to challenge students by organizing tournaments and code kata battles, setting registration and submission deadlines, as well as scoring the students' solutions, thus creating a structured learning environment.

A key benefit of utilizing the platform lies in its automatic testing tools, which provide valuable support to both students and educators during the learning journey. These automation tools allow students to receive automated evaluations for their solutions by integrating with GitHub. This streamlined evaluation process also assists educators in their assessment, enhancing efficiency and objectivity.

The platform also offers gamification badges, which are set up by educators and automatically assigned by the platform to students. These badges serve to motivate students using the platform by providing a fun addition to the learning process.

In summary, CKB provides a comprehensive environment for students to practice and improve their coding skills through code challenges enriched with gamification aspects. The platform also simplifies the work of educators by making the organization of educational activities easier.

1.3. Scope

The platform enables educators to create challenges by first organizing tournaments. Tournaments provide a context for educators to host kata battles, along with colleagues who have been invited to manage the tournament. When creating a tournament, educators provide a deadline for the subscription of students. Throughout tournaments, educators have the opportunity to create and set gamification badges that will be awarded to students upon the tournament's conclusion. Educators can create battles within tournaments by setting up a skeleton project, which includes a description, automation scripts, and tests. They can also define registration and submission deadlines, minimum and maximum sizes for student groups, and scoring parameters. These scoring parameters may include the requirement for manual evaluation on behalf of the educators or the utilization of static analysis tools capable of assessing specific aspects of the code written by students.

Students will be allowed to subscribe to tournaments and form groups to enroll in battles within the given deadlines and parameters set by educators. After the enrollment period concludes, the platform will automatically generate a GitHub repository for the respective battle and send an invitation link to the students who have enrolled. Subscribing to a tournament enables students to receive notifications related to changes in the tournament’s status, such as new upcoming battles, rank updates, and tournament conclusion. Students will also be notified when a new tournament is created. The platform provides students with an assessment of their work, assigning points and a ranking for each battle. These points will contribute to the calculation of the overall score for each student and determine their ranking in tournaments.

1.3. Definitions, abbreviations, acronyms

1.3.1 Definitions

- **Static code analysis**: software verification activity that analyzes source code for quality, reliability, and security without executing the code.
- **Software Platform**: set of software and hardware used to provide a certain service or to host an application.
- **Users**: a collective noun that encompasses both student and educator accounts. It refers to the people using the services offered by the platform.
- **Authenticated users**: users that have completed the login process, and therefore have been authenticated in the eyes of the CKB platform.
- **Code kata**: code katas are programming exercises in a language of choice aimed at helping students learn and improve at coding.
- **Solution**: refers to any code created with the explicit purpose of addressing a specific problem or challenge. In this specific context, solutions are the pieces of code submitted by students to be evaluated in battles.
- **(Gamification) Badges**: elements in the form of rewards that represent the achievements of individual students during tournaments.
- **RESTful API:** API that follows the REST architecture.
- **Authentication token:** an authentication token is a set of characters used to identify and authenticate a user (or set of users).
- **Automatic workflow:** sequence of automated actions.
- **Back-end:** server-side of a web application that handles all the requests from the client. It contains all or part of the business logic and it has access to the main database of the application.
- **Front-end:** client-side of a web application whose main purpose is to provide a graphical interface to the user. It achieves this by sending requests to the back-end to receive and send data.
- **Sticky-session:** Load balancing policy in which, for the whole duration of a session, the requests of a client are handled by the same server. By employing this policy, session data can be stored locally on the server, removing the need for frequent synchronization and enhancing performance. In its basic implementation there is a trade-off with service availability. If a server fails, the session data stored on that server is lost, potentially leading to a temporary disruption in the user's

session. Session replication mechanisms and failovers can be implemented to mitigate this.

1.3.2 Abbreviations

- **CKB:** CodeKataBattle platform.
- **CDN:** Content Delivery Network.
- **The platform,** or simply **Platform**: refers to the CKB platform.
- **ECA**: External Code Analyzers.
- **IaaS:** stands for *Infrastructure as-a-Service*. It's a cloud computing model that provides virtualized computing resources over the internet.
- **[UCn]:** n-th use case.
- **[Rn]:** n-th requirement.

1.3.3 Acronyms

- **API**: Application Programming Interface.
- **GUI**: Graphical User Interface.
- **HTTP:** HyperText Transfer Protocol.
- **HTTPS:** HyperText Transfer Protocol Secure.
- **JSON:** JavaScript Object Notation.
- **REST**: Representational State Transfer.

2. Architectural Design

As stated at the beginning of the *RASD* document, we have decided to implement Code Kata Battle as a classic Client-Server Web Application. We made this choice because of all the benefits that this type of solution provides and because of how well they satisfy many of the requirements that have been underlying in the *RASD* document. The main advantages are:

- **Cross-Platform Compatibility**: Web applications are not tied to a specific operating system and only require a compatible web browser to be used. This means that users can use the application regardless of whether they are using Windows, macOS, Linux, or any other operating system. This also allows the application to be potentially used from mobile devices.
- **Scalability**: Web applications are well-suited for handling a large number of concurrent users. As the popularity of the application grows, a web-based platform allows for easier scalability by adding server resources or employing cloud services to meet increasing demand. This ensures a smooth user experience, even during peak times.
- **Security**: Security protocols such as HTTPS, encryption, and secure authentication mechanisms can be implemented very easily to make Web Applications secure *by design*.
- **Cost-Effective Deployment**: Web applications can be deployed very easily and with relatively little cost using cloud services. Maintenance of the application is also hidden from the user (apart from UI changes) and can be performed quickly using techniques such as Continuous Integration.

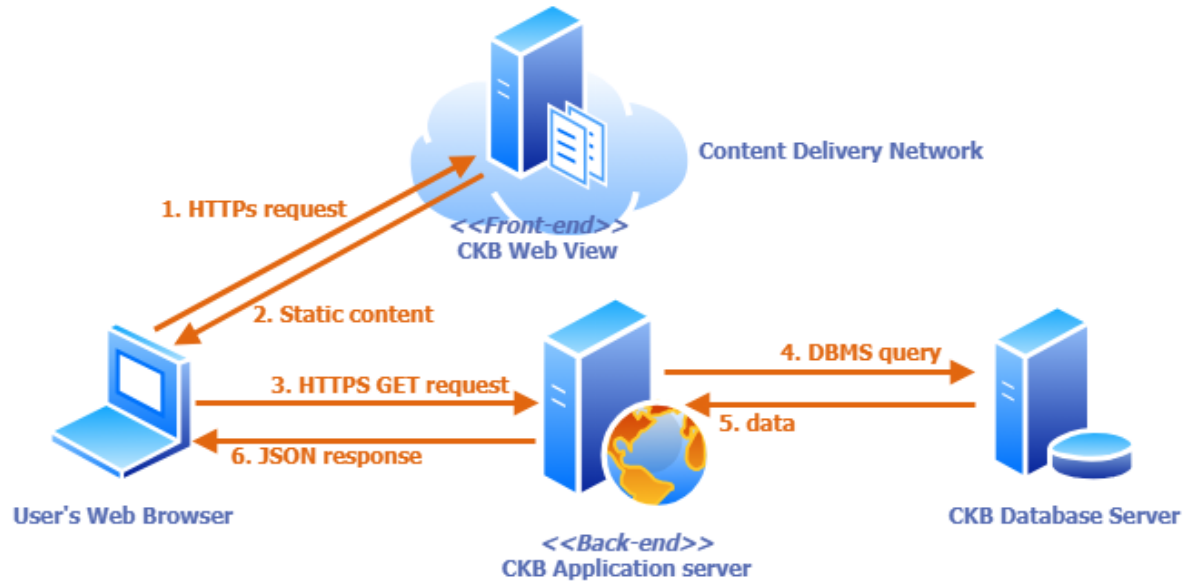
2.1. Web Application Architecture

The basic idea of the architecture is to decouple the back-end from the front-end in a very real way by deploying all the static content that makes up the front-end in a Content Delivery Network (CDN) to allow for faster loading times. This makes the interface load very quickly and improves the user experience significantly. We decided to use a wider geo-replication for the front-end since the contents that need to be distributed are static (HTML, CSS, JavaScript, and media files) and do not require additional operations to deal with sessions. Also, the load balancing is automatically managed by the DNS servers that

send requests to the geographically closest server without any additional logic. This means that it's much more cost-effective to replicate the front-end at a global scale rather than the back-end which is distributed much more sparsely to deal mostly with the high number of users rather than improve response times. The messages that the back-end exchanges with the front-end are, indeed, very small and the response times depend mostly on server-side CPU computations and database operations.

A brief example of how a user's web browser might interact with the system is described in the next section.

When the user's web browser connects to the application, the client application, as well as all its resources, are downloaded very quickly thanks to the geographical proximity of the server made possible by the CDN. After the initial download of the client application, subsequent requests made by the browser are handled by the back-end, operating on the application server. The entirety of the communication between the front-end and the back-end is done using HTTPS and occurs through an API that follows the RESTful architectural style.



Example of the sequence of requests that occurs when a user accesses the application

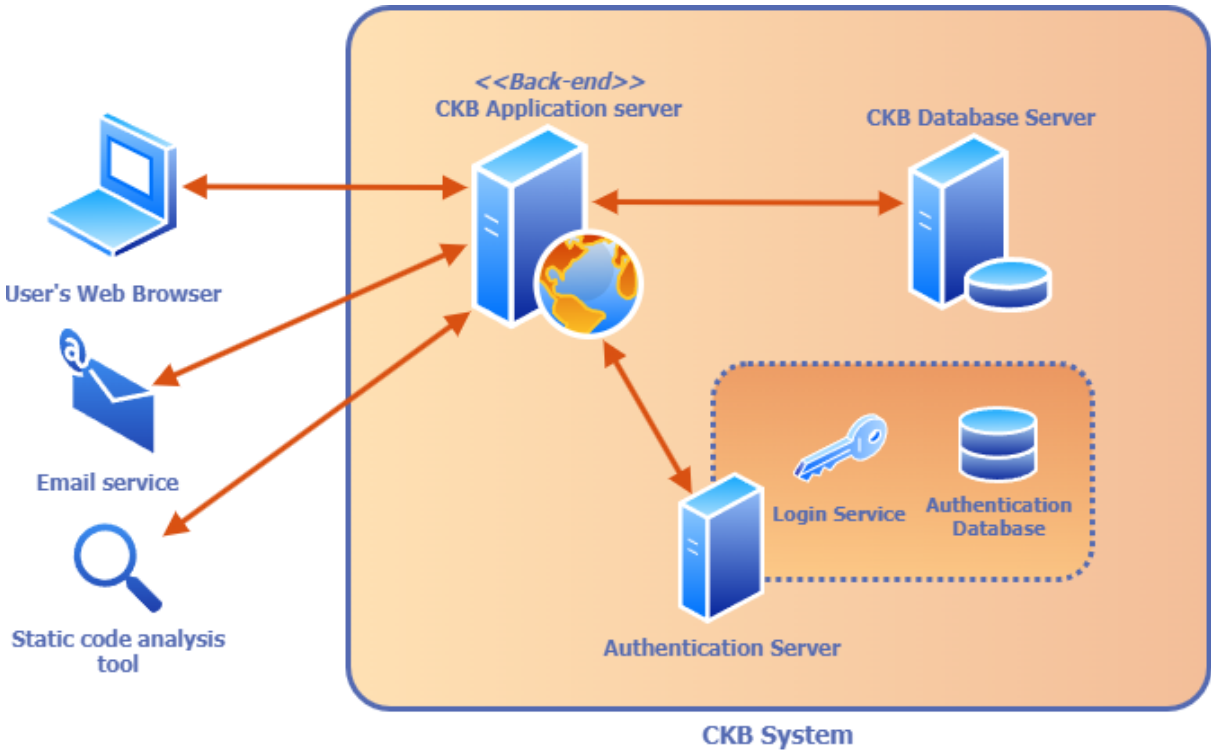
2.6.1 Microservices Architecture

We've chosen to separate user data and its management from the back-end by establishing a dedicated microservice equipped with its own database. This decision is primarily motivated by security considerations, aligned with the *separation of concerns*

principle. Beyond separating scalability domains, this approach allows the implementation of specialized security measures, such as incorporating additional authentication factors in later stages of the application's lifecycle. Moreover, it mitigates the risk associated with a potential breach, ensuring that an intrusion into the application doesn't necessarily compromise the entire authentication service and users' sensitive data.

The interaction between the CKB application and the authentication service mirrors the communication with the front-end but, in this case, the roles are flipped in the sense that the authentication server provides a RESTful API to which the CKB application sends the requests.

CKB also utilizes email services and external tools for static code analysis. We've decided to incorporate them into the microservices category despite their exclusion from the development process.

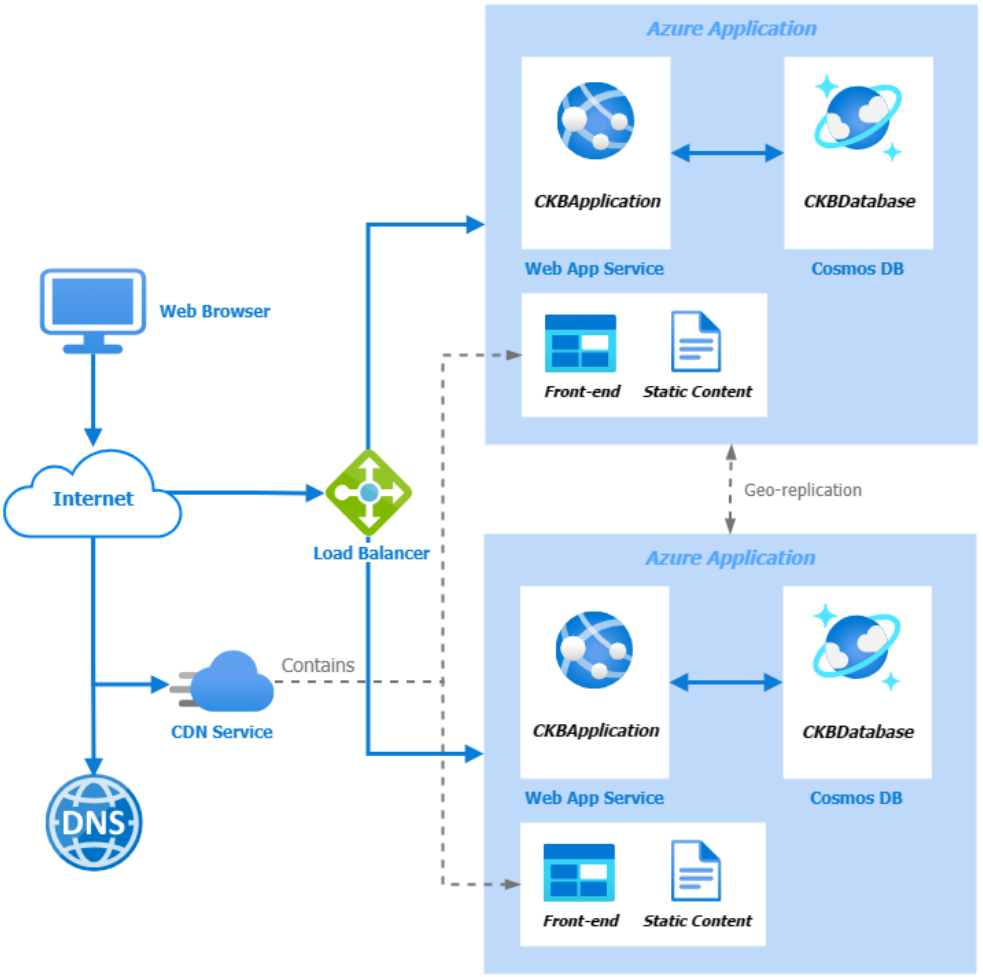


In this figure, the microservices used by Code Kata Battle are represented as well as the domains in which the systems operate. The CDN is not represented to aid clarity.

2.6.2 Cloud Architecture

To easily deploy, maintain, and scale the application, we've decided to use *IaaS* cloud services. More precisely, we've chosen **Microsoft Azure** as our hosting platform since it allows us to easily implement everything we've defined in the previous sections.

The application is deployed using Azure's *Web App Service* with the CDN service for static content. The main application database is also managed by the Web App service using a *Cosmo DB* instance which can easily be globally distributed for better scalability. By doing this, every aspect of deployment and communication between the front-end, back-end, and database is handled by Azure and can easily be configured. Load Balancing parameters, for instance, are configured to strike a balance between *availability* and *performance* by employing a sticky session policy among web application nodes to mitigate coordination overheads. This is complemented by a centralized cache for storing session data on every node implemented using a key-value database. Consequently, if a server in a node fails, the load balancer can redirect requests to another server within the same node, thus maintaining application availability. The sticky session policy prevents different nodes from simultaneously handling the same users, however, due to session data replication at the node level, availability is not significantly compromised, as explained earlier.



The authentication microservice is managed in the same way. The API of the microservice is also deployed using an Azure Web App coupled with a Cosmo DB instance for storing authentication data.

2.2. Components

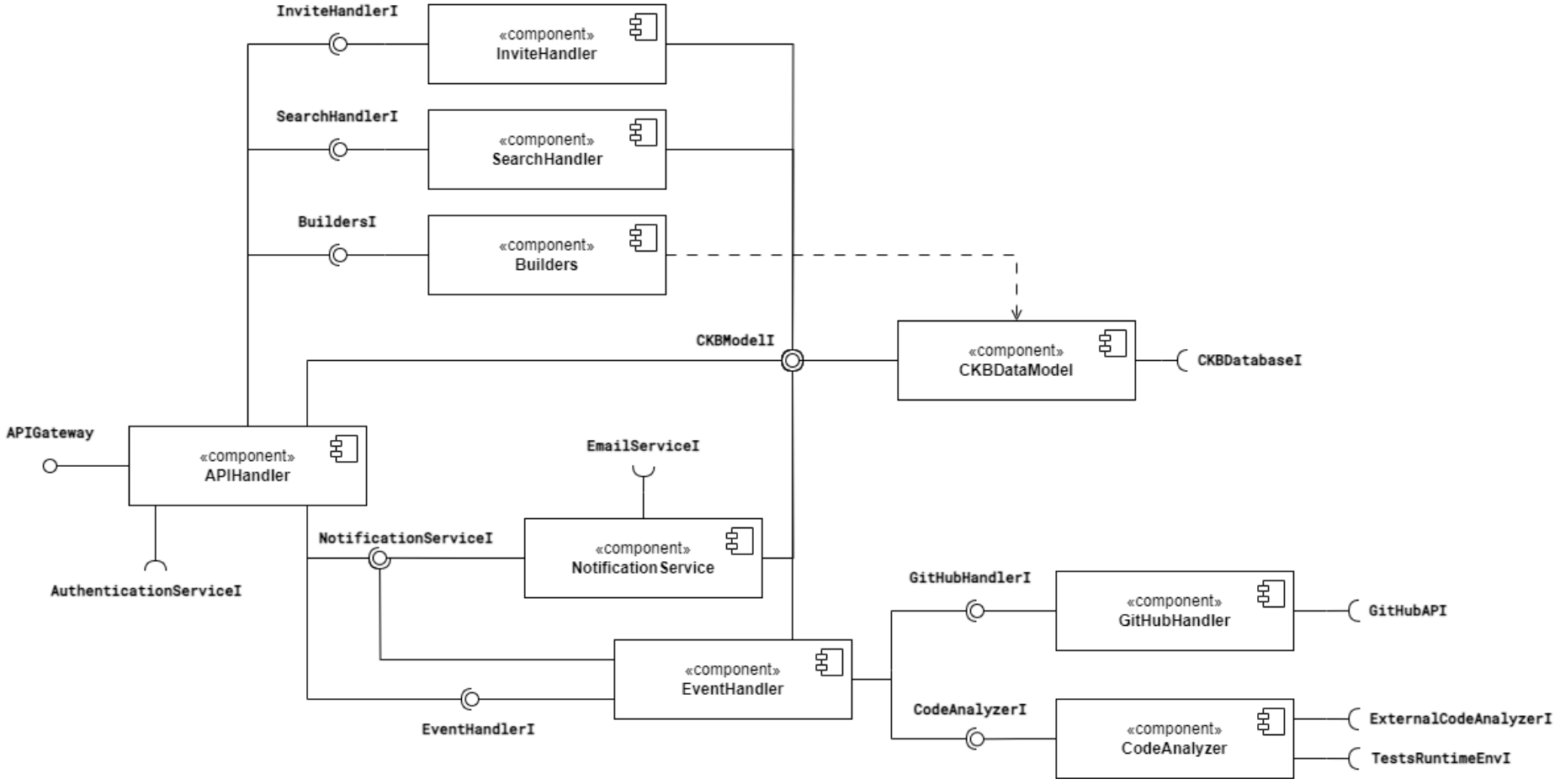
Here are described the main software components (in **bold**) that make up the application and their respective functions. Note that some components are made up of sub-components. For the definition of the components, we adopted a viewpoint that is a blend of high and low level to better guide the development process, already highlighting the logical division to some extent, even at the package level.

- **CKBWebView**: Front-end web application made up of HTML, CSS, and JavaScript code that is downloaded by the client, run by the web browser, and that displays the UI and allows the user to access the application and make requests to the application server.
- **CKBApplication**: Back-end web application. It contains the main business logic and all the subcomponents that are described in the next section.
    - **APIHandler**: Main application interface of the back-end.
        - **RequestsDispatcher**: dispatches the incoming HTTPS requests to the appropriate handler.
        - **RequestHandlers**: defines the methods responsible for handling the requests incoming from the API.
    - **CKBDataModel**: Defines the data structures that represent the entities of the application. It communicates with the Database to retrieve and update data.
        - **Battle**: Model of a battle in a tournament. Handles the registration of groups and the updating of the battle ranks.
        - **Tournament**: Model of a tournament. Handles the subscription of students, the permissions of managers, and the updating of the global score. When a tournament is closed, it also handles the awarding of badges.
        - **User**: Model of a generic user of the platform. It manages the details of the user (username, email) and their notifications and invites.
        - **Badge**: Model of a badge. It handles the check for the rules during the awarding process.

        - **Group**: Model of a group enrolled in a battle. It handles the registration of the scores for the provided solutions and the invited students.
        - **Notification**: Model of a notification.
        - **Invite**: Model of an invite.
    - **Builders**: Component responsible for handling the creation of new entities on behalf of the model (battles, tournaments, badges, users, etc.).
        - **UserAccountBuilder**
        - **TournamentBuilder**
        - **BattleBuilder**
        - **GroupBuilder**
        - **BadgeBuilder**
        - **BadgeRuleBuilder**
    - **NotificationService**: Component responsible for handling the creation and sending of notifications. It requires an EmailService to be able to send notifications via email.
    - **InviteHandler**: Handles the creation and sending of invites for both groups and tournament managers.
    - **SearchHandler**: Handles the search of battles, tournaments, and users. In this basic form it should only allow for text-search, but can be expanded to work with other filters.
    - **CodeAnalyzer**: Handles the automated analysis of the code when a push action occurs. It manages both the calls to the ECAs and the testing environment.
        - **ECARunner**: Component responsible for mediating the calls to the various ECAs and returning the results in the format used by the application.
        - **TestRunner**: Component responsible for mediating the calls to the testing environment and returning the results in the format used by the application.
    - **GitHubManager**: Component responsible for mediating the calls to the GitHubAPI, allowing for the creation of new repositories for battles and pulling changes when a push action is performed.
    - **EventHandler**: Component responsible for scheduling and handling events.

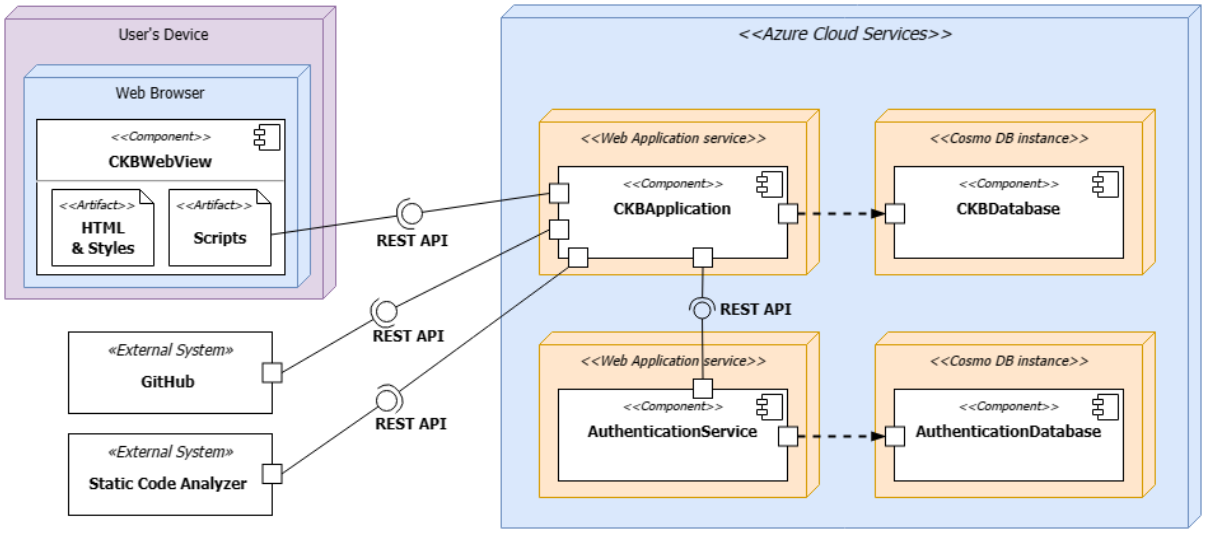
- **CKBDatabase**: DBMS that stores and manages all the data needed for CKB to work. All user data is stored here apart from passwords and all security-sensitive data which is managed by the authentication microservice.
- **AuthenticationService:**
    - **LoginRequestHandler**: Handles HTTPS requests from applications that use the login service.
    - **AuthDataModel**: Data structures used for authentication purposes. It also accesses the authentication database.
- **AuthenticationDatabase**: DBMS that stores and manages all the data needed for authentication.

The diagram below shows the main components of the **CBKApplication** and the interfaces shared between them. Further details about the definition of the interfaces can be found in Section 2.7.



2.3. Deployment view

The following diagram displays how the main components of the application are deployed **at runtime**. More precisely, it shows the locations of the components after a user has connected to the application and has downloaded the front-end from the Content Delivery Network.

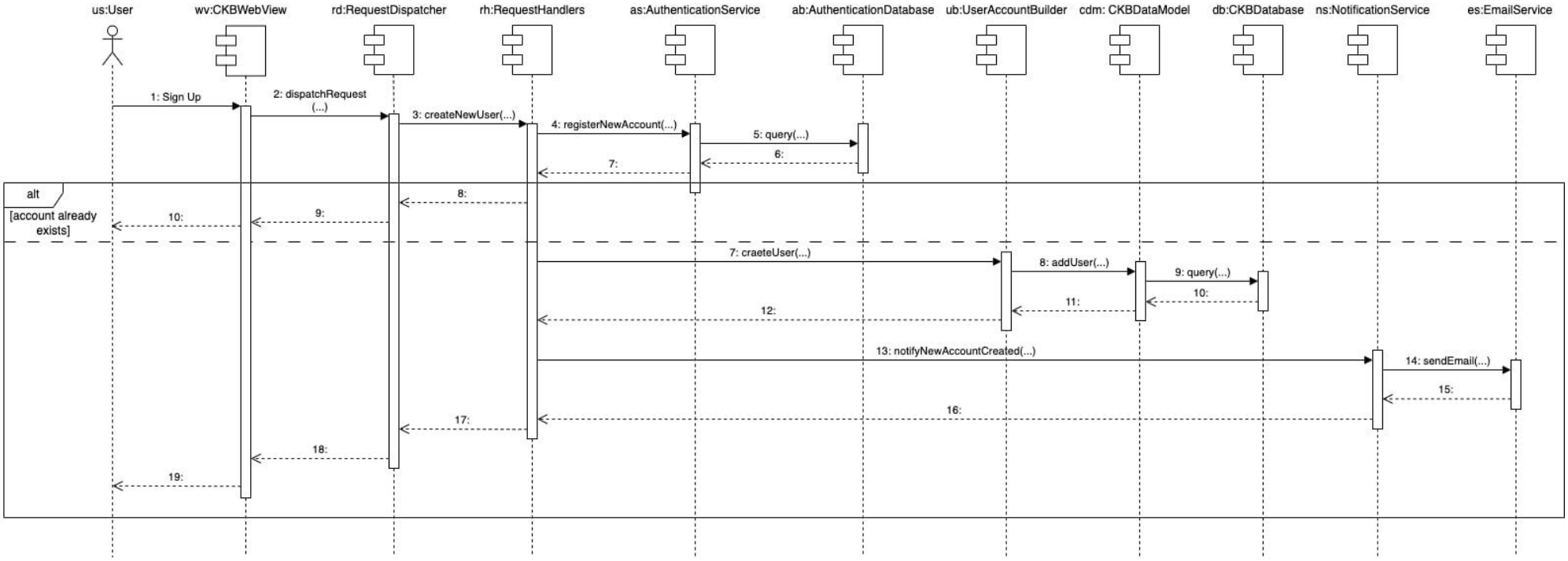


2.4. Runtime view

In the following section we provide a detailed view of the sequence of messages exchanged between components in relation to the already presented sequence diagrams for UCs in the RASD.

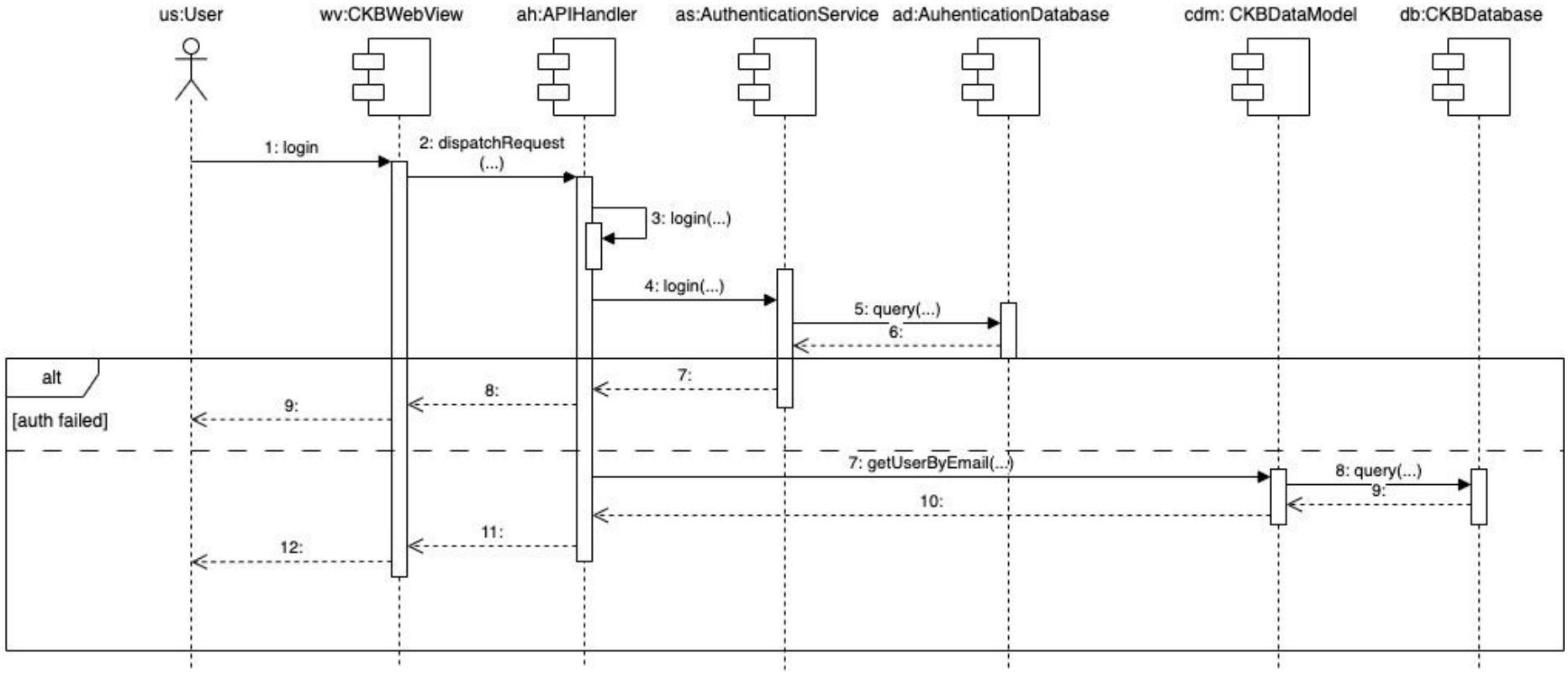
To improve the readability of diagrams, the interface methods used by messages do not show parameters, if any. All methods are to be considered consistent with those presented in Section 2.7. Also, to reduce the overall size of the diagrams the sub-components are not explicitly shown with the exception of some.

[UC1]: Creation of new CKB account



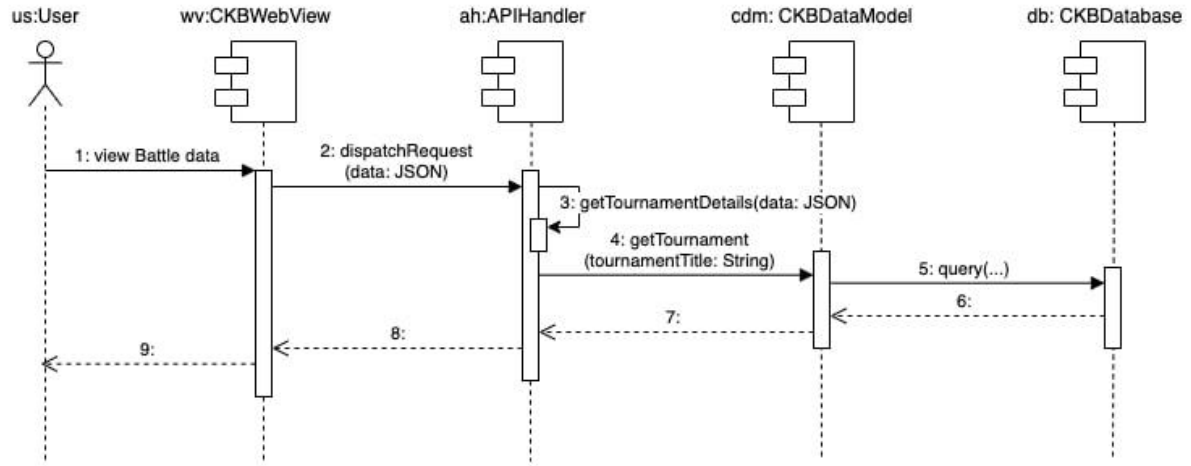
This sequence diagram represents the moment of creating an account within the CKB platform. The user sends the sign-up request which is managed by the RequestHandlers which has the purpose of coordinating the actions to be performed. First of all, the credentials are registered in the authentication microservice and then the user's information is saved within the platform database. All this, if the account does not already exist, ends with the sending of the email notification of the successful creation of the account.

[UC2]: Login



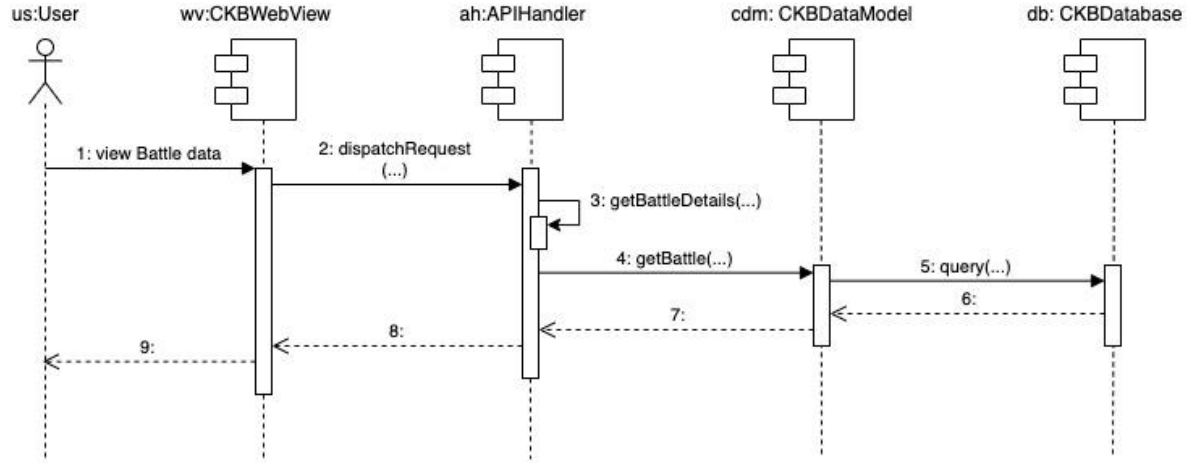
This sequence diagram describes how the login process of a user is handled by the system. The login request is managed by the APIHandler which delegates control of credentials to the authentication microservice. If the credentials are correct, the RequestHandler loads the user's information for storing the session; while if the credentials are incorrect, an error is returned to the user.

[UC3]: View tournament data



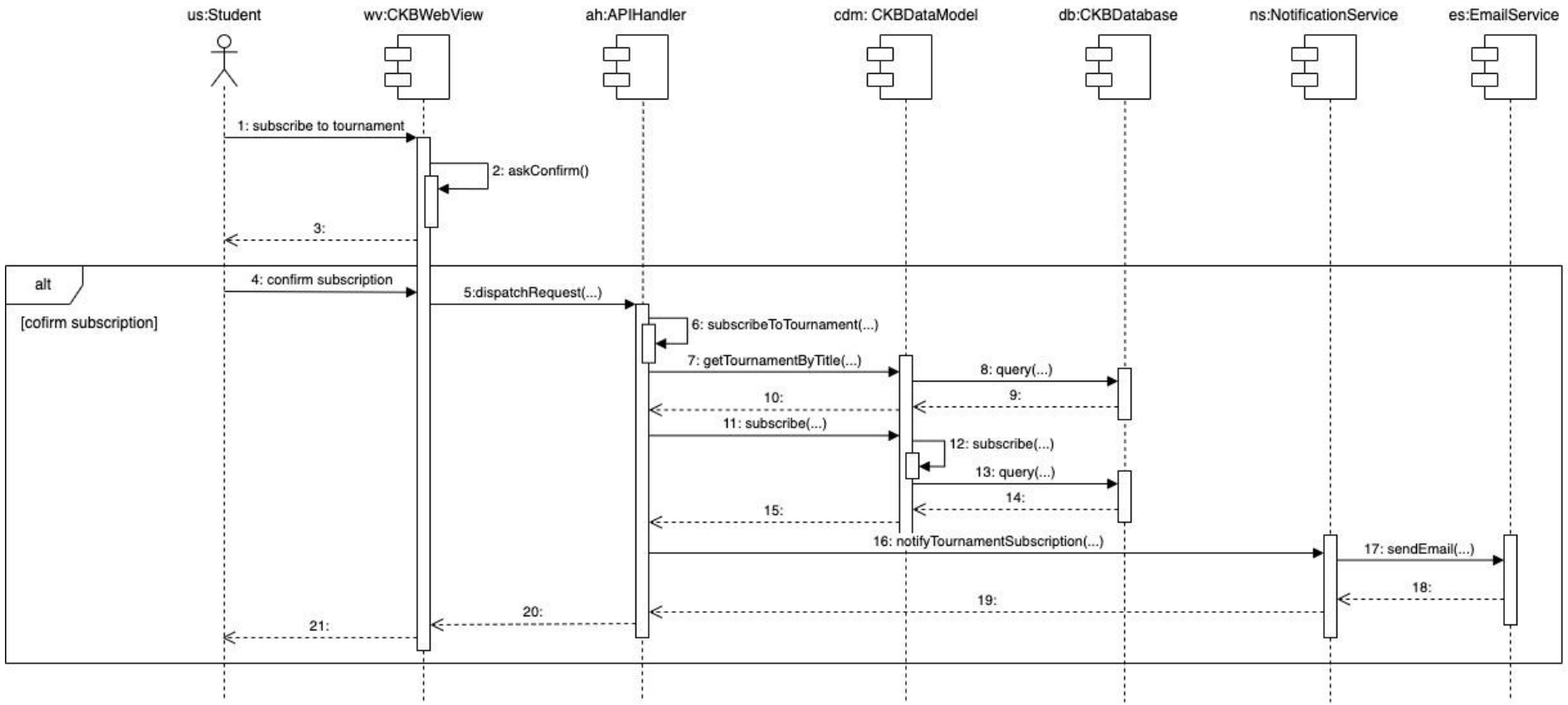
This sequence diagram describes the internal operations of the platform to display tournament data. The request is handled by the APIHandler which fetches the tournament data from the CKBDataModel which in turn queries the database. The data is then sent to the user.

[UC4]: View battle data



This sequence diagram describes the same operations as the previous sequence diagram, but in this case the server loads the data for the battle instead of the tournament.

[UC5]: Subscribe to tournament

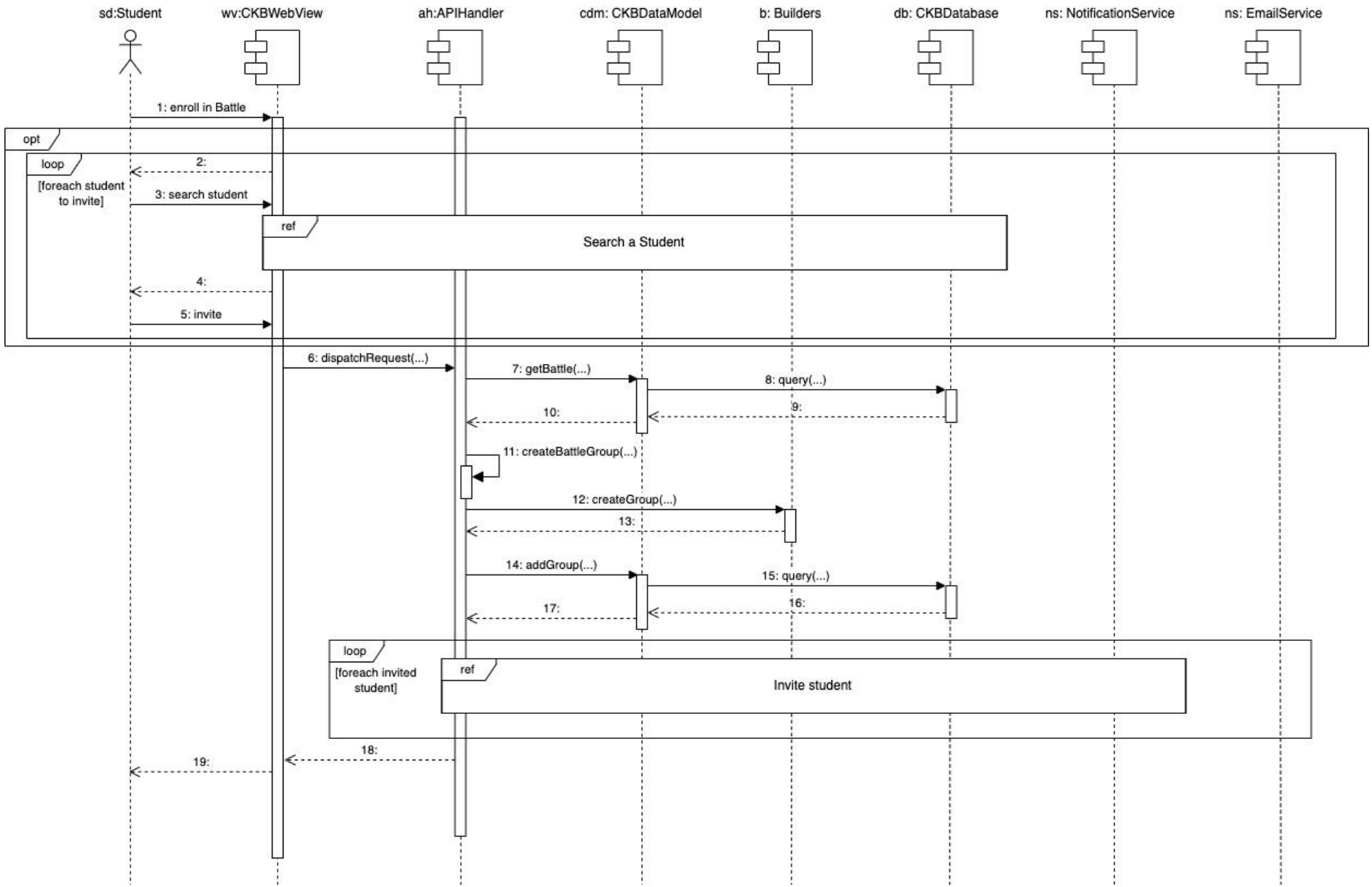


This sequence describes the registration of a student into a tournament. The registration request is sent to the APIHandler which calls the specific RequestHandler to load the data of the tournament and sends the registration order to the CKBDataModel which stores the action within the database. Finally, a notification of successful registration is created, stored and sent via email.

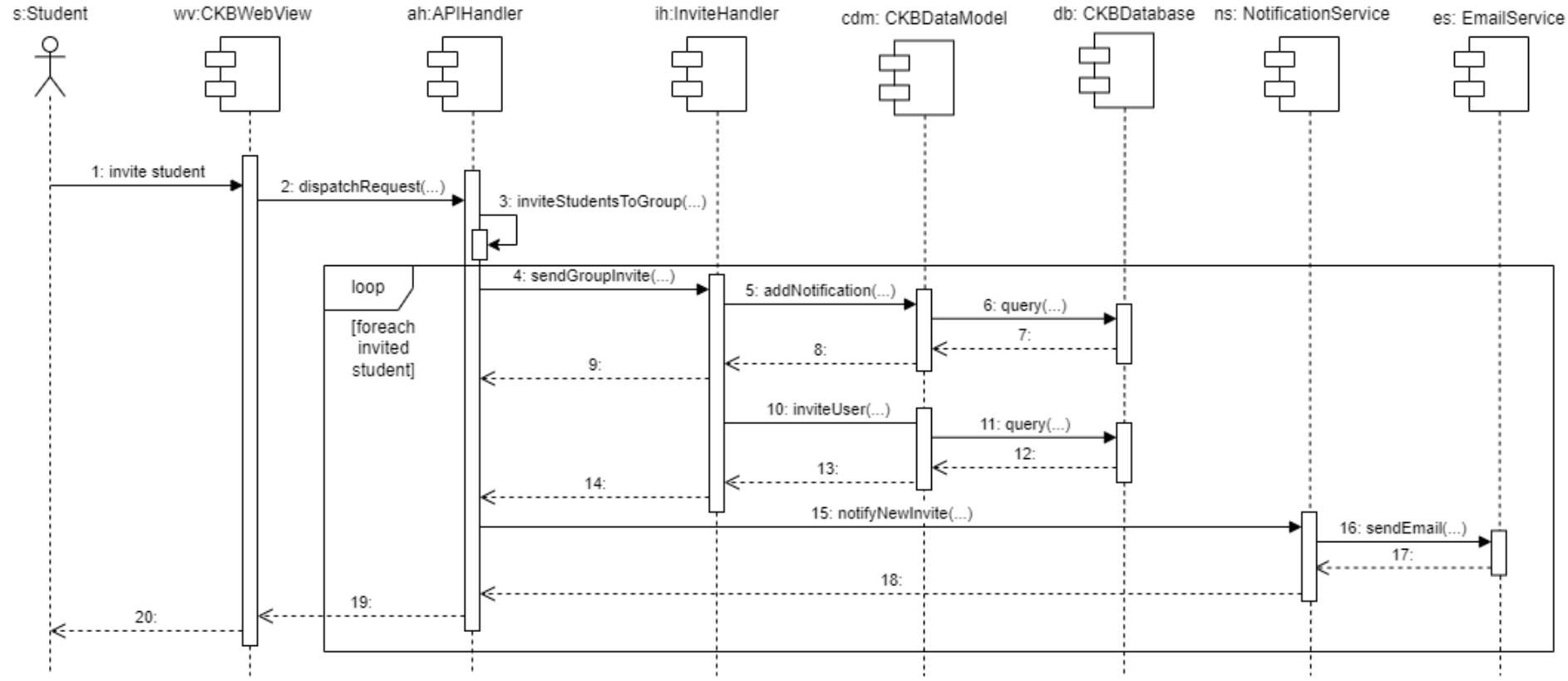
[UC6]: Enroll into battle

This sequence diagram describes the enrollment of a student into a battle. Notice that this operation is allowed if and only if the battle is still open for registrations.

The student, when registering for a battle, can decide whether to search for other students with whom to create a group or register by himself as a single student group. In any case, the registration request is sent to the APIHandler which creates the group using the Builders component and updates the battle information. If the student has expressed interest in inviting other students to the group, invitations are automatically sent.

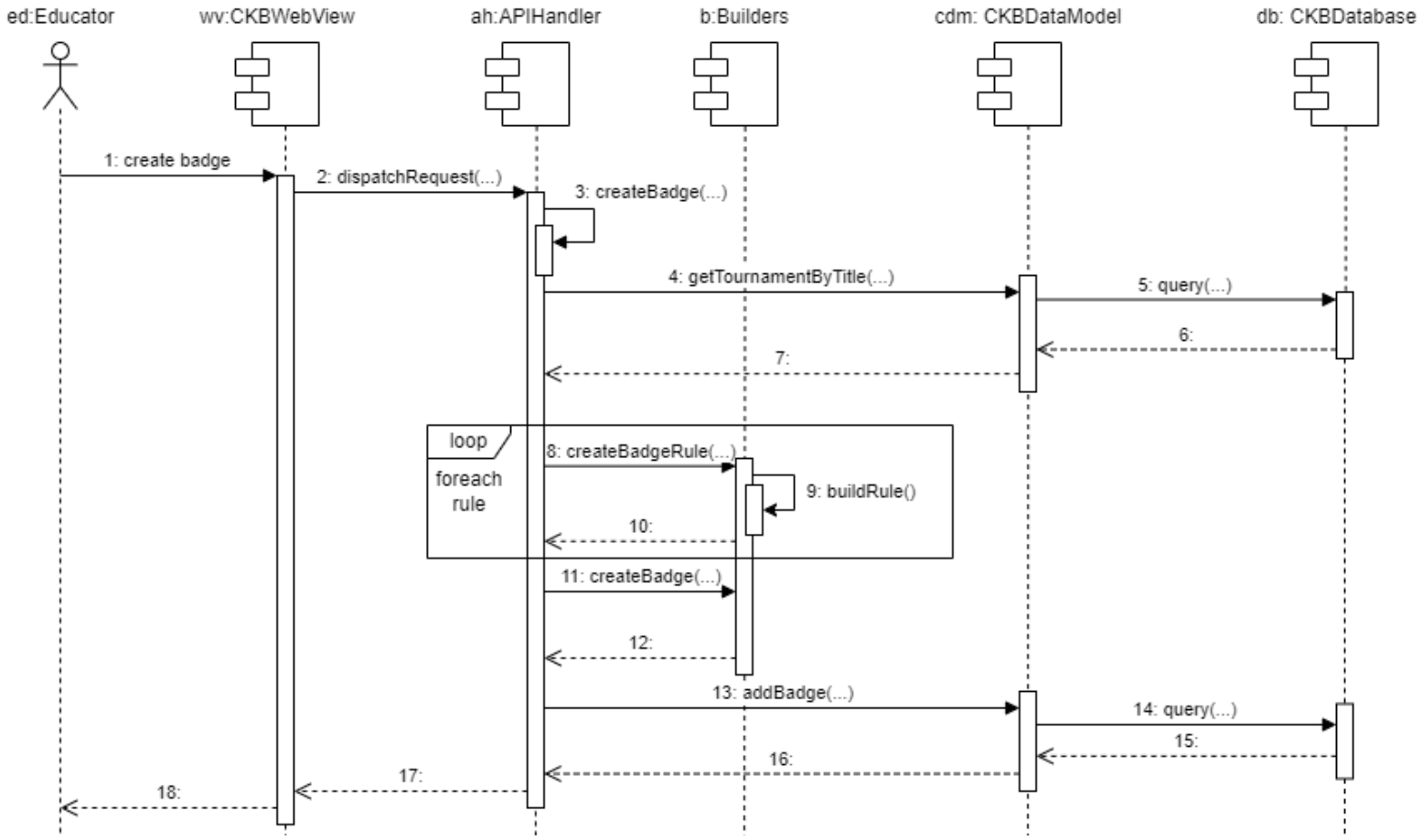

sd:Student
wv:CKBWebView
ah:APIHandler
cdm: CKBDataModel
b: Builders
db: CKBDatabase
ns: NotificationService
ns: EmailService
1: enroll in Battle
opt
loop
[foreach student to invite]
2:
3: search student
ref
Search a Student
4:
5: invite
6: dispatchRequest(...)
7: getBattle(...)
8: query(...)
9:
10:
11: createBattleGroup(...)
12: createGroup(...)
13:
14: addGroup(...)
15: query(...)
16:
17:
loop
[foreach invited student]
ref
Invite student
18:
19:

[UC7]: Invite students to group



This sequence diagram describes the operations that the Platform carries out when a student invites other students to join in their group for a battle. The request is sent to the APIHandler which dispatches it to the InviteHandler that, in turn, generates for each invited user a new invite and sends a notification using the email service. The notification and the invite are also stored for displaying it to the invited users when they access the application and to allow them to accept or deny the invite.

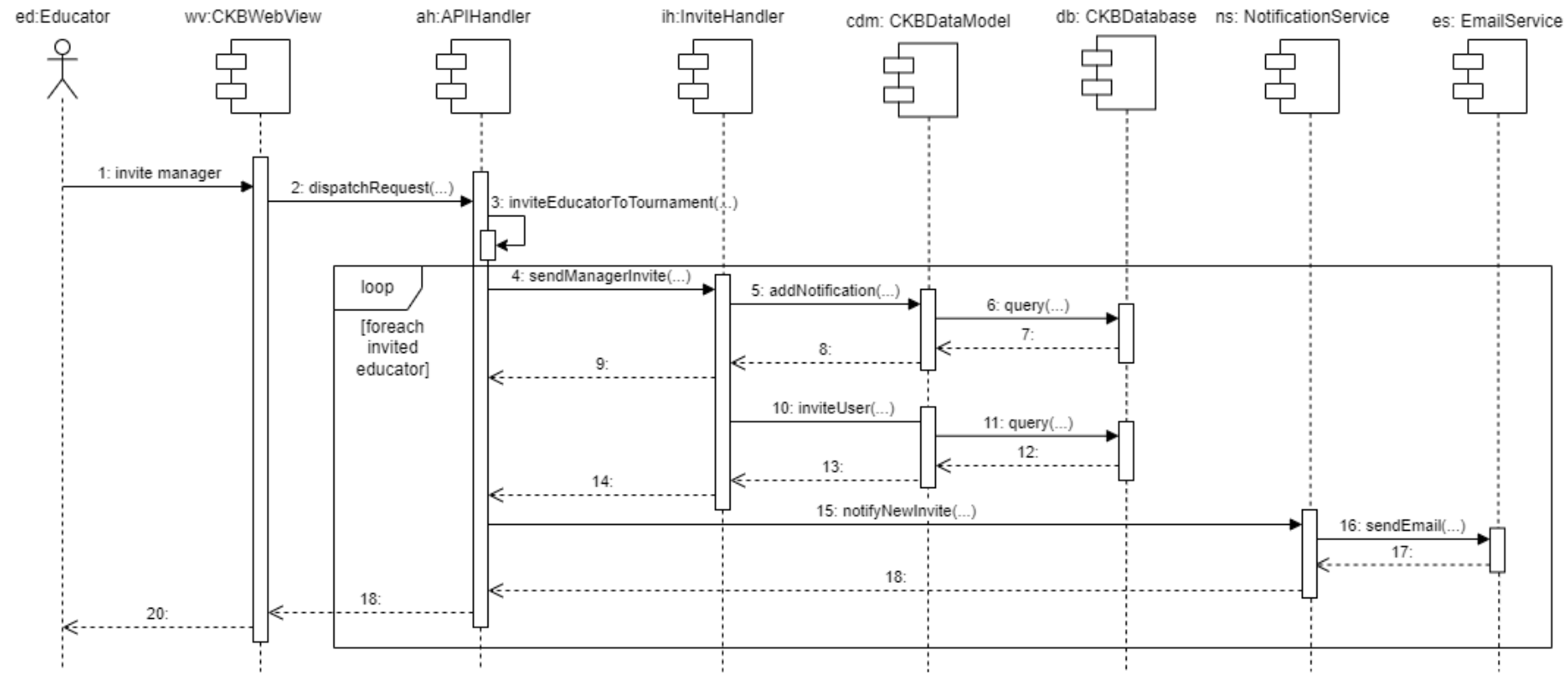
[UC8]: Badge creation



This sequence diagram displays the creation of a badge by an educator for a specific tournament. Once the request is received, the RequestDispatcher component delegates the badge creation request to the specific RequestHandler which, in turn, will generate the rules

and then the badge itself using the Builders component. Once the badge has been created, and the rules have been assigned to it, it's stored in the platform's database.

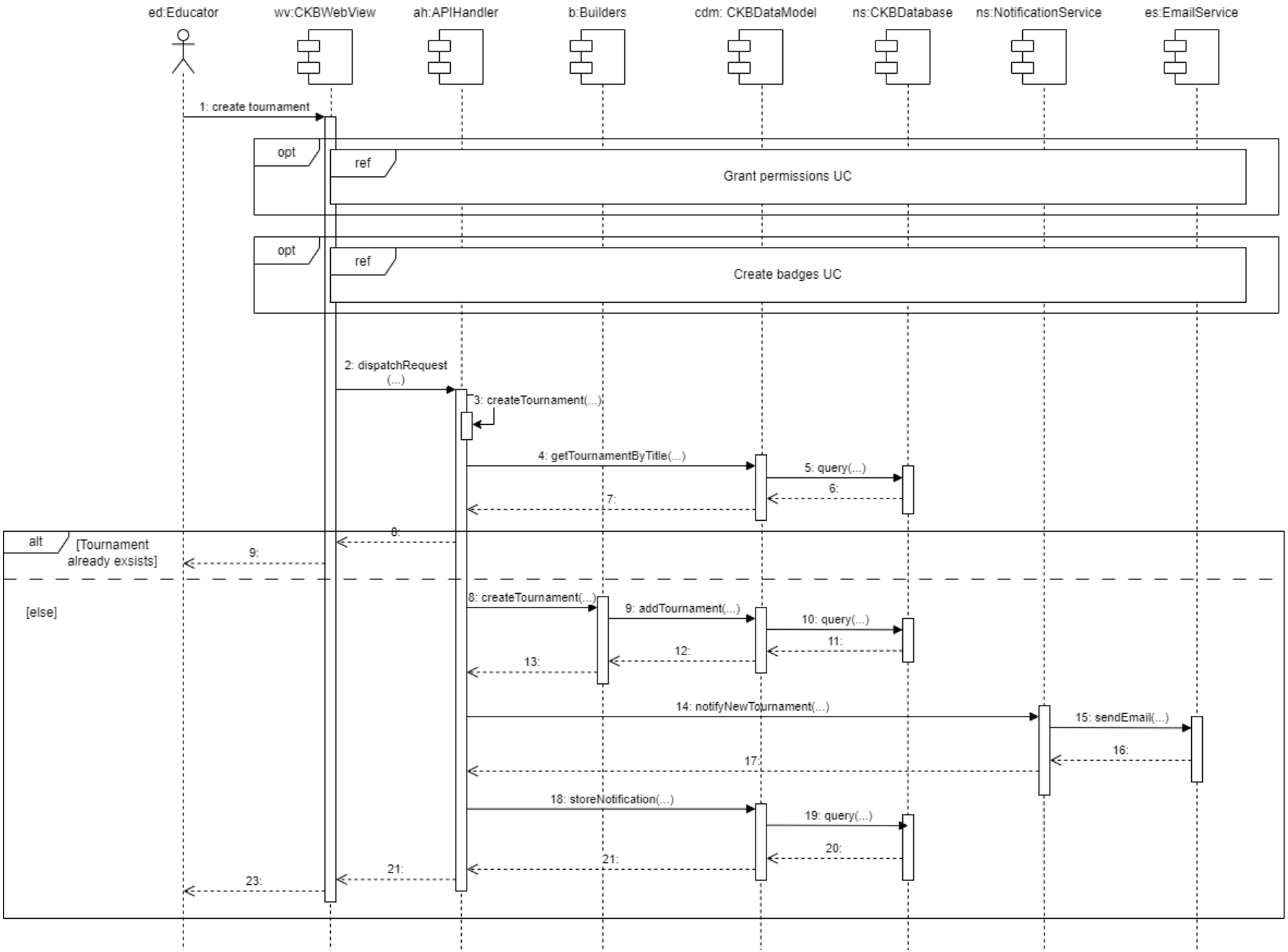
[UC9]: Grant permissions to other educators



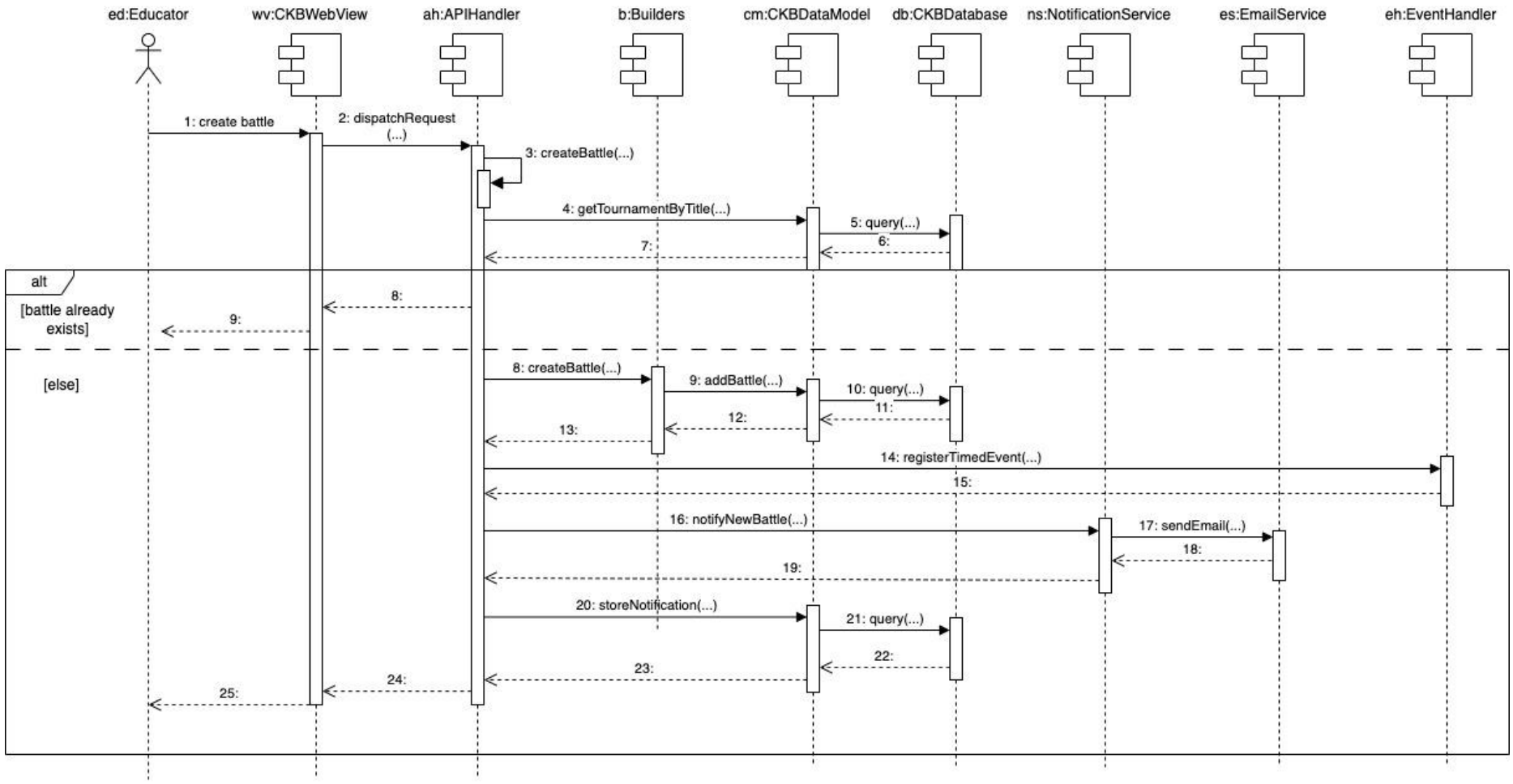
This sequence diagram explains the internal operations of the platform when an educator invites other educators to manage a tournament. First the request is dispatched to the InviteHandler component which generates for each invited educator a new invite and sends a notification using the email service. The notification and the invite are also stored for displaying it to the invited educators when they access the application and to allow them to accept or deny the invite.

[UC10]: Tournament creation

This sequence diagram describes an educator creating a tournament. At this stage, the educator can grant permissions to other educators and create badges for the tournament. The request to create a tournament is managed by the APIHandler which is forwarded to the Builders component specialized, in this case, in the creation of tournaments. The created tournament is saved in the database. Those registered to the platform are then notified via email that a new tournament is available. This notification is also saved in the platform's internal notification database.


ed:Educator
wv:CKBWebView
ah:APIHandler
b:Builders
cdm: CKBDataModel
ns:CKBDatabase
ns:NotificationService
es:EmailService
1: create tournament
opt
ref
Grant permissions UC
opt
ref
Create badges UC
2: dispatchRequest (...)
3: createTournament(...)
4: getTournamentByTitle(...)
5: query(...)
6:
7:
alt
[Tournament already exsists]
8:
9:
[else]
8: createTournament(...)
9: addTournament(...)
10: query(...)
11:
12:
13:
14: notifyNewTournament(...)
15: sendEmail(...)
16:
17:
18: storeNotification(...)
19: query(...)
20:
21:
21:
23:

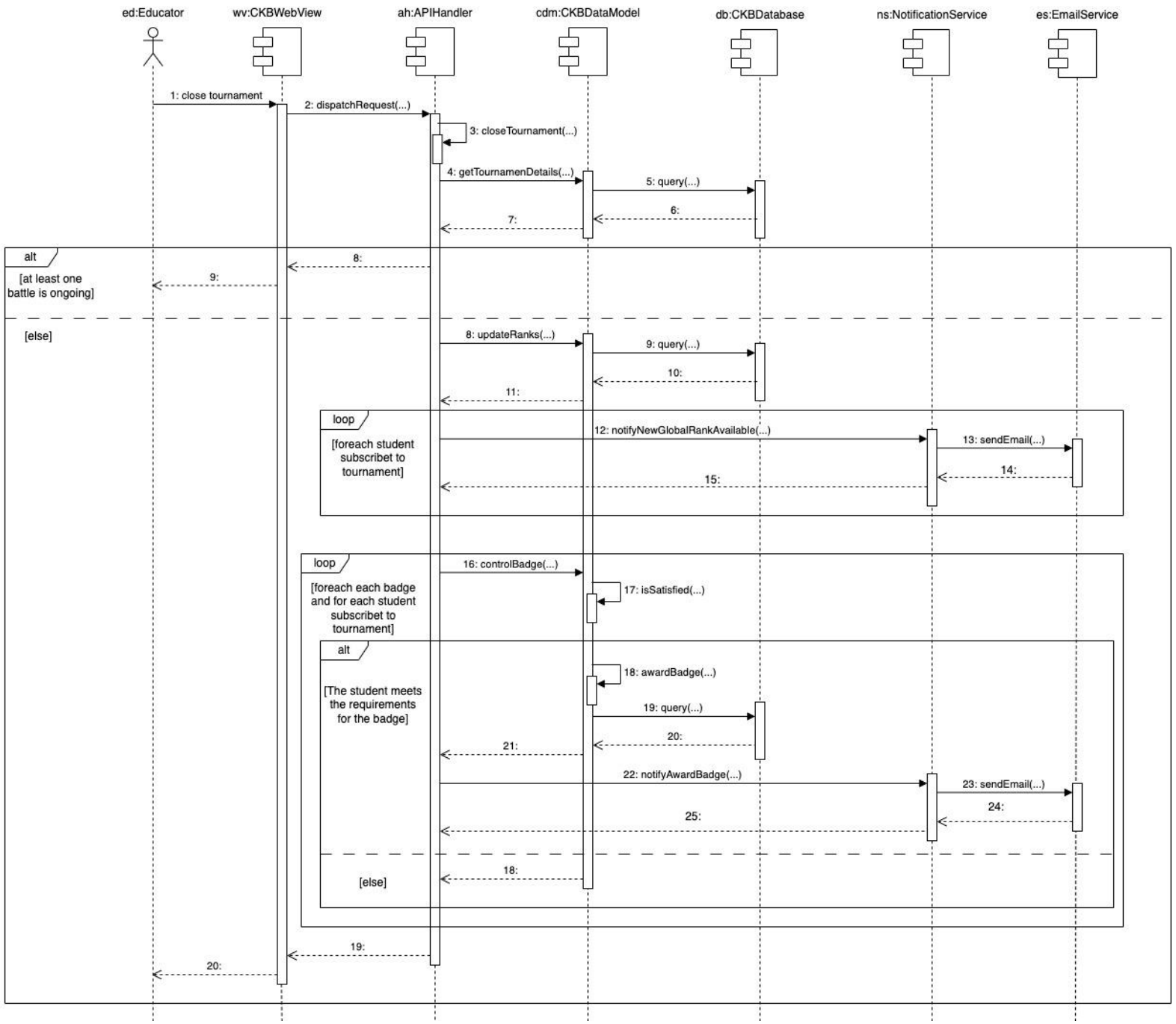
[UC11]: Creation of battles



This sequence diagram describes the flow of operations for creating a battle. First, the educator requests the battle creation which is handled by the APIHandler. It is verified whether or not there is an equal battle, if not, the process moves forward with the actual creation. APIHandler sends the request to the Builders component to create the battle with all the information sent by the educator and is then saved in the database. Finally, emails are sent to each student registered for that tournament.

[UC12]: Closing of a tournament

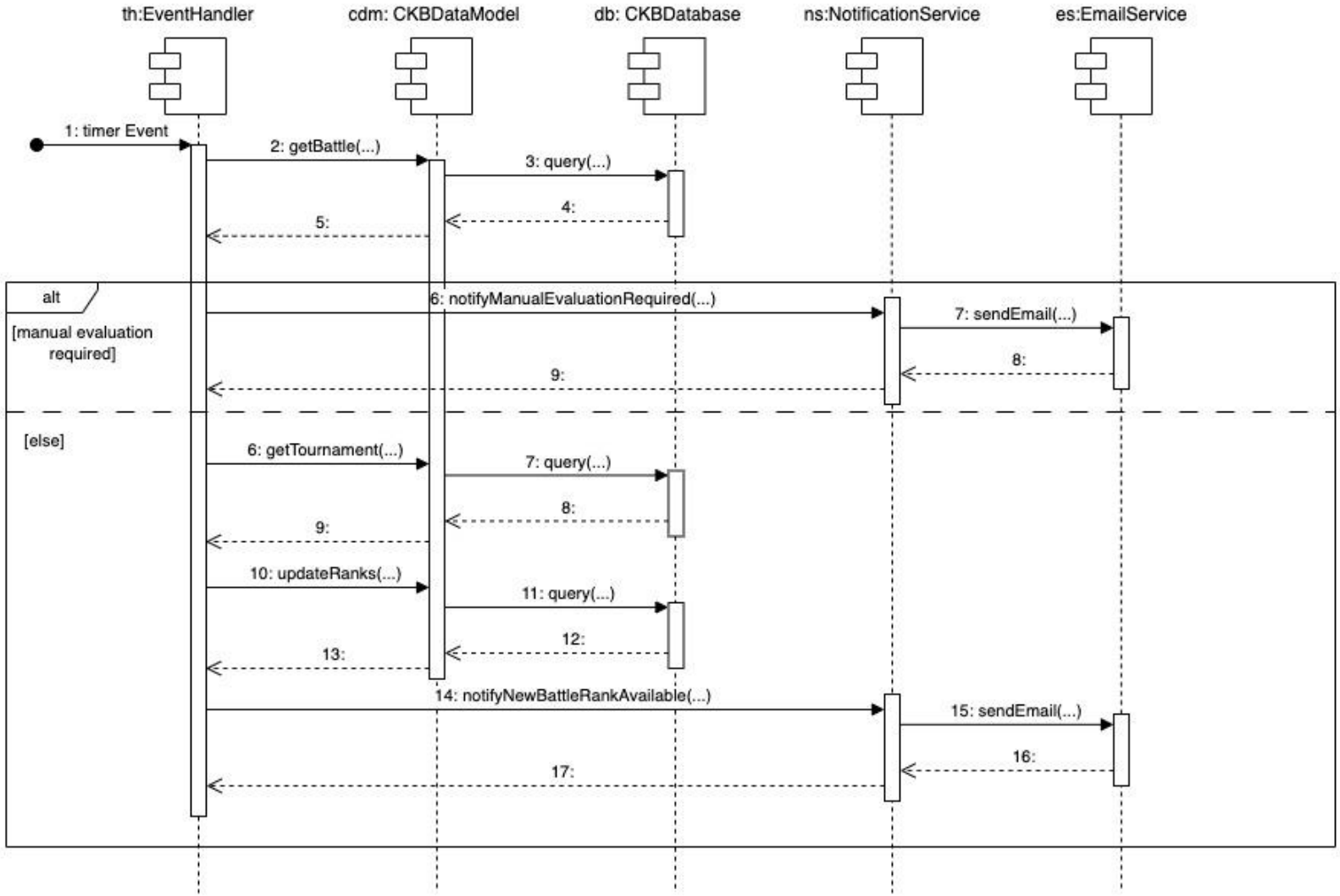
This sequence diagram describes when a tournament closes. The educator sends the request to close the tournament to the APIHandler which then checks that there are no open battles. If they are all closed, then the global rank of students in the tournament are updated and the tournament is flagged as closed. Each student registered to it is notified of the event. The platform also checks whether the students who participated in the tournament satisfy the requirements to receive badges. If so, the assignment is saved in the database and an email is sent to the students.

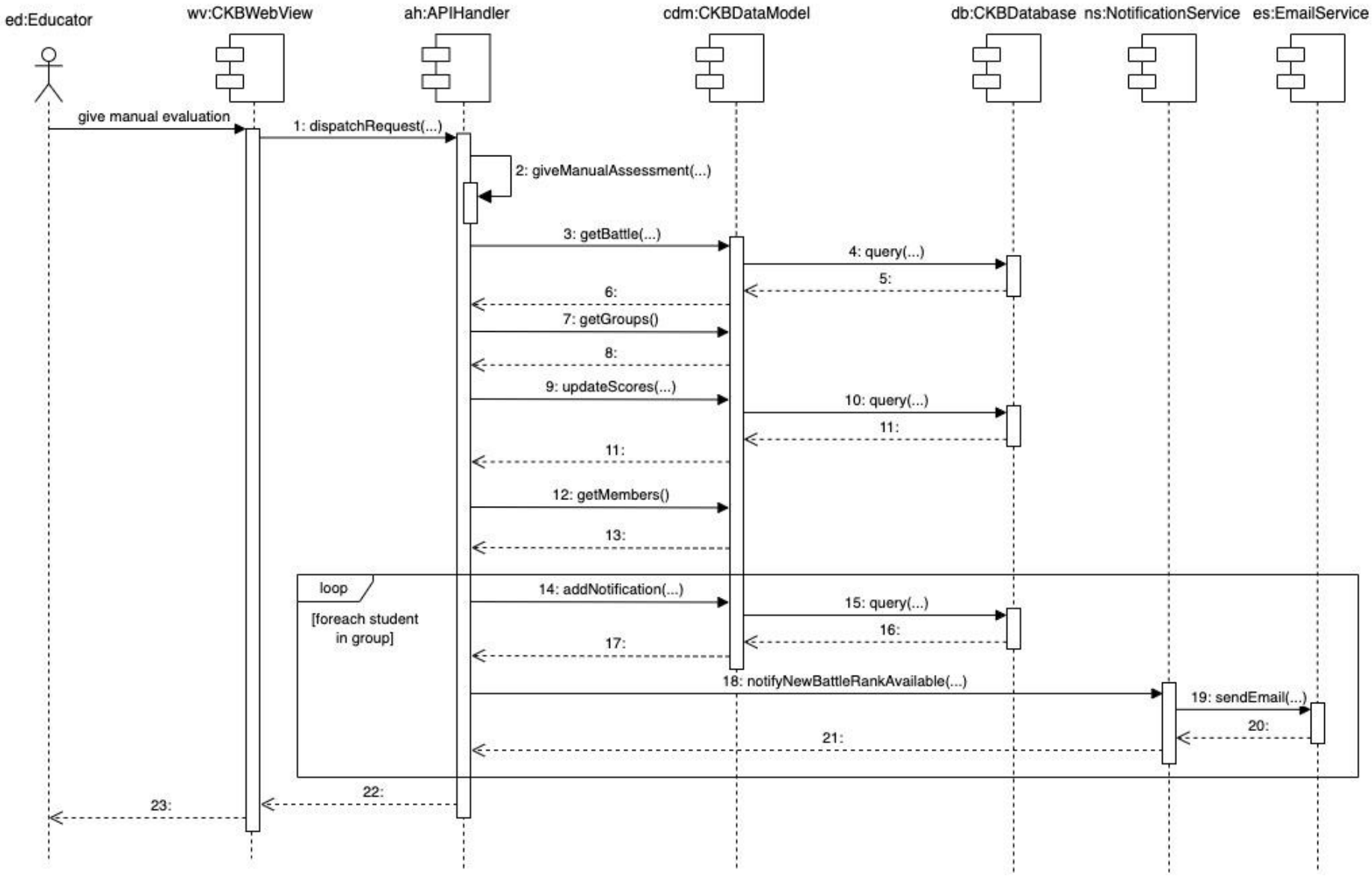

ed:Educator
wv:CKBWebView
ah:APIHandler
cdm:CKBDataModel
db:CKBDatabase
ns:NotificationService
es:EmailService
1: close tournament
2: dispatchRequest(...)
3: closeTournament(...)
4: getTournamenDetails(...)
5: query(...)
6:
7:
alt
[at least one battle is ongoing]
8:
9:
[else]
8: updateRanks(...)
9: query(...)
10:
11:
loop
[foreach student subscribet to tournament]
12: notifyNewGlobalRankAvailable(...)
13: sendEmail(...)
14:
15:
loop
[foreach each badge and for each student subscribet to tournament]
16: controlBadge(...)
17: isSatisfied(...)
alt
[The student meets the requirements for the badge]
18: awardBadge(...)
19: query(...)
20:
21:
22: notifyAwardBadge(...)
23: sendEmail(...)
24:
25:
[else]
18:
19:
20:

[UC13]: Give manual assessment

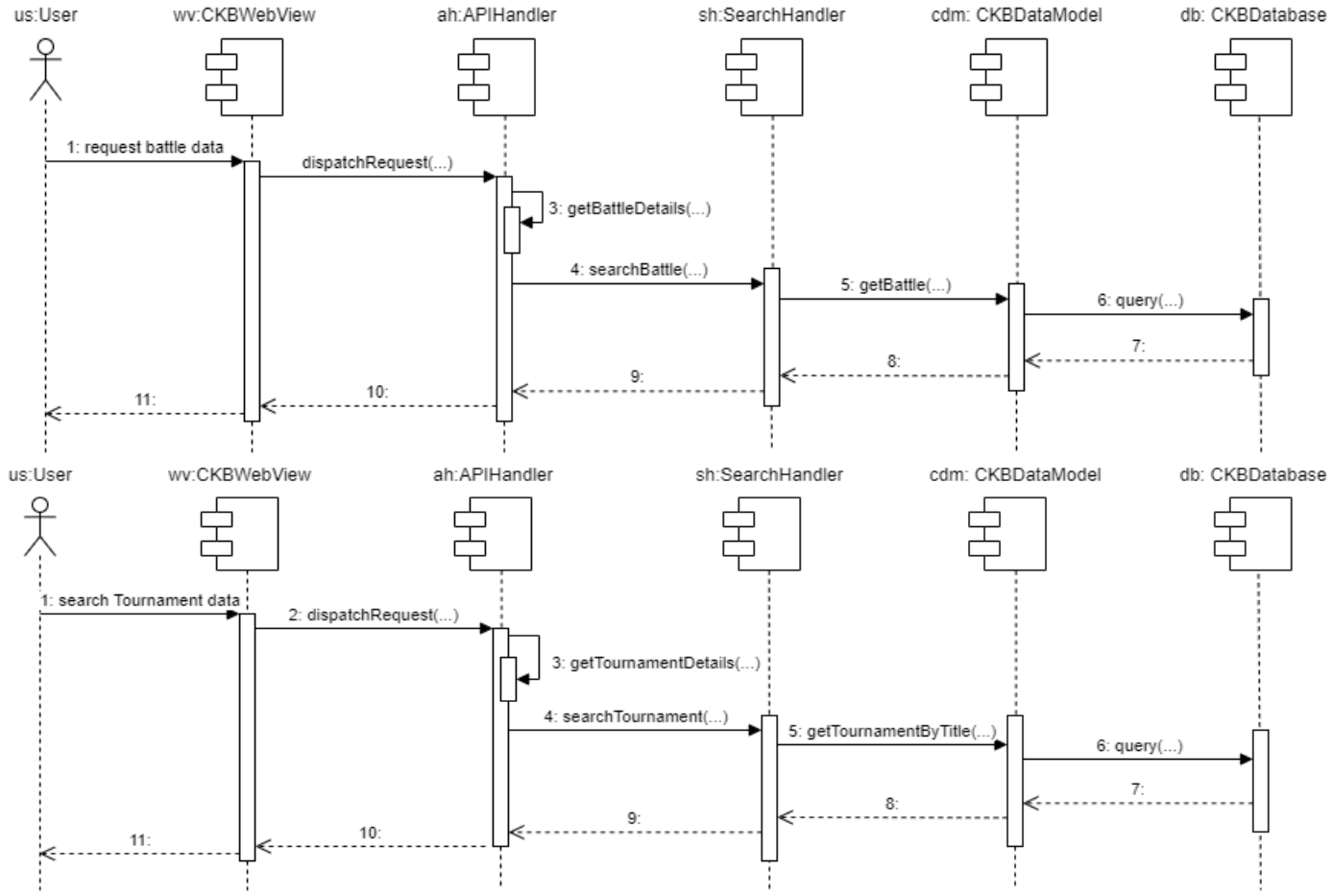
The first sequence diagram describes the sequences of operations from when the EventHandler triggers at the battle submission deadline. First, the EventHandler loads the battle data and, if a manual evaluation is required, it sends an email to urge the educators to score students' work. If manual evaluation is not required, the points are calculated for each group, and the battle rank is therefore updated. All this information is saved in the database and an email is sent to all students participating in the battle notifying them that a new rank is available.

The second sequence diagram describes the process of giving a manual evaluation. The educator requests to insert their manual evaluation into the platform, this request is intercepted by the APIHandler which asks the CKBDataModel to load the battle data and each participating group. The score of each group is updated with the score given in the manual evaluation and an email is sent to all students in the group.


th:EventHandler
cdm: CKBDataModel
db: CKBDatabase
ns:NotificationService
es:EmailService
1: timer Event
2: getBattle(...)
3: query(...)
4:
5:
alt
[manual evaluation required]
6: notifyManualEvaluationRequired(...)
7: sendEmail(...)
8:
9:
[else]
6: getTournament(...)
7: query(...)
8:
9:
10: updateRanks(...)
11: query(...)
12:
13:
14: notifyNewBattleRankAvailable(...)
15: sendEmail(...)
16:
17:


ed:Educator
wv:CKBWebView
ah:APIHandler
cdm:CKBDataModel
db:CKBDatabase
ns:NotificationService
es:EmailService
give manual evaluation
1: dispatchRequest(...)
2: giveManualAssessment(...)
3: getBattle(...)
4: query(...)
5:
6:
7: getGroups()
8:
9: updateScores(...)
10: query(...)
11:
11:
12: getMembers()
13:
loop
[foreach student in group]
14: addNotification(...)
15: query(...)
16:
17:
18: notifyNewBattleRankAvailable(...)
19: sendEmail(...)
20:
21:
22:
23:

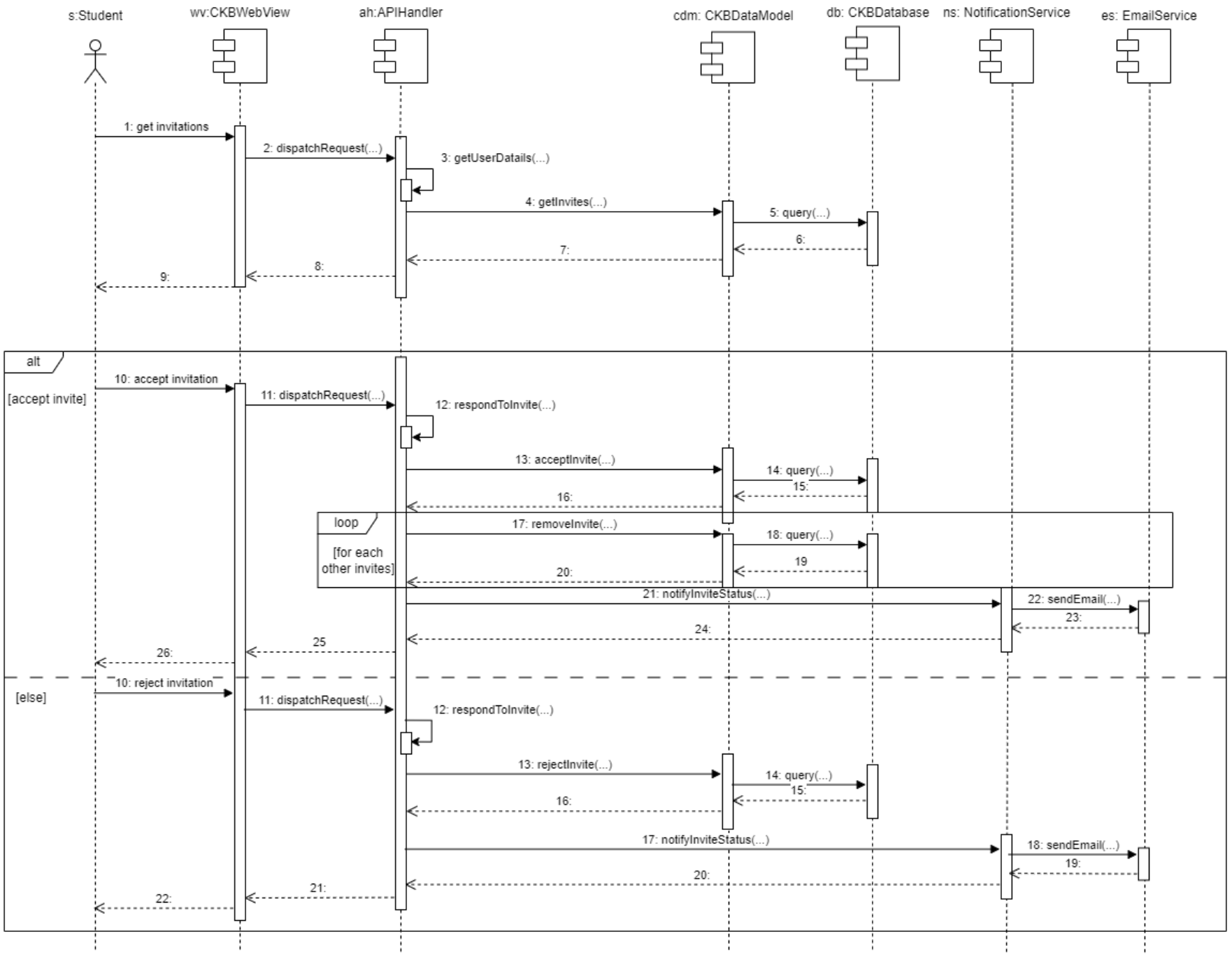
[UC14]: Search battle / tournament



These sequence diagrams describe the search for a tournament or battle respectively. The search request is handled by the APIHandler which looks for the requested tournament/battle through the SearchHandler component.

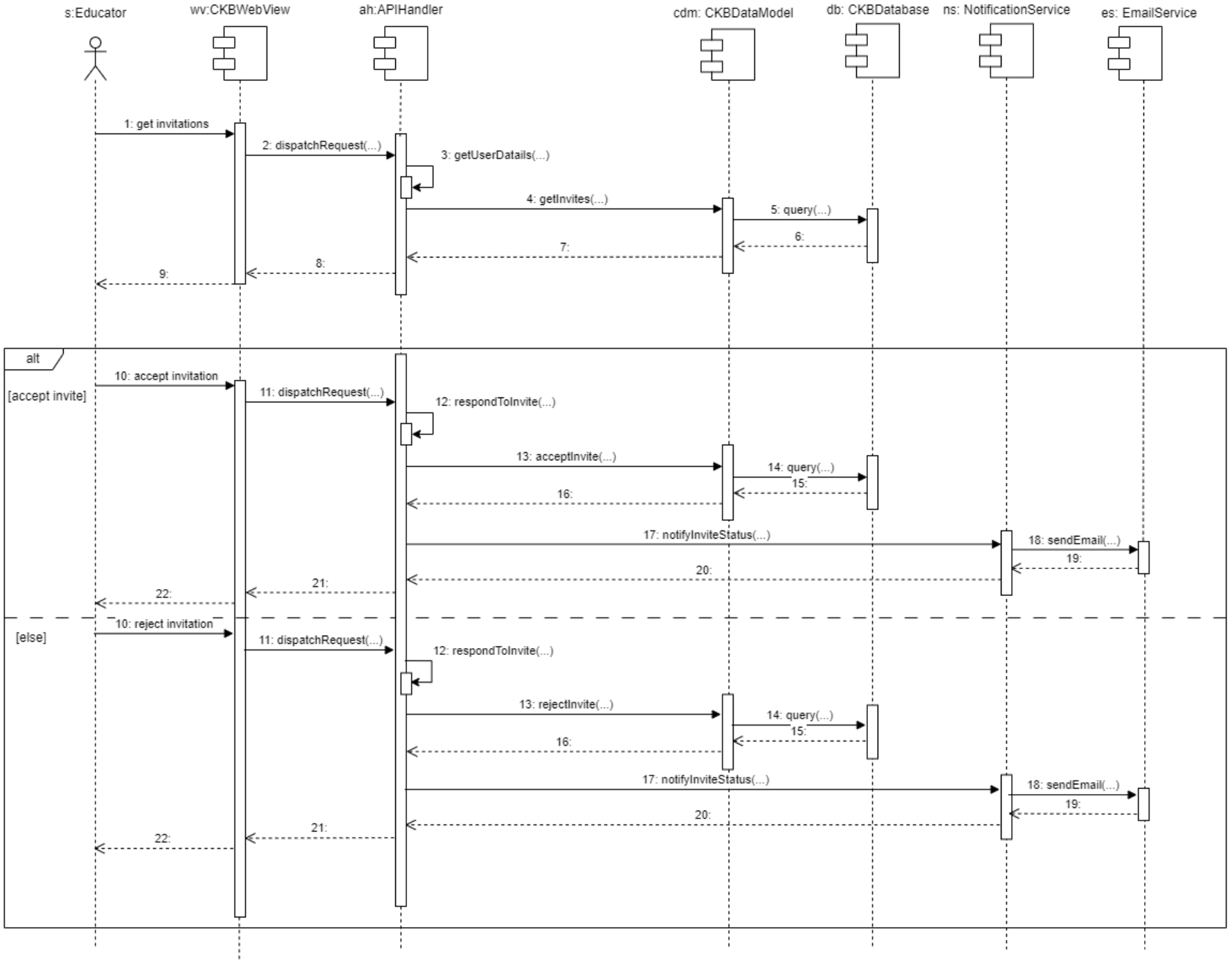
[UC15]: Manage invitations to groups

This sequence diagram shows the internal operations for accepting or rejecting invites to groups in battles. The student makes a request to view the list of invitations that they have received. The student can accept or decline the invitations. If they accept the invitation, they are recorded in the model and in the database. All other invitations for that particular battle are automatically rejected.


s:Student
wv:CKBWebView
ah:APIHandler
cdm: CKBDataModel
db: CKBDatabase
ns: NotificationService
es: EmailService
1: get invitations
2: dispatchRequest(...)
3: getUserDatails(...)
4: getInvites(...)
5: query(...)
6:
7:
8:
9:
alt
[accept invite]
10: accept invitation
11: dispatchRequest(...)
12: respondToInvite(...)
13: acceptInvite(...)
14: query(...)
15:
16:
loop
[for each other invites]
17: removeInvite(...)
18: query(...)
19
20:
21: notifyInviteStatus(...)
22: sendEmail(...)
23:
24:
25
26:
[else]
10: reject invitation
11: dispatchRequest(...)
12: respondToInvite(...)
13: rejectInvite(...)
14: query(...)
15:
16:
17: notifyInviteStatus(...)
18: sendEmail(...)
19:
20:
21:
22:

[UC16]: Manage invitations to tournament

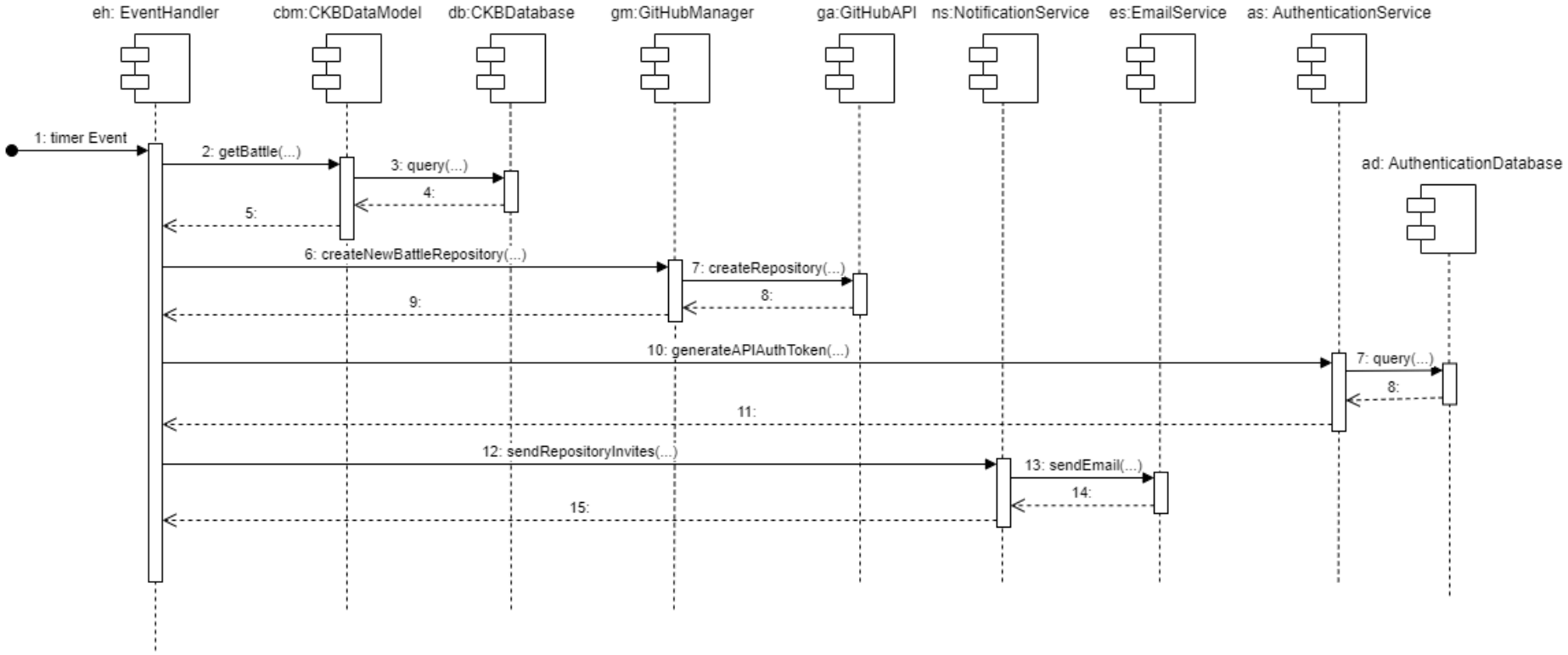
This sequence diagram shows the internal operations for accepting or rejecting invites to manage tournaments. The educator makes a request to view the list of invitations that they have received. The educator can either accept or decline the invitations. If they accept, the choice is recorded within the model and in the database. Whether the educator accepts or rejects the invite, an email is sent to notify the inviter of the acceptance or rejection of the invitation.


s:Educator
wv:CKBWebView
ah:APIHandler
cdm: CKBDataModel
db: CKBDatabase
ns: NotificationService
es: EmailService
1: get invitations
2: dispatchRequest(...)
3: getUserDatails(...)
4: getInvites(...)
5: query(...)
6:
7:
8:
9:
alt
[accept invite]
10: accept invitation
11: dispatchRequest(...)
12: respondToInvite(...)
13: acceptInvite(...)
14: query(...)
15:
16:
17: notifyInviteStatus(...)
18: sendEmail(...)
19:
20:
21:
22:
[else]
10: reject invitation
11: dispatchRequest(...)
12: respondToInvite(...)
13: rejectInvite(...)
14: query(...)
15:
16:
17: notifyInviteStatus(...)
18: sendEmail(...)
19:
20:
21:
22:

Additional diagrams

The following sequence diagrams do not reference any particular use case but are useful to illustrate some of the internal functions of the platform that are relevant to this project. In particular, they represent the behavior of the platform when handling events such as the beginning of a battle (and subsequent creation of a repository) and the automatic evaluation of students' solutions when pushing on GitHub.

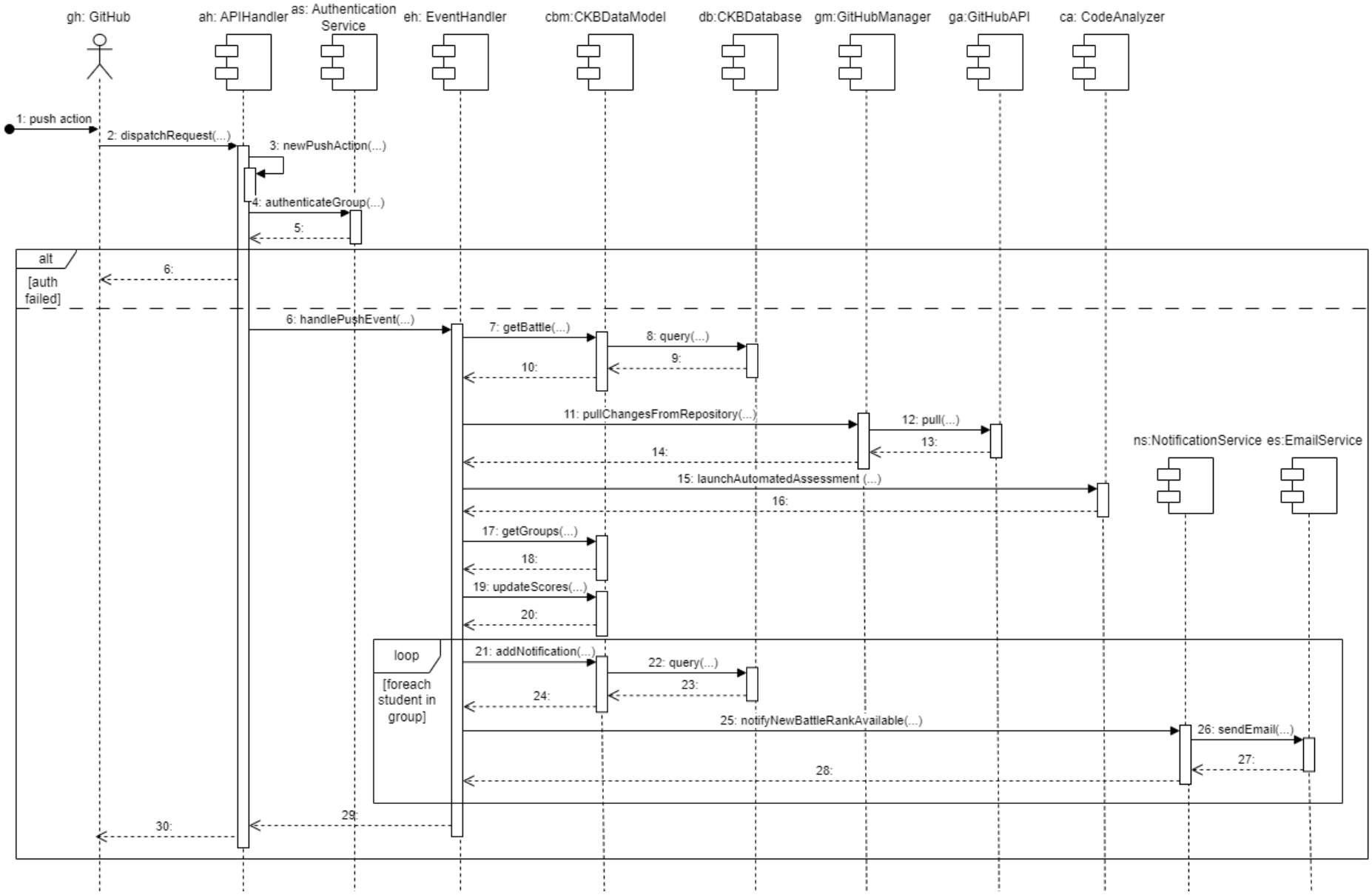
Battle start



This sequence diagram shows what happens when the enrollment deadline for a battle is reached. Upon reaching the enrollment deadline, the EventHandler loads the battle and creates the repository by making a request to the GitHubManager component which forwards the request to GitHub.All groups participating in the battle will receive an invite to the new repository along with a group-unique token used for API calls.

Push action and Automated Evaluation

The following diagram shows the process of automated evaluation performed when a new push action occurs on the main branch of the GitHub repository of a group. More precisely, the push action triggers an automatic workflow in the repository that makes a call to the platform's API that takes on the task of handling the automated evaluation. Note that, as also mentioned in the RASD, the call to the API must be authenticated to prevent security issues. Once the automated evaluation process is concluded, the students in the group are notified of the update of their score and rank.


gh: GitHub
ah: APIHandler
as: Authentication Service
eh: EventHandler
cbm:CKBDataModel
db:CKBDatabase
gm:GitHubManager
ga:GitHubAPI
ca: CodeAnalyzer
ns:NotificationService
es:EmailService
1: push action
2: dispatchRequest(...)
3: newPushAction(...)
4: authenticateGroup(...)
5:
alt
[auth failed]
6:
6: handlePushEvent(...)
7: getBattle(...)
8: query(...)
9:
10:
11: pullChangesFromRepository(...)
12: pull(...)
13:
14:
15: launchAutomatedAssessment (...)
16:
17: getGroups(...)
18:
19: updateScores(...)
20:
loop
[foreach student in group]
21: addNotification(...)
22: query(...)
23:
24:
25: notifyNewBattleRankAvailable(...)
26: sendEmail(...)
27:
28:
29:
30:

2.5. Component Interfaces

2.5.1 Main Components Interfaces

The following section presents the public interfaces of the components for the CKBApplication as presented in the component diagram in Section 2.4.

- **APIGateway**:
  - `dispatchRequest(req: Request, res: Response): void`
- **CKBDataModel**:
  - `getUserByName(username: String): User`
  - `getUserByEmail(email: String): User`
  - `addUser(user: User): void`
  - `getTournamentByTitle(title: String): Tournament`
  - `addTournament(tournament: Tournament): void`
  - `getTournaments(): List<Tournament>`
  - `getBattle(title: String): Battle`
  - `addNotification(notification: Notification): void`
- **NotificationServiceI**:
  - `notifyNewAccountCreated(user: User): void`
  - `notifyTouranmentSubscription(tournament: Tournament, user: User): void`
  - `notifyNewTournament(tournament: Tournament): void`
  - `notifyPermissionUpdate(tournament: Tournamet): void`
  - `notifyManualEvaluationRequired(educators: List<Educator>, battle: Battle): void`
  - `notifyNewBattleRankAvailable(students: List<Student>, battle: Battle): void`
  - `notifyNewGlobalRankAvailable(students: List<Student>, tournament: Tournament): void`
  - `sendRepositoryInvites(battle: Battle, repositoryUrl: String)`
  - `notifyNewInvite(from: User, to: User, invite: Invite): void`
  - `notifyInviteStatus(from: User, to: User, invite: Invite, accepted: boolean): void`
- **AuthenticationServiceI**:
  - `registerNewAccount(user: User, password: String): void`
  - `login(user: User, password: String): boolean`
  - `generateAPIAuthToken(group: Group): String`
  - `authenticateGroup(token: String): boolean`
  - `getGroupToken(group: Group): String`
- **BuildersI**:

    - createUser(username: String, email: String, password: String): User
    - createTournament(owner: Educator, title: String, deadline: DateTime, managers: List<Educator>, badges: List<Badge>): Tournament
    - createBattle(title: String, description: String, enrollmentDeadline: DateTime, submissionDeadline: DateTime, minGroupSize: Integer, maxGroupSize: Integer, scripts: File, projectFiles: List<File>): Battle
    - createBadge(title: String, iconUrl: String, rules: List<BadgeRule>): Badge
    - createBadgeRule(components: List<Object>): BadgeRule
- **EventHandlerI**:
    - start(): void
    - stop(): void
    - registerTimedEvent(event: ITimedEvent, listener: IListener): void
    - handlePushEvent(respositoryUrl: String): EvaluationResults
- **GitHubManagerI**:
    - createNewBattleRepository(battle: Battle, files: List<File>): boolean
    - pullChangesFromRepository(repositoryUrl: String): List<File>
- **CodeAnalyzerI**:
    - launchAutomatedAssessment(projectDirectory: String, automationScripts: File, evaluationParameters: List<EvalParameter>): EvaluationResults
- **InviteHandlerI**:
    - sendGroupInvite(from: Student, to: Student, battle: Battle): void
    - sendManagerInvite(from: Educator, to: Educator, tournament: Tournament): void
    -
- **SearchHandlerI**:
    - searchBattle(battleTitle: String): List<Battle>
    - searchTournament(tournamentTitle: String): List<Tournament>
    - searchStudent(usernameOrEmail: String): List<Student>
    - searchEducator(usernameOrEmail: String): List<Educator>

2.5.2 Sub-components Interfaces

The following section presents the interfaces for components located within those shown in the component diagram.

APIHandler:

- **APIGateway:**
  - `dispatchRequest(req: Request, res: Response): void`
- **RequestHandlersI**:
  - `createNewAccount(data: JSON): JSON`
  - `login(data: JSON): JSON`
  - `subscribeToTournament(data: JSON): JSON`
  - `createBattleGroup(data: JSON): JSON`
  - `getBattleDetails(data: JSON): JSON`
  - `getTournamentDetails(data: JSON): JSON`
  - `getUserDetails(data: JSON): JSON`
  - `getGroupDetails(data: JSON): JSON`
  - `createTournament(data: JSON): JSON`
  - `createBattle(data: JSON): JSON`
  - `createBadge(data: JSON): JSON`
  - `newPushAction(data: JSON): JSON`
  - `giveManualAssessment(data: JSON): JSON`
  - `inviteStudentsToGroup(data: JSON): JSON`
  - `inviteEducatorToTournament(data: JSON): JSON`
  - `closeTournament(data: JSON): JSON`
  - `respondToInvite(data: JSON): JSON`

CKBDataModel:

- **UserI**:
  - `getUsername(): String`
  - `getEmail(): String`
  - `addInvite(invite: Invite): void`
  - `getInvites(): List<Invite>`
  - `removeInvite(invite: Invite): void`
  - `addNotification(notification: Notification): void`
  - `getNotifications(): List<Notification>`
  - `removeNotification(notification: Notification): void`
- **StudentI**:
  - `getBadges(): List<Badge>`
  - `awardBadge(badge: Badge): void`
- **TournamentI**:
  - `getOwner(): Educator`
  - `getManagers(): List<Educator>`
  - `getBattle(battleTitle: String): Battle`

    - `getBadges(): List<Badge>`
    - `addBattle(battle: Battle): void`
    - `addBadge(badge: Badge): void`
    - `updateRanks(): void`
    - `subscribe(student: Student): void`
    - `getSubscriptionDeadline(): DateTime`
    - `getInvited(): List<Student>`
    - `inviteUser(educator: Educator): void`
    - `acceptInvite(username: String): void`
    - `rejectInvite(username: String): void`
    - `close(): void`
- BattleI:
    - `getGroups(): List<Group>`
    - `addGroup(group: Group): void`
    - `removeInvalidGroups(): List<Group>`
    - `updateRanks(): void`
    - `setRepository(repositoryUrl: String): void`
    - `getEvaluationParameters(): List<EvalParameter>`
- GroupI:
    - `getMembers(): List<Student>`
    - `getInvited(): List<Student>`
    - `inviteUser(student: Student): void`
    - `acceptInvite(username: String): void`
    - `rejectInvite(username: String): void`
    - `updateScores(evaluationParameters: Dictionary<EvalParameter, Integer>): void`
    - `getScores(): Dictionary<EvalParameter, Integer>`
    - `getRepository(): String`
    - `getAuthenticationToken(): String`
- BadgeI:
    - `isSatisfied(student: Student): boolean`
- NotificationI:
    - `getSource(): String`
    - `getMessage(): String`
- InviteI:
    - `getSource(): User`
    - `getMessage(): String`
    - `getContext(): IInvitableContext`

Builders:

- UserAccountCreatorI:
    - `createUser(username: String, email: String, password: String, type: AccountType): User`
- TournamentBuilderI:

    - `build(): Tournament`
    - `setOwner(owner: Educator): TournamentBuilder`
    - `setTitle(title: String): TournamentBuilder`
    - `setSubscriptionDeadline(deadline: DateTime): TournamentBuilder`
    - `inviteManager(educator: Educator): TournamentBuilder`
    - `addBadge(badge: Badge): TournamentBuilder`
- **BattleBuilderI**:
    - `build(): Battle`
    - `setTitle(title: String): BattleBuilder`
    - `setDescription(description: String): BattleBuilder`
    - `setDeadlines(enrollmentDeadline: DateTime, submissionDeadline: DateTime): BattleBuilder`
    - `setGroupsSize(minSize: Integer, maxSize: Integer): BattleBuilder`
    - `setAutomationScripts(scripts: File): BattleBuilder`
    - `setProjectFiles(files: List<File>): BattleBuilder`
- **BadgeBuilderI**:
    - `build(): Badge`
    - `setName(name: String): BadgeBuilder`
    - `setIcon(iconUrl: String): BadgeBuilder`
    - `addRule(rule: BadgeRule): BadgeBuilder`
- **BadgeRuleBuilderI**:
    - `build(): BadgeRule`
    - `addOperator(operator: BadgeRuleOperator): BadgeRuleBuilder`
    - `addVariable(variable: BadgeRuleVariable): BadgeRuleBuilder`
    - `addConstant(constant: Object): BadgeRuleBuilder`

CodeAnalyzer

- **TestRunnerI**:
    - `launchUnitTests(projectDirectory: String, scipts: File): Dictionary<String,TestStatus>`
- **ECARunnerI**:
    - `launchExternalCodeAnalyzers(projectDirectory: String evaluationParameters: List<EvalParameter>): Dictionary<EvalParameter, Integer>`

2.6. Patterns

- **Model View Controller pattern (MVC)**:

  For the implementation of the web application, we opted for the widely recognized MVC pattern to enhance component decoupling. This pattern offers several advantages, enabling parallelization in both the implementation and testing phases. For example, it allows the independent implementation of CKBWebView and CKBDataModel.

- **Facade pattern**:

  The facade pattern is used at the interface of the CKBApplication for the APIGateway. This allows us to hide all the business logic of the application behind a very simple interface that uses a standard language for communication (JSON).

- **Builder pattern**:

  The application places a significant emphasis on the dynamic creation of various entities, including Battles, Tournaments, Badges, and more. In order to enhance flexibility and facilitate the creation of these entities with customizable parameters, we have adopted the Builder pattern. This design choice empowers us to efficiently construct complex objects while providing a clear and intuitive interface, ensuring that the creation process remains versatile and adaptable to evolving requirements.

- **Observer pattern**:

  The application implements the observer pattern to facilitate the temporal scheduling of events. Specifically, the EventHandler component assumes the responsibility of registering and scheduling events, such as the start and conclusion of battles. This architecture allows designated listeners to easily manage and respond to these events at the appropriate time.

- **Decorator pattern**:

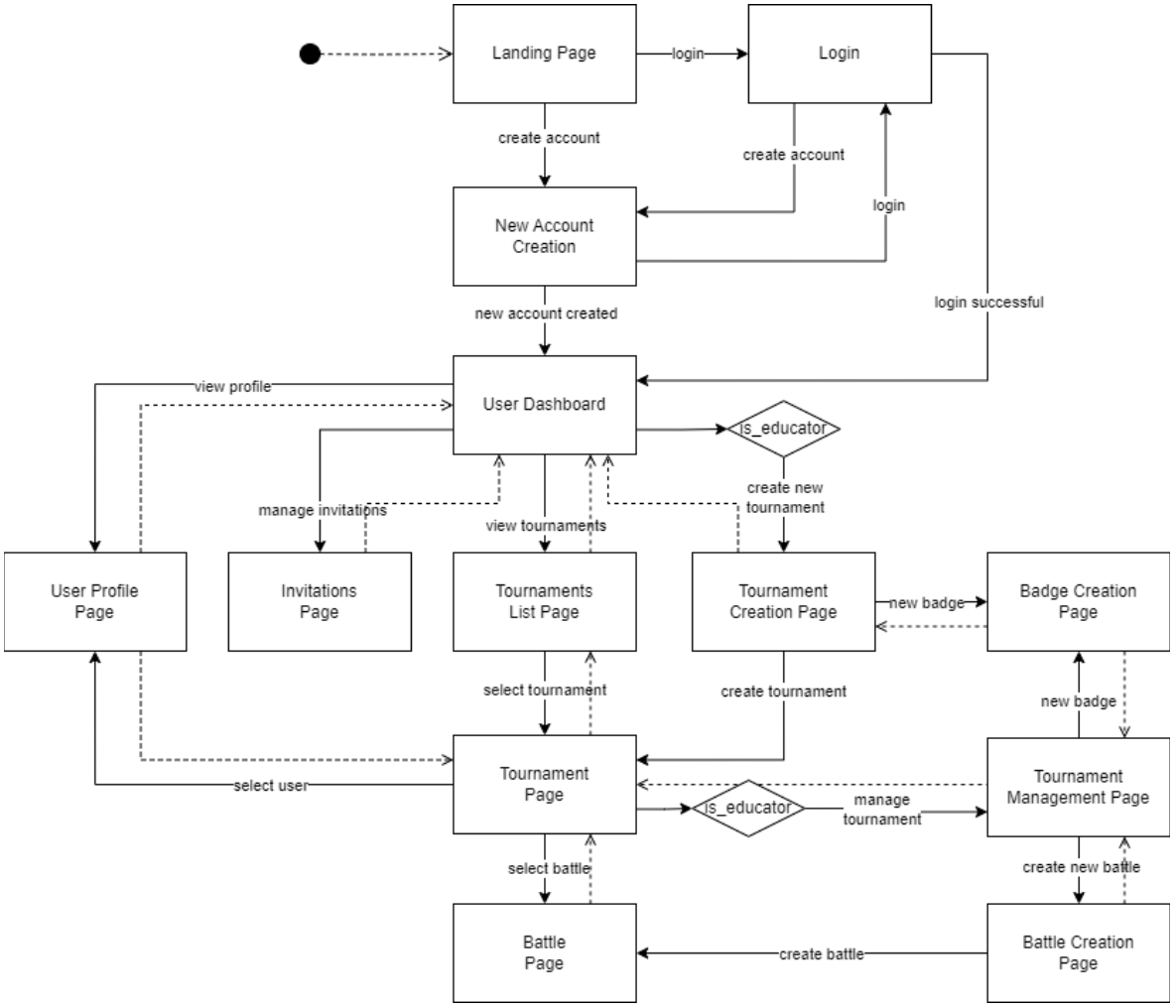
  The decorator pattern plays a role in managing badge rules within the application. Due to the flexible and inherently unknown structure of these rules, the decorator pattern proves well-suited for this task as it enables the composition of badge rules

using pre-defined components, offering a scalable and adaptable solution to accommodate the dynamic nature of said rules.

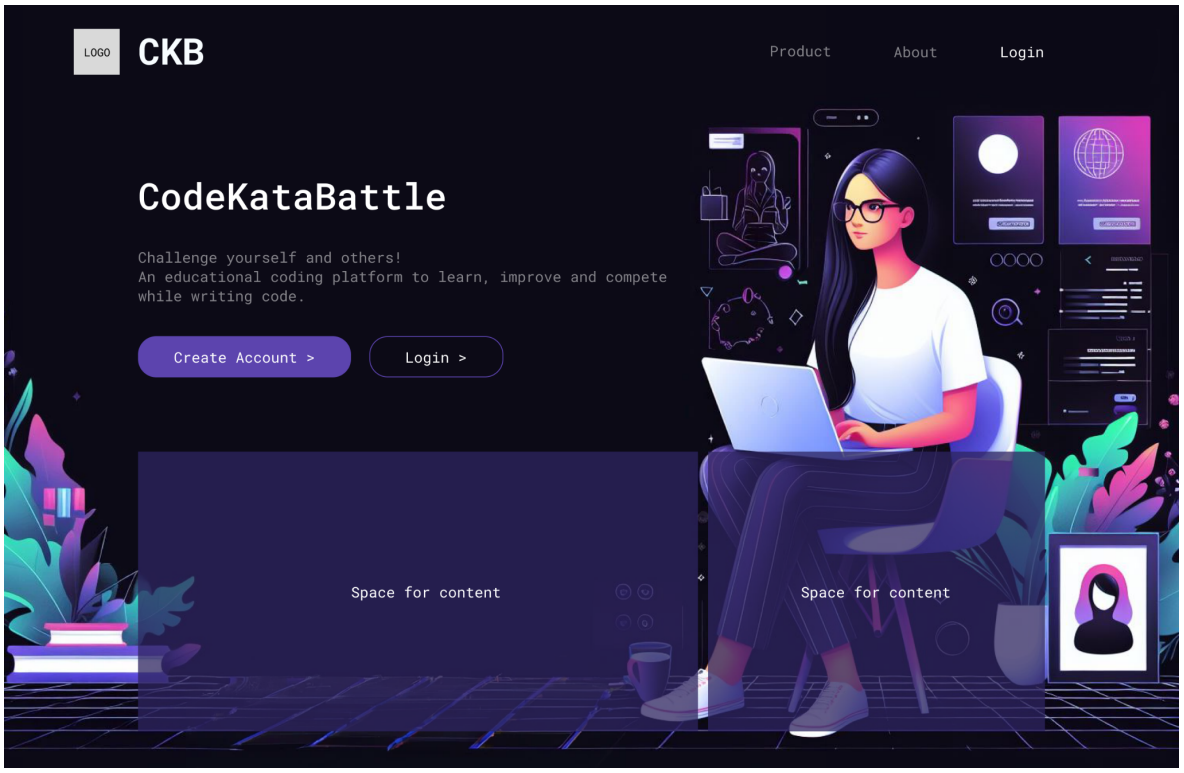
3. User Interface Design

Regarding the user interface (UI), we follow the guidelines already outlined in the RASD to try and guarantee accessibility for all users. In the following section, we will present the navigation diagram for the webpage along with a small selection of layouts of the various user views.

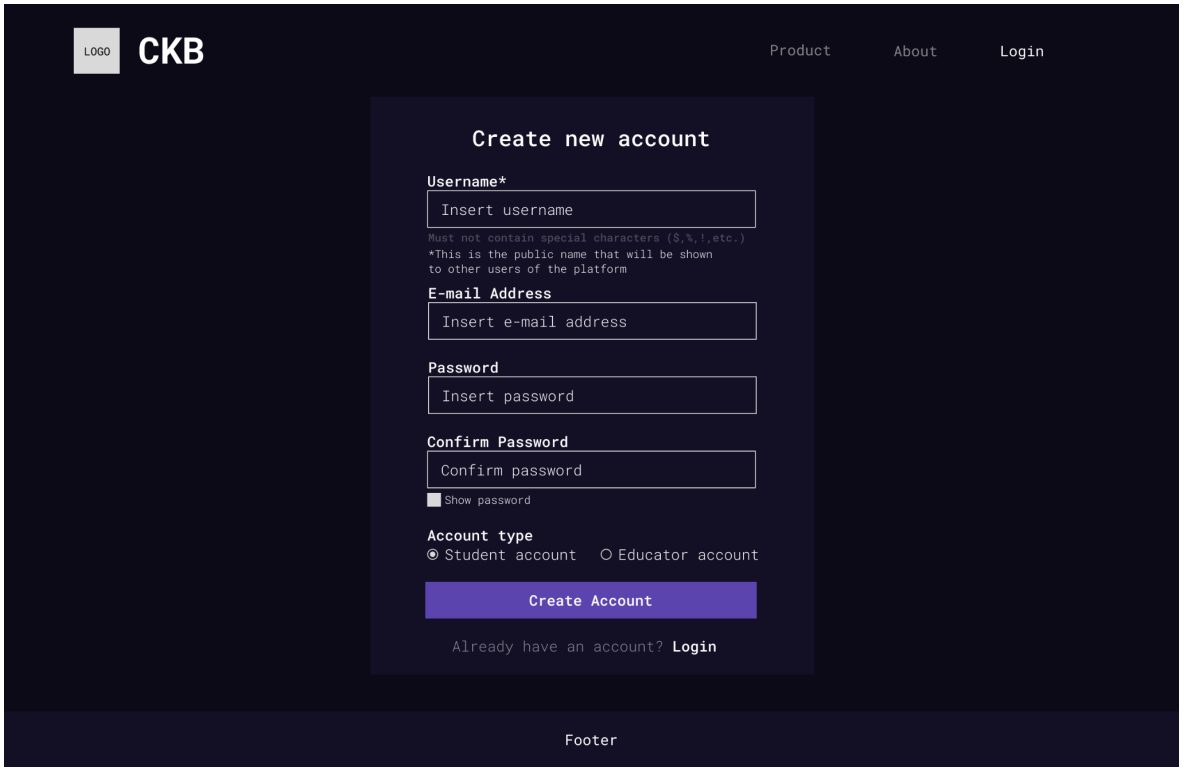
A complete overview of the designs for each of the user's views is made available at the following link Drive/CKB_UI_Design.pdf and will be used as a reference in the implementation phase.



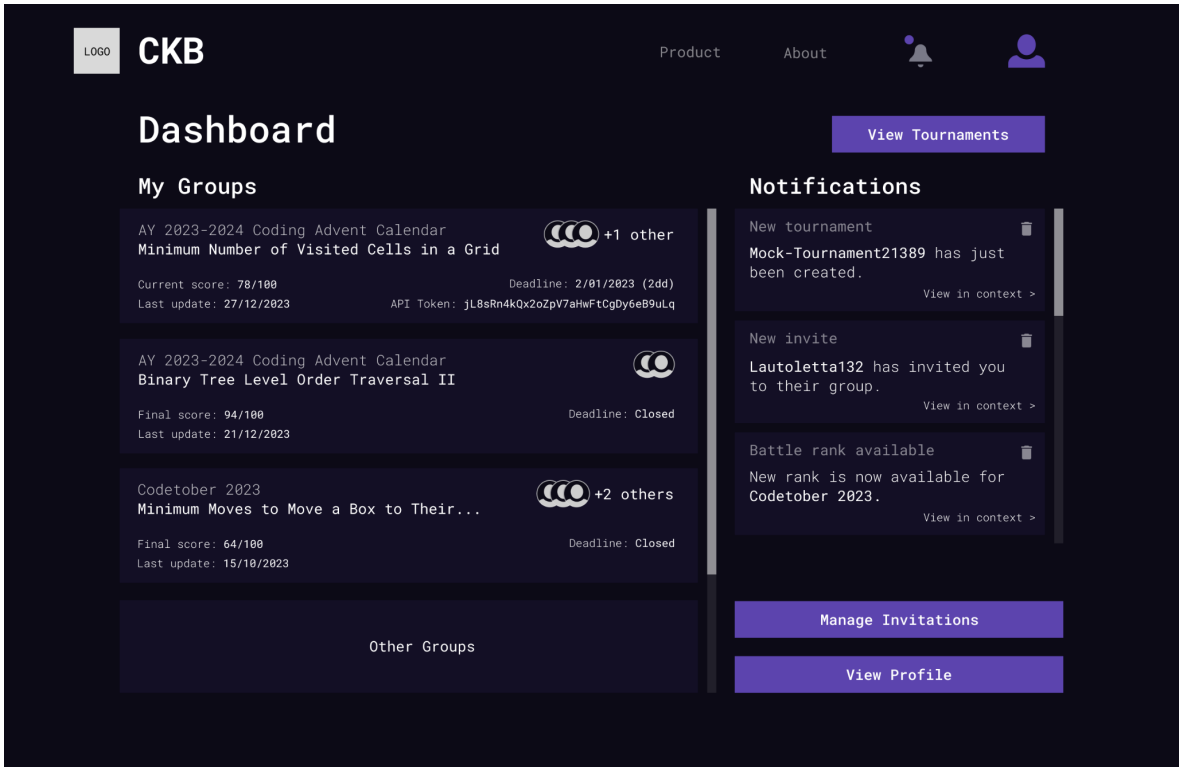
The dashed arrows in the diagram show the possibility of going back to the previous view through the use of a dedicated button on the UI.



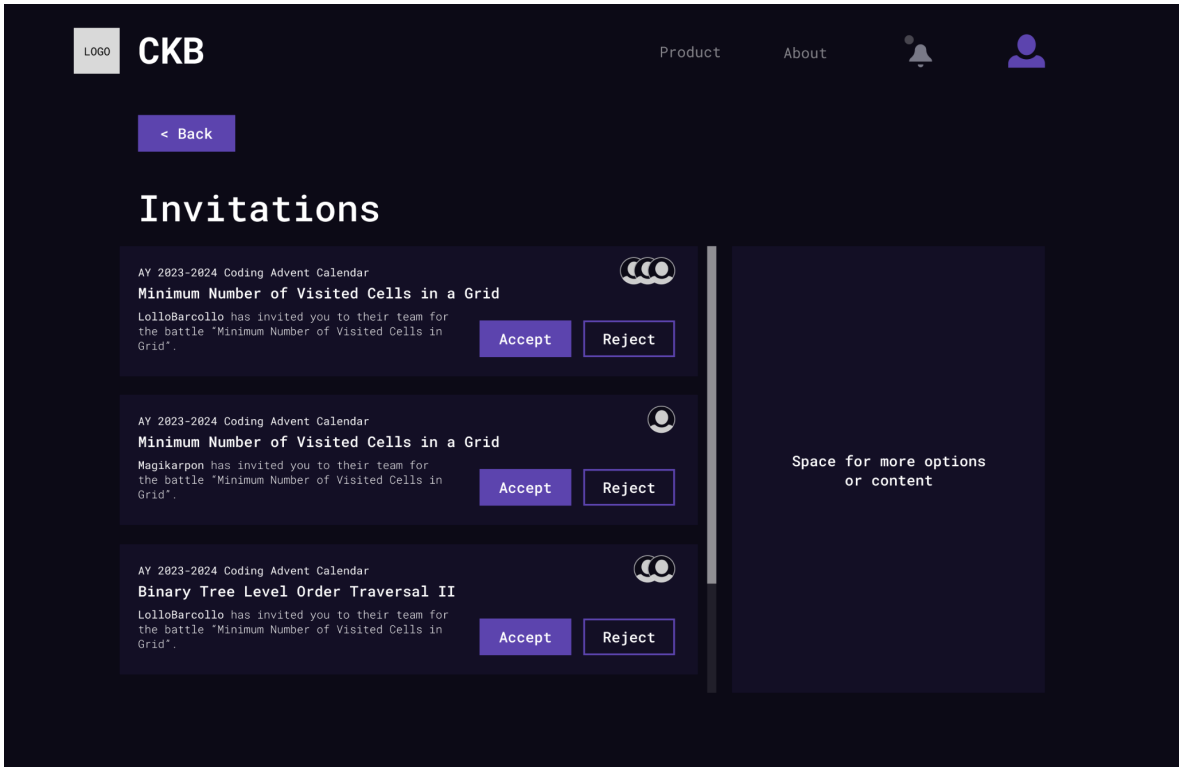
Landing page



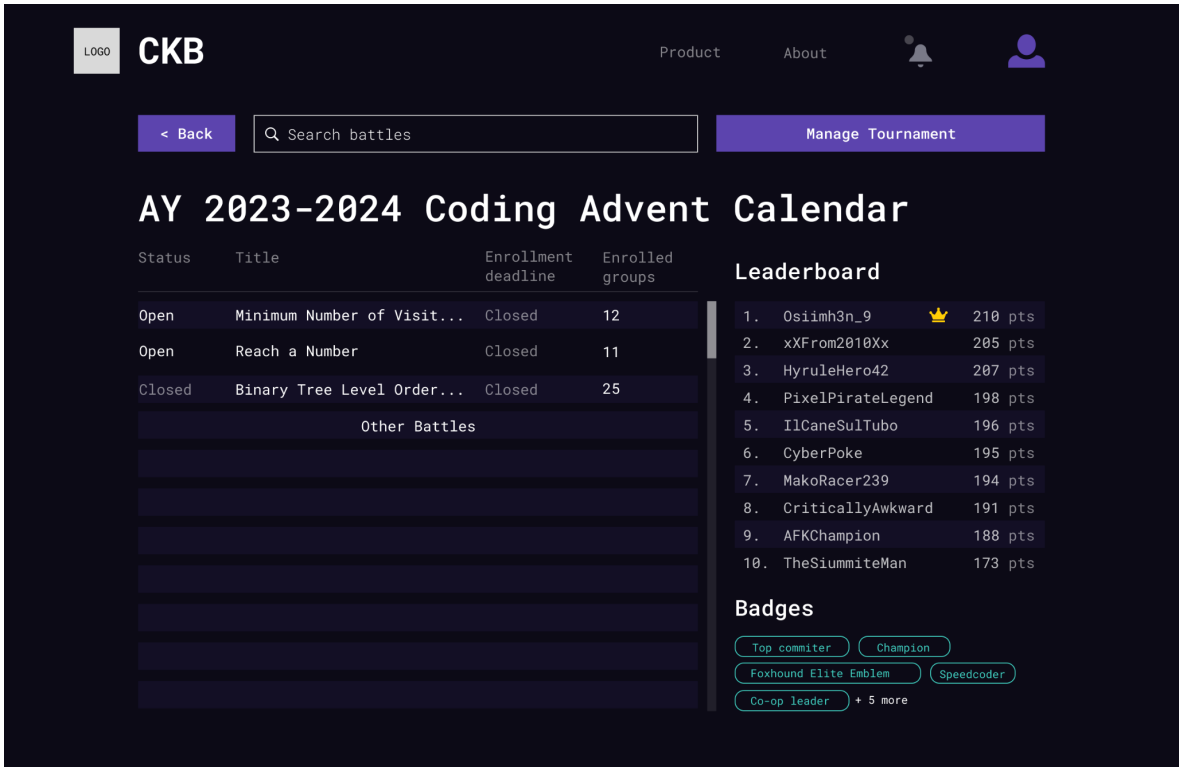
Account creation page



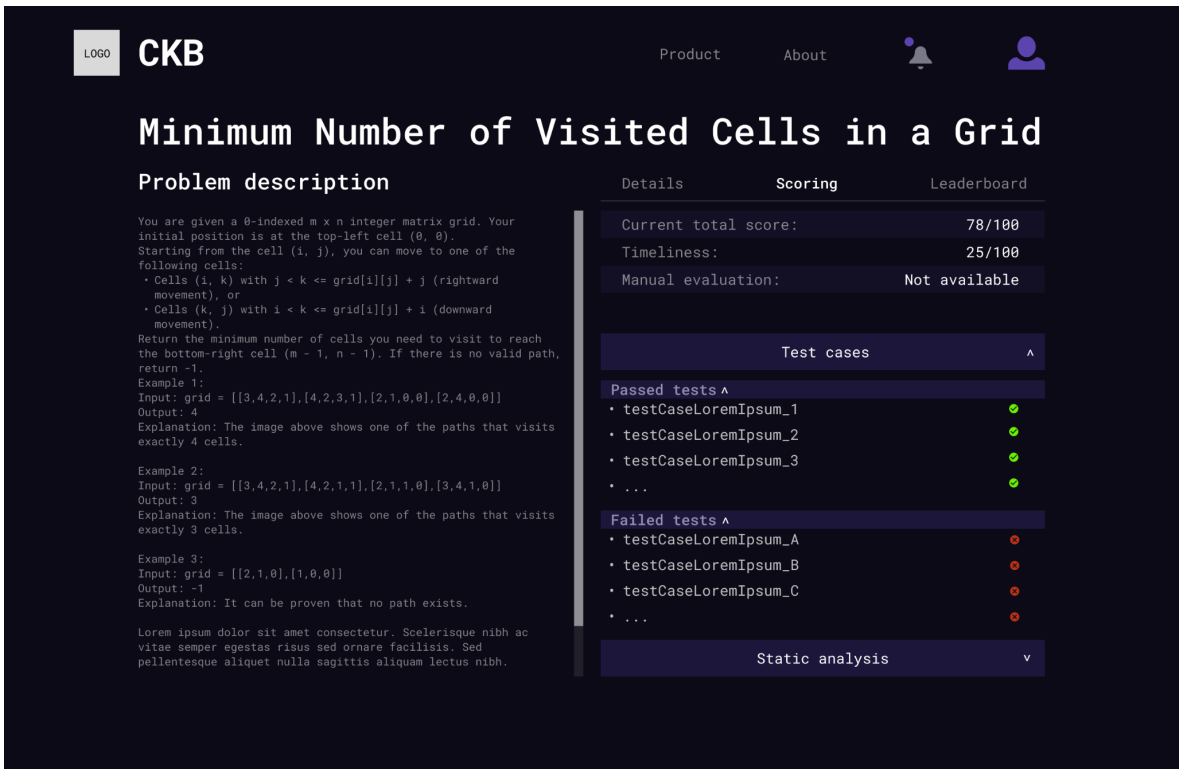
Student dashboard



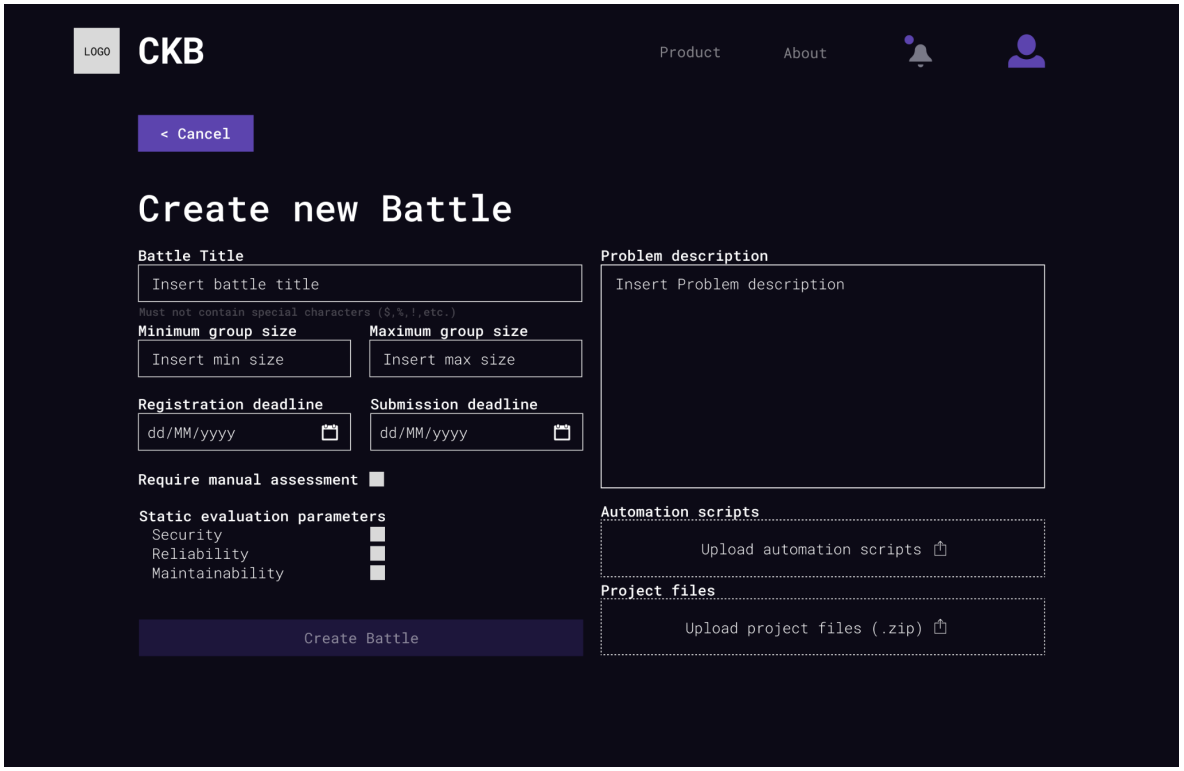
Student Invitations page



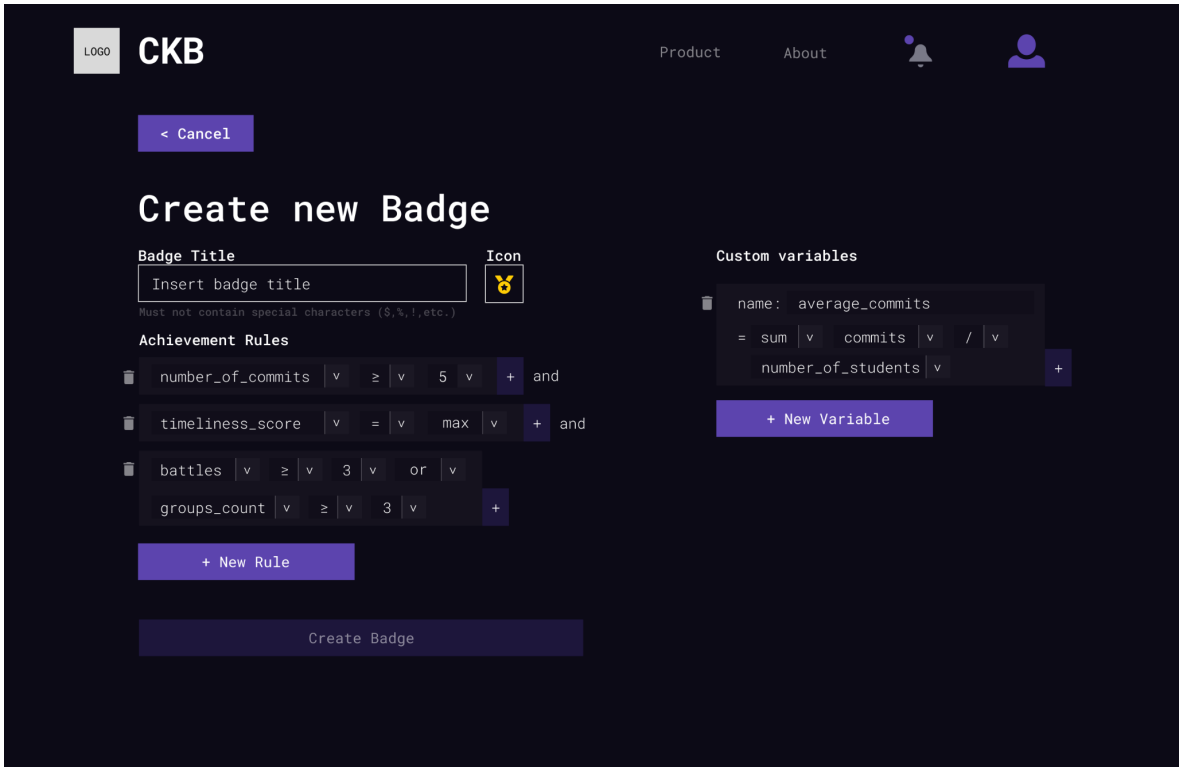
Tournament page (from the Educator's point of view)



Battle page from the Student's point of view



Battle creation page



Badge creation page

Please note that the styling and graphics for the views are meant to function as a mockup and not as the definitive representation of what the final product will end up being. Also, the presented layouts are shown as viewed from a laptop, and thus will be slightly different on mobile devices.

4. Requirements Traceability

In this section we show how different components work together to meet the requirements defined in the RASD document. For simplicity, we only mention the main components and not their sub-components, unless some sub-components are not involved in a requirement. In that case, all the sub-components that participate in the fulfillment of the requirement will be indicated. A more streamlined tabular representation is used at the end of this section.

[R1]: The system allows the user to sign up

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **User**
  ◦ **Builders**
  ▪ **UserAccountBuilder**
  ◦ **NotificationService**
• **CKBDatabase**
• **AuthenticationService**
• **AuthenticationDatabase**

[R2]: The system allows the registered user to log in

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **User**
• **AuthenticationService**
• **AuthenticationDatabase**

[R3]: The system allows the student to subscribe to tournaments
• **CKBWebView**

- **CKBApplication**
  - **APIHandler**
  - **CKBDataModel**
    - **User**
    - **Tournament**
- **CKBDatabase**

[R4]: The system allows the student to enroll in a battle

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **User**
  ▪ **Tournament**
  ▪ **Battle**
• **CKBDatabase**

[R5]: The system allows the student to invite another student to a battle, creating a group

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **User**
  ▪ **Tournament**
  ▪ **Battle**
  ▪ **Group**
  ▪ **Invite**
  ▪ **Notification**
  ◦ **Builders**
  ▪ **GroupBuilder**
  ◦ **NotificationService**
  ◦ **InviteHandler**
• **CKBDatabase**

[R6]: The system allows the educator to create a tournament

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **Tournament**
        - **Notification**
    - **Builders**
        - **TournamentBuilder**
    - **NotificationService**
- **CKBDatabase**

[R7]: The system allows the educator to create badges within a tournament

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **Tournament**
        - **Badge**
        - **Notification**
    - **Builders**
        - **BadgeBuilder**
        - **BadgeRuleBuilder**
    - **NotificationService**
- **CKBDatabase**

[R8]: The system allows the educator to create a battle within a tournament

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **Tournament**
        - **Battle**
        - **Notification**
    - **Builders**

        - **BattleBuilder**
    - **GitHubManager**
    - **NotificationService**
- **CKBDatabase**

[R9]: The system allows the educator to grant permissions to another educator

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **Tournament**
        - **User**
        - **Notification**
        - **Invite**
    - **NotificationService**
    - **InviteHandler**
- **CKBDatabase**

[R10]: The system allows the educator to make a manual assessment of the solution provided by the groups if specified in the scoring configurations

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **User**
        - **Group**
        - **Tournament**
        - **Battle**
- **CKBDatabase**

[R11]: The system must create the GitHub repository containing the code kata for the battle

- **CKBApplication**
    - **CKBDataModel**

        - **Tournament**
        - **Battle**
    - **GitHubManager**
- **CKBDatabase**

[R12]: The system must run tests and give an evaluation to the provided solutions

- **CKBApplication**
    - **CKBDataModel**
        - **Group**
        - **Tournament**
        - **User**
        - **Battle**
    - **CodeAnalyzer**
- **CKBDatabase**

[R13]: The system allows the user to see the current rank evolving during the battle

- **CKBWebView**
- **CKBApplication**
    - **APIHandler**
    - **CKBDataModel**
        - **Tournament**
        - **User**
        - **Battle**
        - **Group**
- **CKBDatabase**

[R14]: The system must update the personal tournament score of each student, that is the sum of all battle scores received in that tournament, at the end of each battle

- **CKBApplication**
    - **CKBDataModel**
        - **User**
        - **Tournament**
        - **Group**
        - **Battle**

- **EventHandler**
- **CKBDatabase**

[R15]: The system allows the educator who created the tournament to close it

- **CKBWebView**
- **CKBApplication**
  - **APIHandler**
  - **CKBDataModel**
    - **User**
    - **Tournament**
- **CKBDatabase**

[R16]: The system should assign a badge to one or more students at the end of the tournament if the students have fulfilled the badge's requirements to achieve it

- **CKBApplication**
  - **CKBDataModel**
    - **User**
    - **Badge**
    - **Tournament**
  - **EventHandler**
- **CKBDatabase**

[R17]: The system allows the user to view another student's badges

- **CKBWebView**
- **CKBApplication**
  - **APIHandler**
  - **CKBDataModel**
    - **User**
    - **Badge**
- **CKBDatabase**

[R18]: The system should automatically close battles after their submission deadline is

reached

• **CKBApplication**
  ◦ **CKBDataModel**
  ▪ **Battle**
  ◦ **EventHander**
• **CKBDatabase**

[R19]: The system allows users to search by battle and by tournament

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **Battle**
  ▪ **Tournament**
  ◦ **SearchHandler**
• **CKBDatabase**

[R20]: The system allows invited users to either accept or reject

• **CKBWebView**
• **CKBApplication**
  ◦ **APIHandler**
  ◦ **CKBDataModel**
  ▪ **User**
• **CKBDatabase**

[R21]: If configured, when the battle ends the system must notify the educators of the tournament that a manual evaluation is required to consolidate scores

• **CKBWebView**
• **CKBApplication**
  ◦ **CKBDataModel**
  ▪ **User**
  ▪ **Tournament**
  ▪ **Notification**

- **NotificationService**
- **CKBDatabase**

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21
CKBWebView	X	X	X	X	X	X	X	X	X	X			X		X				X	X	X
CKBApplication	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
APIHandler	X	X	X	X	X	X	X	X	X	X			X		X				X	X	
RequestsDispatcher	X	X	X	X	X	X	X	X	X	X			X		X				X	X	
RequestsHandler	X	X	X	X	X	X	X	X	X	X			X		X				X	X	
CKBDataModel	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Battle				X	X			X		X	X	X	X	X				X	X		
Tournament			X	X	X	X	X	X	X	X	X	X	X	X	X	X			X		X
User	X	X	X	X	X				X	X		X	X	X	X	X	X			X	X
Badge							X									X	X				
Group					X					X		X	X	X							
Notification					X	X	X	X	X												X
Invite					X				X											X	
Builders	X				X	X	X	X													
UserAccountBuilder	X																				
TournamentBuilder						X															
BattleBuilder								X													
GroupBuilder					X																

BadgeBuilder							x													
BadgeRuleBuilder							x													
NotificationService	x				x	x	x	x	x											x
InviteHandler					x				x											
SearchHandler																		x		
CodeAnalyzer											x									
ECARunner											x									
TestRunner											x									
GitHubManager								x			x									
EventHandler													x		x		x			
CKBDatabase	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
AuthenticationService	x	x																		
AuthenticationDatabase	x	x																		

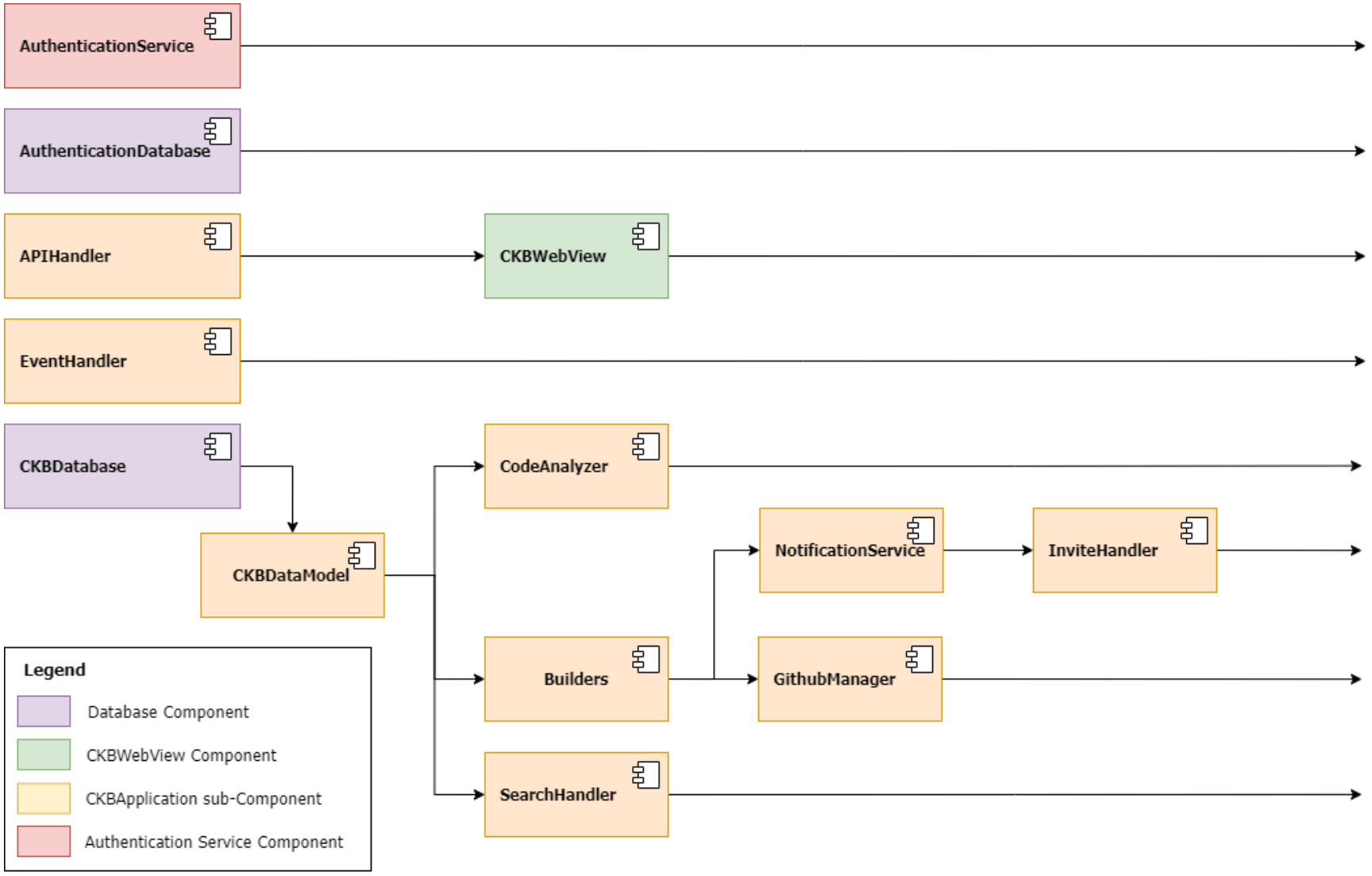
5. Implementation, Integration, and Test Plan

For the implementation and testing of the application we decided to embrace a Test-Driven Development (TDD) approach combined with a bottom-up strategy.

More precisely, we plan to use the Design Document as a foundation for crafting test cases that outline the expected behavior of the application prior to any code implementation. These initial tests are deliberately designed to fail if no corresponding implementation exists. This is done to guide the development by providing a clear roadmap of functionality requirements to fulfill. Using this approach correctly, makes sure that every component is thoroughly tested as it is built and therefore improves overall robustness of the system.

Integration testing is performed in the same way with the addition of a Continuous Integration and Deployment (CI/CD) pipeline running on the remote repository used as a shared codebase. This pipeline runs a series of automated integration tests every time someone wants to add to the production codebase, allowing everyone working on the project to see the issues that may arise and that could depend on an already tested and deployed component.

5.1. Development Plan



Thanks to the chosen architecture, specifically the separation between business logic and interface, simultaneous development of multiple components becomes easier. For example, by defining an interface for the application using OpenAPI specification, the front-end (the CKBWebView component) can be developed concurrently with the back-end with minimal use of stub components. Indeed, utilizing tools such as Postman and Swagger allows us to conduct tests without requiring the complete implementation of the application logic.

The authentication service and its relative database can also be simultaneously developed with the rest of the application.

6. Effort Spent

6.1. Shared effort

This is the effort spent by all the members of the group during various meetings and/or group discussions.

Activity	**Effort spent**
Initial setup and task allocation	1,30h
Meeting for component diagrams	1h
Total shared effort	**2,30h**

5.2. Individual Effort

Bersani Michele	
Activity	**Effort spent**
Creation of components and subcomponent	2h
Component diagram	4h
Sequence diagram	11h
Update and upload sequence diagram	7h
Document review	2h
Total individual effort	**26h**

Chiappini Paolo	
Activity	**Effort spent**
UI design	9h
Review	1h
UI description and Navigation Diagram	1h
Review and draft for Sequence Diagrams	2h
Component interfaces	6h
Components review	3h
Document and diagrams review	8h
Total individual effort	**30h**

Fraschini Andrea	
Activity	**Effort spent**
Document setup	2h
Main components definitions and architecture	3h
Cloud architecture	3h
Component deployment diagram	4h
Requirement traceability	2h
Requirement mappings	2h
Implementation, Integration and Test Plan	4h
Revisions	5h
Total individual effort	**26h**

7. References

- Microsoft Azure cloud services: https://azure.microsoft.com/
- Postman: https://www.postman.com/
- Swagger: https://swagger.io/