SaaS Application Project: Tournament Organizer

Team Hairless Cats

Andrew Cheah, Bryce Wilson, Leah Jo, Chunga Lee,
Munkhtur Myagmar (Mogi!), Joshua Tan

Internal Design Document

Version 1.1
April 19th, 2022

Contents

I. Introduction

We may have new entities in the back-end source code due to the refactoring in the efforts to add new functions for our SaaS product. However, some of the functions these entities should provide (e.g. Bracket style tournament) are not updated to connect on the front-end for this Warranty Period submission. Since they are excluded in this submission's design itself, they are not updated for this Design document.

This project aims to provide a SaaS product to facilitate tournament organization and play for Amazon staff. It allows fluid administration of tournament games and allows players to easily set their availability.

This document outlines our team's internal design choices that we have made for building the application. We will focus on describing the technical requirements when designing to balance system availability, scalability, and maintenance. Furthermore, the system design assumptions include a secure database and a limited number of users.

We will specify important details including the technology stack that we are using, the software architecture and the choice of complex algorithms. The details are technical in nature, but serve any layman and provide a reference for the readers in understanding the mechanics of our application. In doing this, we hope to make information easily accessible, accelerate the learning process for a new developer maintaining the application, and facilitate greater quality assurance testing.

II. Programming Environment

Table 1. Programming Tools, Languages and associated versions.

Programming Tool/Language	Version
Integrated development environment (IDE)	Open to developer choice
Java	17+ (Amazon Corretto)
Java Spring Boot	2.6.3+
ReactJS	17+
PostgreSQL	13
Git	2.35+

The table above summarizes the programming tools and languages that we would be primarily using for this project. As required by our project sponsor Amazon, our backend would be built on top of Java Spring Boot framework. We choose Spring Boot as it is a reputable and widely used web framework. Most importantly, it helps to focus on building the core features of the application without worrying too much on the low-level technical details.

For our frontend, we will use the popular ReactJS library. ReactJS is well supported and is relatively easy to use. Using ReactJS would allow us to easily have support for popular browsers such as Chrome, Safari and Firefox. Additionally, it is easy to integrate compatible CSS libraries such as Bootstrap 4 or Tailwind CSS to design the web application for both desktops and mobile devices.

For our database, we would be using PostgreSQL. It is a popular relational database and our team members are comfortable with using it as well. While we could have opted for a non-relational database, we decided to keep things simple and stick to what we knew.

Each developer on the team is free to use their IDE of choice and our main version control system would be Git.

III. Production and Test environments

a. Backend

Our Java-based backend and PostgreSQL database would be hosted on Amazon's EC2 and RDS services respectively. For testing, we would start up the backend locally on our machines in test mode; this allows us to trigger asynchronous events manually.

For backend testing, we used Springboot testing environment to build our unit tests.

b. Frontend

Our application frontend would be hosted on Amazon's S3 service. For testing, we would set up a simple python webserver on our machines. This test web server would allow us to serve static files quickly and view any local changes without affecting anyone else.

IV. Software Architecture

We have designed a software architecture for us to begin with. For starters, we will be assuming that the tournament will be a round-robin system and each player only needs to play with every other player once. Matches will be 1-on-1. This tournament-style is simple enough and does not add any other unnecessary complexities such as team matches or bracket-style competitions. We will continue to add different tournament styles and rulesets as we develop this application further.

Figure 1 shows our class diagram for this very simple setup. Figures 2, 3 and 4 are sequence diagrams showcasing how the various classes interact with one another to create tournaments, add players, and create schedules.

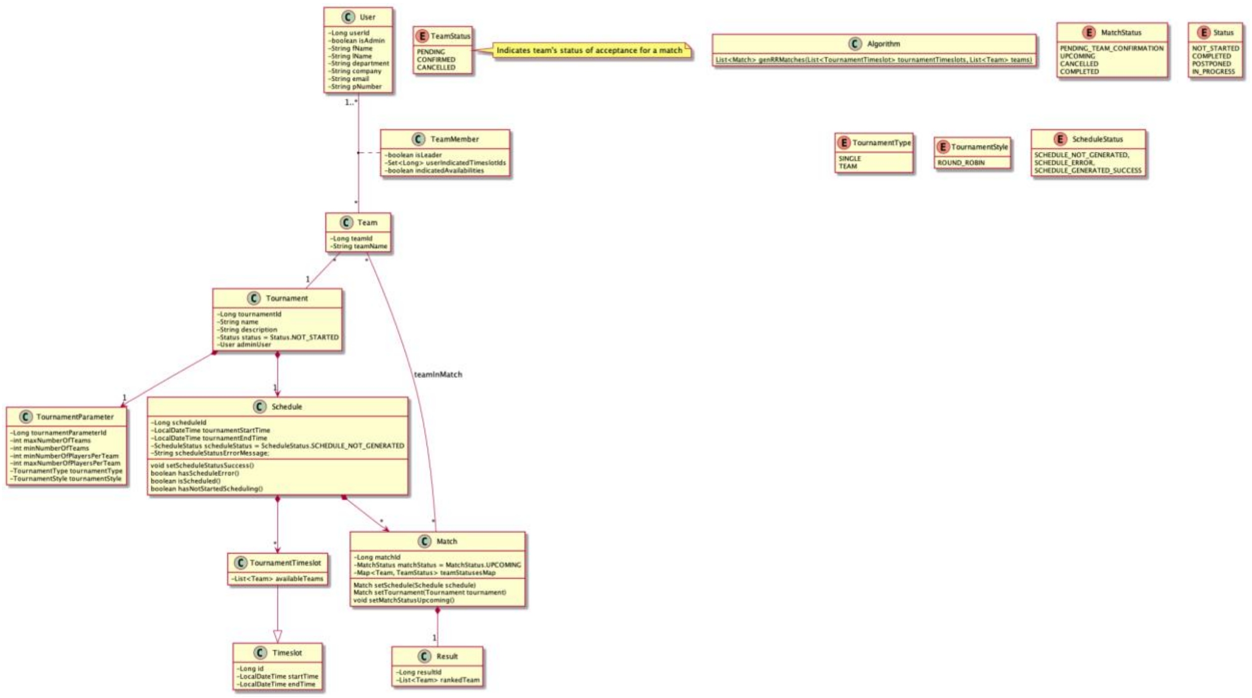
User
-Long userId
-boolean isAdmin
-String fName
-String lName
-String department
-String company
-String email
-String pNumber
TeamStatus
PENDING
CONFIRMED
CANCELLED
Indicates team's status of acceptance for a match
Algorithm
List<Match> genRRMatches(List<TournamentTimeslot> tournamentTimeslots, List<Team> teams)
MatchStatus
PENDING_TEAM_CONFIRMATION
UPCOMING
CANCELLED
COMPLETED
Status
NOT_STARTED
COMPLETED
POSTPONED
IN_PROGRESS
1..*
TeamMember
-boolean isLeader
-Set<Long> userIndicatedTimeslotIds
-boolean indicatedAvailabilities
TournamentType
SINGLE
TEAM
TournamentStyle
ROUND_ROBIN
ScheduleStatus
SCHEDULE_NOT_GENERATED,
SCHEDULE_ERROR,
SCHEDULE_GENERATED_SUCCESS
*
Team
-Long teamId
-String teamName
*
1
*
Tournament
-Long tournamentId
-String name
-String description
-Status status = Status.NOT_STARTED
-User adminUser
teamInMatch
1
1
TournamentParameter
-Long tournamentParameterId
-int maxNumberOfTeams
-int minNumberOfTeams
-int minNumberOfPlayersPerTeam
-int maxNumberOfPlayersPerTeam
-TournamentType tournamentType
-TournamentStyle tournamentStyle
Schedule
-Long scheduleId
-LocalDateTime tournamentStartTime
-LocalDateTime tournamentEndTime
-ScheduleStatus scheduleStatus = ScheduleStatus.SCHEDULE_NOT_GENERATED
-String scheduleStatusErrorMessage;
void setScheduleStatusSuccess()
boolean hasScheduleError()
boolean isScheduled()
boolean hasNotStartedScheduling()
*
*
*
TournamentTimeslot
-List<Team> availableTeams
Match
-Long matchId
-MatchStatus matchStatus = MatchStatus.UPCOMING
-Map<Team, TeamStatus> teamStatusesMap
Match setSchedule(Schedule schedule)
Match setTournament(Tournament tournament)
void setMatchStatusUpcoming()
1
Timeslot
-Long id
-LocalDateTime startTime
-LocalDateTime endTime
Result
-Long resultId
-List<Team> rankedTeam

Figure 1. Class Diagram

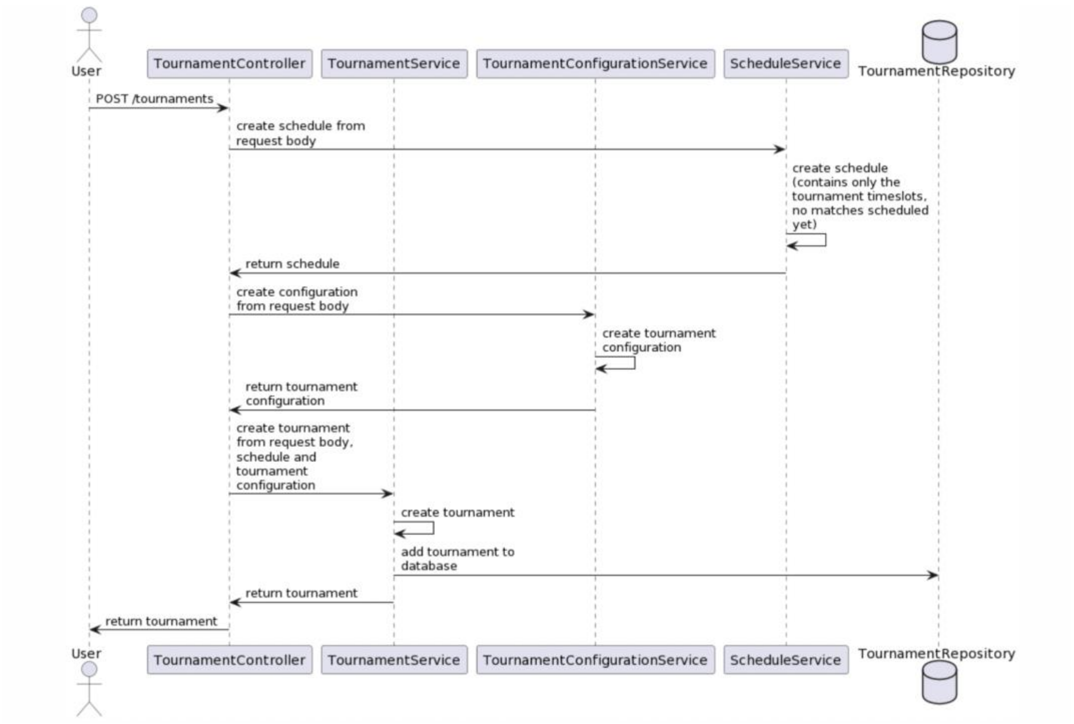
User
TournamentController
TournamentService
TournamentConfigurationService
ScheduleService
TournamentRepository
POST /tournaments
create schedule from request body
create schedule (contains only the tournament timeslots, no matches scheduled yet)
return schedule
create configuration from request body
create tournament configuration
return tournament configuration
create tournament from request body, schedule and tournament configuration
create tournament
add tournament to database
return tournament
return tournament

Figure 2. Create Tournament Sequence Diagram

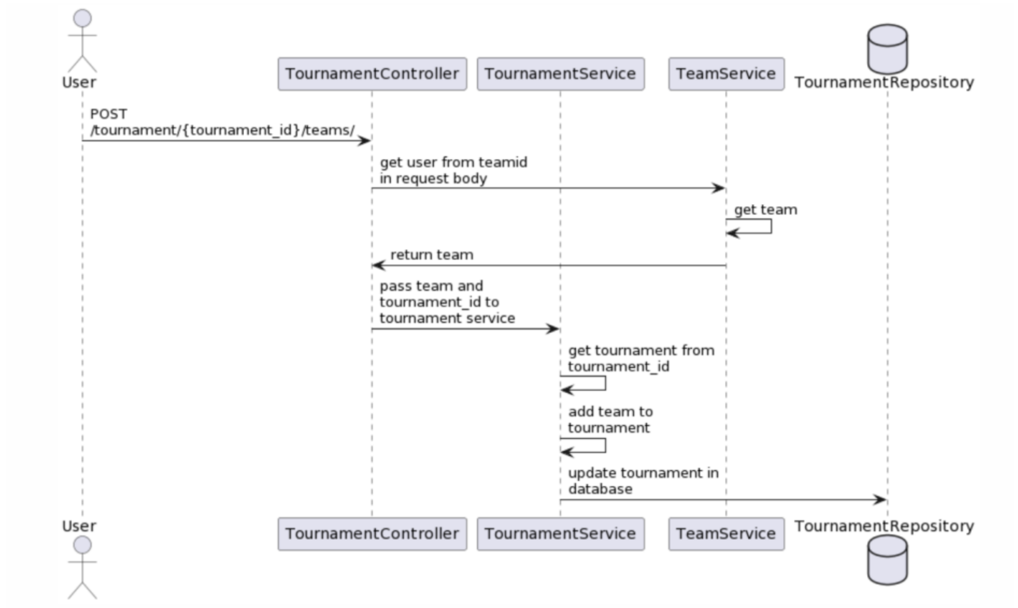
User
TournamentController
TournamentService
TeamService
TournamentRepository
POST /tournament/{tournament_id}/teams/
get user from teamid in request body
get team
return team
pass team and tournament_id to tournament service
get tournament from tournament_id
add team to tournament
update tournament in database

Figure 3. Team Join Tournament Sequence Diagram

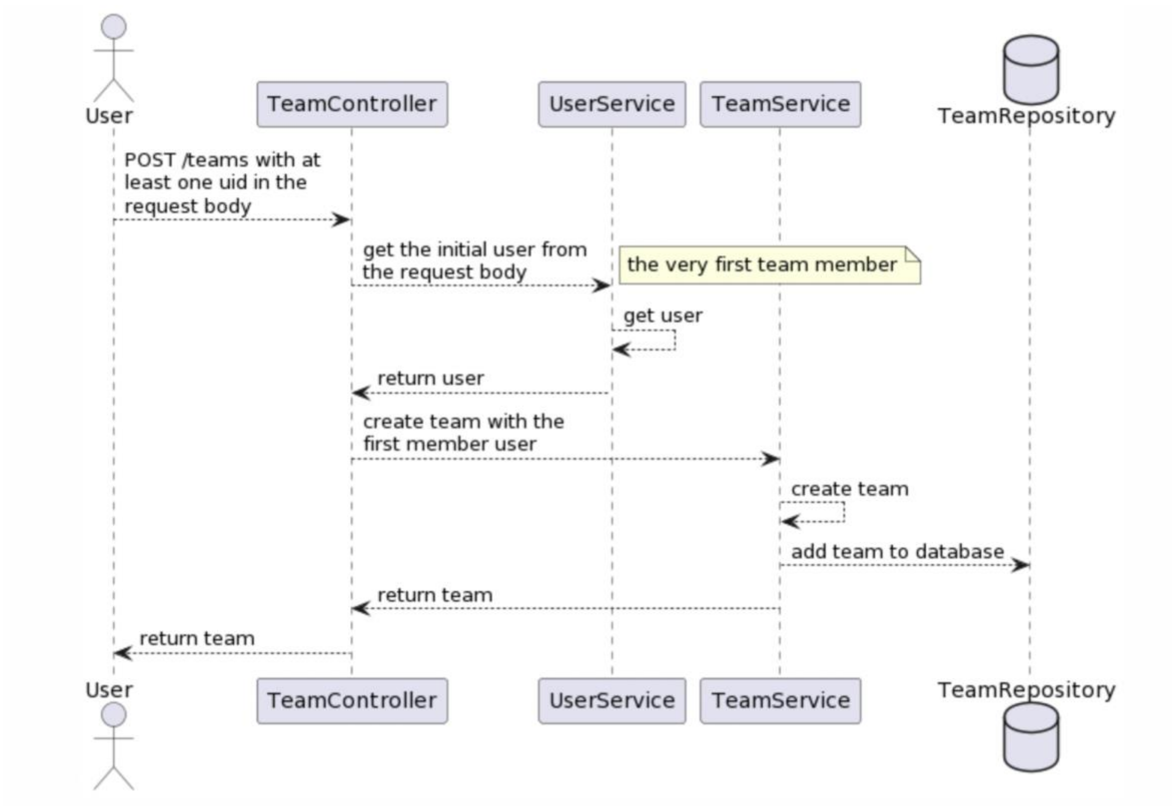
User
TeamController
UserService
TeamService
TeamRepository
POST /teams with at least one uid in the request body
get the initial user from the request body
the very first team member
get user
return user
create team with the first member user
create team
add team to database
return team
return team

Figure 4. Create Team Sequence Diagram

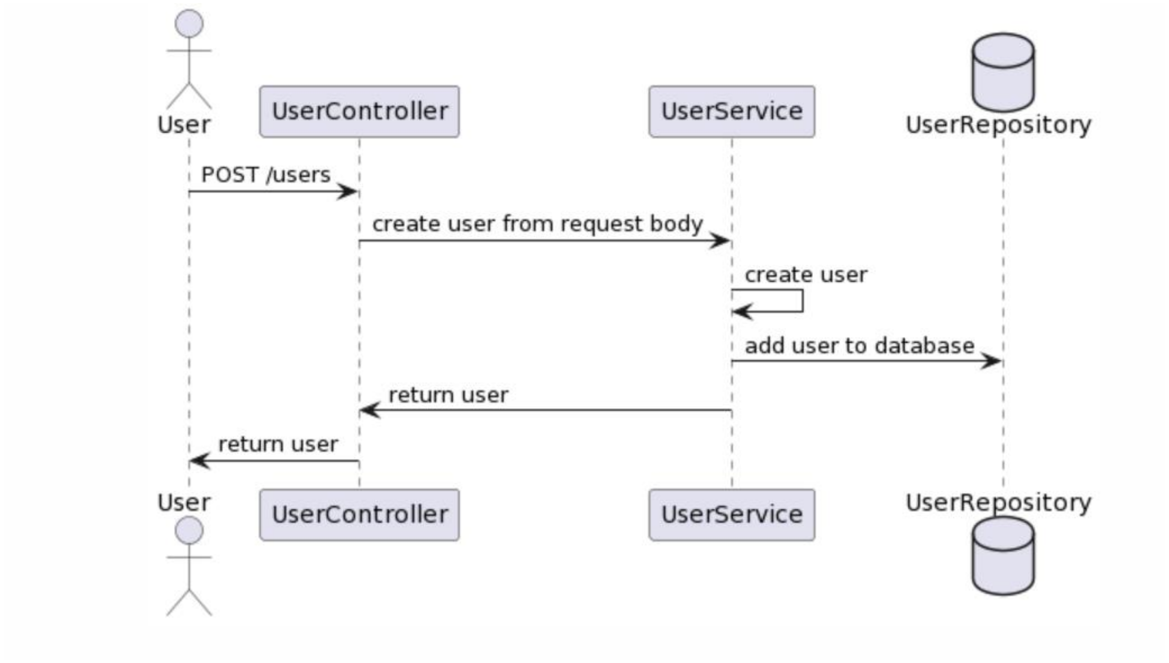

Figure 5. Create User Sequence Diagram

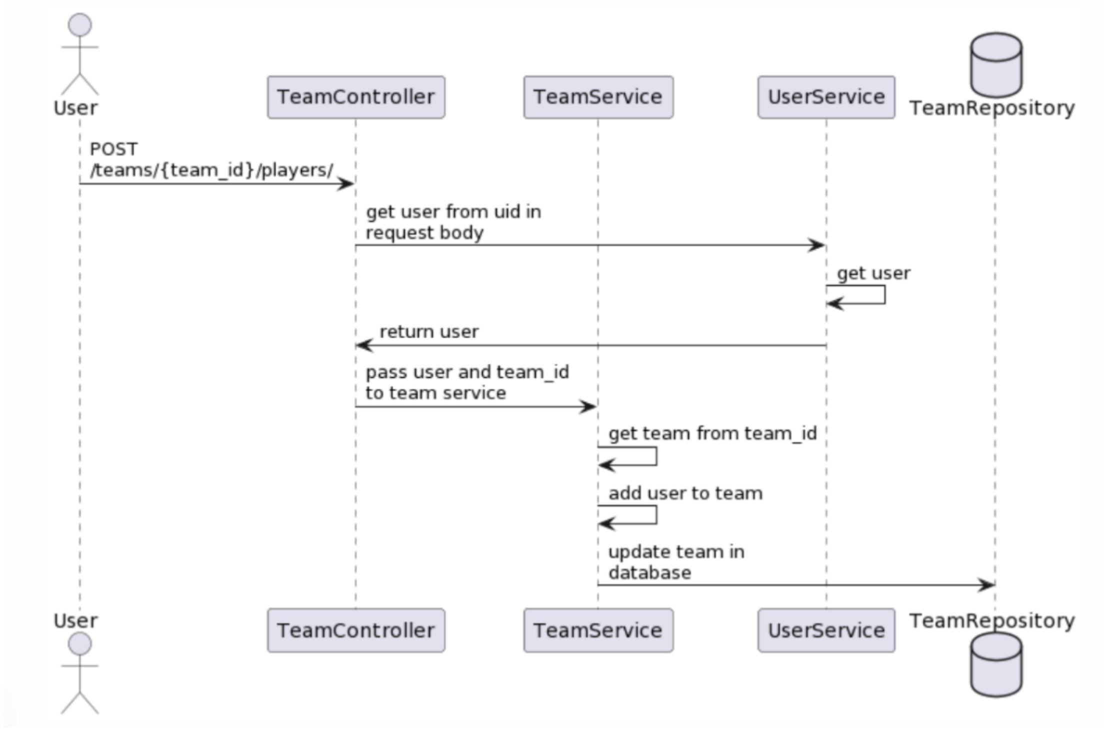

Figure 6. User Join Team Sequence Diagram

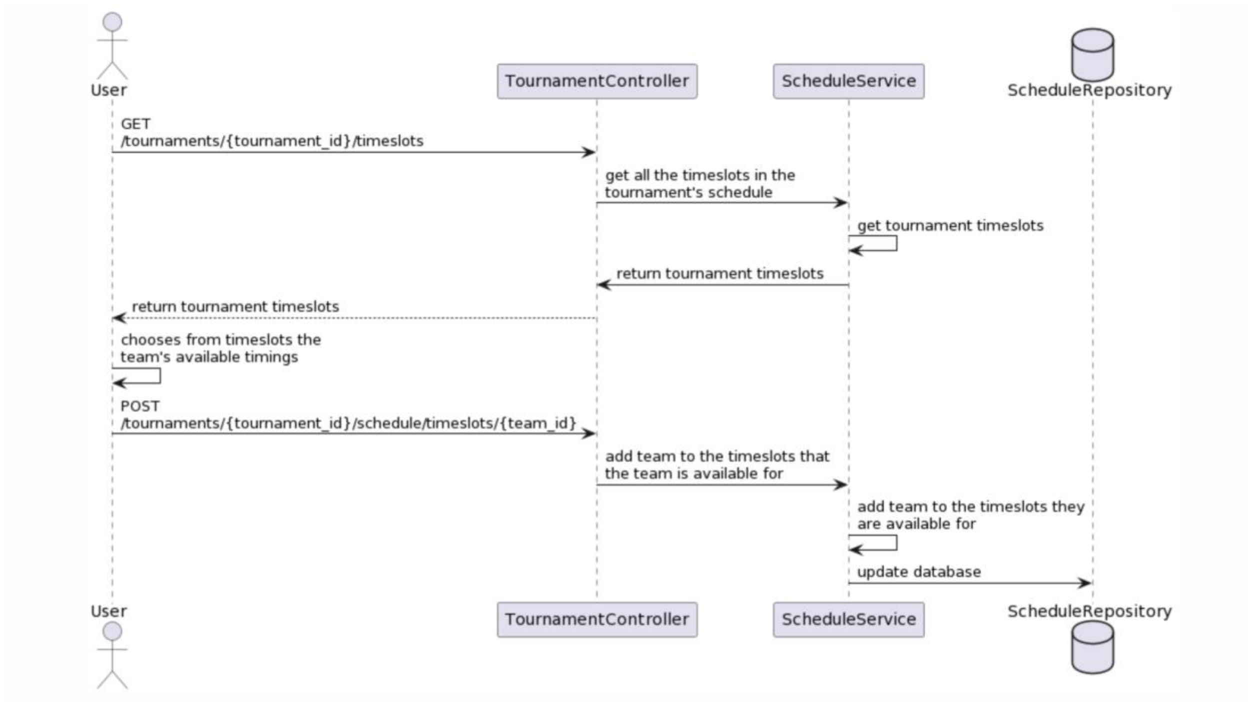
User
TournamentController
ScheduleService
ScheduleRepository
GET /tournaments/{tournament_id}/timeslots
get all the timeslots in the tournament's schedule
get tournament timeslots
return tournament timeslots
return tournament timeslots
chooses from timeslots the team's available timings
POST /tournaments/{tournament_id}/schedule/timeslots/{team_id}
add team to the timeslots that the team is available for
add team to the timeslots they are available for
update database
User
TournamentController
ScheduleService
ScheduleRepository

Figure 7. Team Adding Tournament Availability

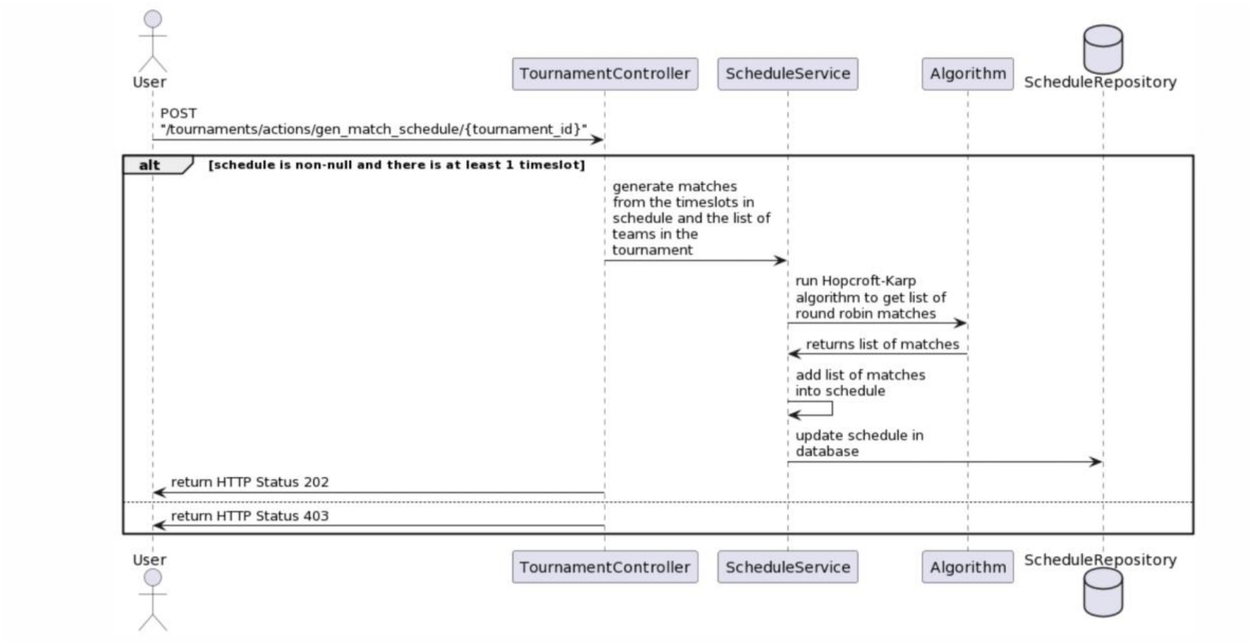
User
TournamentController
ScheduleService
Algorithm
ScheduleRepository
POST
"/tournaments/actions/gen_match_schedule/{tournament_id}"
alt
[schedule is non-null and there is at least 1 timeslot]
generate matches from the timeslots in schedule and the list of teams in the tournament
run Hopcroft-Karp algorithm to get list of round robin matches
returns list of matches
add list of matches into schedule
update schedule in database
return HTTP Status 202
return HTTP Status 403

Figure 8. Generating Round Robin Match Schedule Sequence Diagram

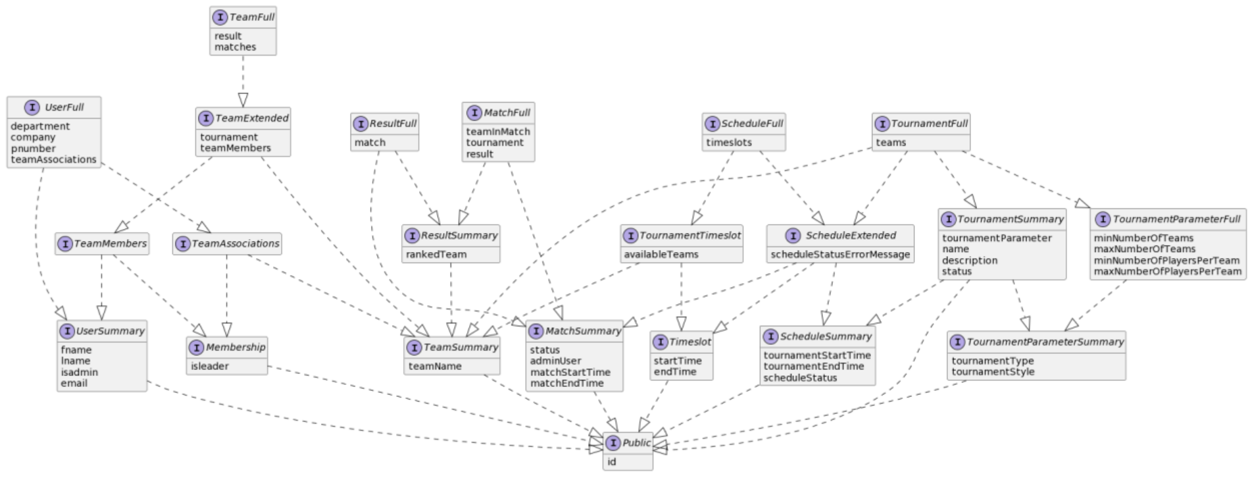

Figure 9. Views

V. Data Design

For this project, we will create a relational database written in PostgreSQL stored on Amazon's RDS service.

During the implementation process, we have utilized Spring Boot to create functional dependencies between entities. It hides away our relational instances and tables to allow programmers to worry less about the data dependencies. Therefore, visualizing our relationships is opaque, but we have continued to build structures on top of our initial data design.

Refer to Figure 5. ER Diagram for a visual representation of our entities that was planned initially. In the diagram, participation restraints are shown in bold, as well as the absence of arrows indicate a many-sided relationship. For example, a user can administer many tournaments, and a tournament can be administered by one or more users.

Refer to Table 2. to ascertain our specific entities which contain the table definitions.

- Primary keys are underlined.
- **Foreign keys** are bolded.

Refer to Table 3. to view our functional dependencies in an itemized form.

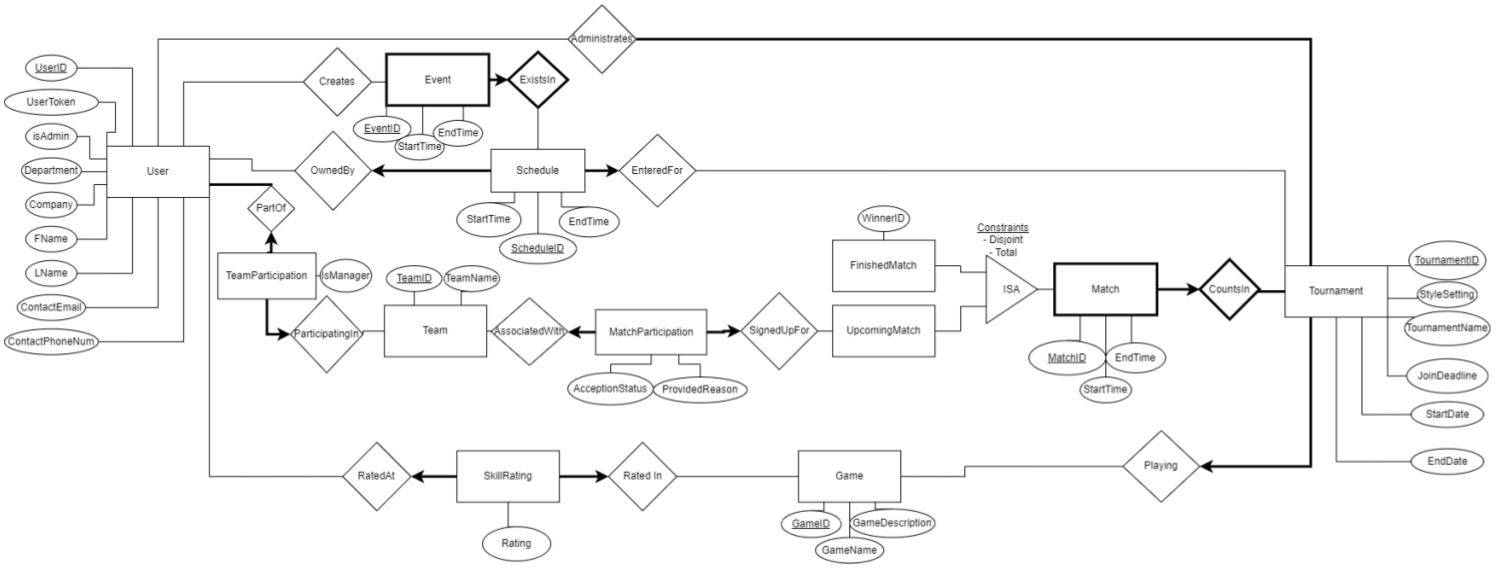

Figure 5. Initial ER Diagram Version 0.1 (Notation as taught in CPSC 304)

Table 2. Table Definitions in relational schema with keys and constraints identified. Primary keys are bolded, and foreign keys underlined

User(user_id:String, UserToken:String, isAdmin:boolean, department:String, company:String, fName:String, lName:String, email:String, pNumber:String) - Primary key is user_id - UserToken, isAdmin, department, company, fName, lName, email, pNumber are NOT NULL
Schedule(schedule_id:long, **tournament_id:long**, tournamentStartTime:date, tournamentEndTime:date) - Primary key is schedule_id - tournament_id is foreign key referencing Tournament(tournament_id) - tournament_id, tournamentStartTime, tournamentEndTime are NOT NULL
Team(teamID:long, **tournament_id:long**, teamName:String)) - Primary key is team_id - tournament_id is foreign key referencing Tournament(tournament_id) - tournament_id, teamName is NOT NULL
TeamMember(**user_id:long**, **team_id:long**, isLeader:boolean)) - Primary key is (user_id, team_id) - user_id is foreign key referencing User(user_id) - team_id is foreign key referencing Team(team_id)
Tournament(tournamentID:long, **schedule_id:long**, **user_id:String**, name:String, StyleSetting:TournamentStyle**,** , description:String, status:TournamentStatus, tournamentParameter:TournamentParameter) - Primary key is tournamentID - user_id is a foreign key referencing User(user_id) - schedule_id is a foreign key referencing Schedule(schedule_id) - schedule_id, user_id, StyleSetting**,** name, description, tournamentParameter are NOT NULL
Timeslot(timeslot_id:long, startTime:date, endTime:date, **schedule_id:long**) - Primary key is (timeslot_id, schedule_id) - schedule_id is a foreign key referencing Schedule(schedule_id) - schedule_id, startTime, endTime are NOT NULL
Match(**match_id:long**, **tournament_id:long**, **result_id:long**, **schedule_id:long,** matchStatus:MatchSatus, matchStartTime:date, matchEndTime:date) - Primary key is (match_id) - user_id is a foreign key referencing User(user_id) - tournament_id is foreign key referencing Tournament(tournament_id)

- schedule_id is a foreign key referencing Schedule(schedule_id)
- result_id is a foreign key representing a Result(result_id)
- Tournament_id, result_id, schedule_id, matchStatus, matchStartTime, matchEndTime are NOT NULL

TeamsInMatch(**match_id:long**, **team_id:long**, teamStatus:TeamStatus)

- Primary key is (match_id, team_id)
- team_id is foreign key referencing Team(team_id)
- match_id is a foreign key referencing Match(match_id)
- teamStatus is NOT NULL

Result(result_id:long, **match_id:long**)

- Primary key is (result_id)
- match_id is a foreign key referencing Match(match_id)
- Match_id is NOT NULL

TeamResultMap(**result_id:long, team_id:long**, rank:integer)

- Primary key is (result_id, team_id)
- rank is NOT NULL

Availability(**timeslot_id:long**, **user_id:String**, **team_id:long**)

- Primary key is (timeslot_id, user_id)
- Timeslot_id is foreign key referencing Timeslot(timeslot_id)
- user_id is a foreign key referencing User(user_id)
- team_id is foreign key referencing Team(team_id)

Table 3. Functional Dependencies

With Attribute(s) →	May Determine Attribute(s)
user_id	UserToken, isAdmin, department, company, fName, lName, email, pNumber
schedule_id	tournament_id, tournamentStartTime, tournamentEndTime
teamID	tournament_id, teamName
user_id, team_id	isLeader:boolean

tournamentID	schedule_id, user_id, name, StyleSetting, , description, status, tournamentParameter
timeslot_id	startTime, endTime, schedule_id
match_id	tournament_id, result_id, schedule_id, matchStatus, matchStartTime, matchEndTime
match_id, team_id	teamStatus
result_id	match_id
result_id, team_id	rank

API Design

Based on the *Requirement Traceability Matrix* from *Project Requirements*, we provide REST API that would be applicable for developing our minimum viable product. High priority functions were considered first when designing API.

The API Design is fully up to date and accessible from the github repository.
link: https://github.com/CPSC319-Winter-term-2/Hairless-Cat-AWS-Backend/tree/master/docs

VI. Algorithms

There are primarily 2 algorithms used in our application.

The first is used for aggregating team member availability. The algorithm is a simple iteration through all the team members and keeping track of the tournament time slots they are available. Then the algorithm would pick out from these time slots, those which satisfy the minimum number of players needed to play a match. These time slots would then be identified as the team's available time slots.

The second algorithm is the scheduling algorithm for a simple round robin matching system. In a round robin matching system, every pair of teams would need to play a match. But they can only play if both teams are free.

Based on this, we can model our situation as a bipartite graph. On the left partition, every vertex m is a match that needs to be played between team i and team j, for all teams i, j. On the right partition, every vertex ts is a time slot that is available for a match. An edge e between any m and ts means that the teams playing in match m are available to play at the time slot ts.

Solving the scheduling problem can be simply reduced to finding the *maximum cardinality matching* in the bipartite graph. A maximum cardinality matching is simply a set of as many edges as possible with the property that no two edges share an endpoint.

Assuming such a matching exists, then every edge in the solution set indicates that the match m will play at that time slot ts. If there is a match m that is not adjacent to any time slot ts in the solution set, then our scheduling algorithm has failed to find a timing for m to be played based on the given team availabilities.

In such a scenario, teams will have to either re-indicate their availabilities or the tournament administrator can arbitrarily assign the unassigned matches a time slot.

To build the bipartite graph needed for our algorithm, we simply create vertices for all $\frac{n}{2}$ matches that need to be played in the left partition.

Next, we create vertices for all time slots in the right partition of the graph. Finally, we run our algorithm to solve the maximum cardinality matching problem.

Once we have the solution, we simply create matches based on the edges in the solution set. The algorithm we used for solving the scheduling problem would be the *Hopcroft-Karp* algorithm. The pseudocode for the algorithm is taken from Hopcroft–Karp Algorithm for Maximum Matching | Set 1 (Introduction) - GeeksforGeeks and reiterated here as follows:

Pseudocode:

```
1. Initialize Maximal Matching M as empty.
2. While there exists an Augmenting Path p
    a. Remove matching edges of p from M and add not-matching edges of p to M
       (This increases size of M by 1 as p starts and ends with a free vertex)
3. Return M.
```

VII. Security

To facilitate user logins, we will be relying on Amazon's Cognito service. It is very scalable and supports sign-in with social identity providers, such as Apple, Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0 and OpenID Connect.

VIII. Notable Trade-Offs

Table 4. Design Choices with advantages, and disadvantages

Design Choices	**Advantage**	**Disadvantage**
Amazon Cognito	Security: Using an external service means that we do not need to worry as much about the complexity of properly storing user passwords.	Complexity: It is more complex to implement than simple login
Need Amazon email to log in	Security: ensures that only company employees can become users and access the tournaments.	Time/Additional efforts: It needs an additional process to check the employee has an amazon email in which increases time.
Providing Admin code	Security/Quality: ensures that only administrators who have an admin access code that the application provides can create an admin account.	Time: it needs more time to get and verify the code. Additional efforts: It needs to add processes (functions) to check the verification code.
Providing the functionality to verify the game results	Quality: ensures honest sportsmanship and accurate representation of live game results. Performance: ensures the accurate generation of the next match and schedule based on the game results.	Time: Time increases to have more steps to get the result of the tournaments.
Providing many Tournaments styles	Quality/Performance: By providing more tournaments styles, it increases the quality of the program by helping the admin create and provide various types of tournaments for employees.	Time: Time will increase to create the tournaments. Data Capacity: More data used to save more tournaments types

At least 10 Concurrent users	Performance: ensure that multiple users can access the platform at the same time.	Time/Performance: It may increase running time because concurrent users try to use the application.
At least optimize for a maximum user base of 300	Performance: ensures that the application works with multiple users. User Capacity: It guarantees lots of employees can register and use the app for participation.	Management: Without limiting the number of the users, it can be hard to
Response time limitation: Response time for searches/filters to be less than 4 seconds	Performance: It makes the user not wait a long time to get a response.	System Risks (refer to *X.Notable Risks*)

IX. Notable Risks

Table 5. Requirements with potential risks corresponding to potential solutions

System requirements	**Potential risks**	**Potential solution**
Response time limitation: Response time for searches/filters to be less than 4 seconds	What happens if we go over the 4 seconds response time? Users may be upset if it takes too long. 1. Other connection errors to the server may have occurred. 2. Algorithm is too slow	1. Manage Database: clean up old tournament data. (Delete matching and scheduling data after the match. It leaves just game results and record data.)
Support in multi-device and multi-platform	It does not show up properly in some browsers (ex. Working in chrome but not in Safari) or devices.	Using React Framework properly as react supports multiple browsers and device screen sizes. Don’t use browser-specific CSS unless it has a fallback.
Matching Algorithm Process for scheduling	1. It may take a long time to run synchronously with many users and complex schedules. 2. There may be no schedule that works.	1. The algorithm can run asynchronously. 2. If there is no schedule that works, the admin can manually set game times or change who is matched up. Alternatively, the algorithm can be given permission to change matching if match competitors have disjoint schedules.