COMP 3004 - Deliverable #3
System Architecture and Design

`Brackit` - Mobile Tournament Bracket Creation

Metadata

Team / App Name: `Brackit`

Team member names

Name	Student ID
Jaime Herzog	101009321
Suohong Liu	101002340
Xiyi Liu	101004577
Alex Trostanovsky	100984702

Contents

Architecture

1.1 Description

1.1.1 Functional & Non-Functional Requirements

`Brackit` addresses an urgent need by tournament organizers and attendants to visualize, manage, and interact with double elimination brackets on their mobile devices. At a high level, we committed to developing a product that will meet the following **functional requirements**:

1. Tournament Organizers (TO's) can create, host, maintain, and visualize double elimination brackets.
2. Registered `Brackit` Users, as well as Guests, can use the application to join created tournaments.
3. `Brackit` will store and maintain user profiles that will describe users' past performance. For example:
   - Matches won/lost
   - Tournaments entered/created

In terms of **non-functional requirements**, we believed `Brackit` should be *usable* on mobile devices. `Brackit` users should be able to:

- View and access all components (Brackets, Rounds, Matches) of a tournament on an Android device.
- Seamlessly enter tournament competitors to brackets on an Android device.

Conceptually, `Brackit` needed to support the creation and maintenance of the following *components*:

- *Tournament*: The highest level of abstraction utilized in Bracket creation. A tournament acts as a *container* for brackets. `Brackit` supports double-elimination tournaments, where competitors cease to be eligible to win the tournament after losing two matches [1].
- *Bracket*: Given the number of entrants and their corresponding seeds (ranks), Double elimination brackets dictate competitor matchups and the progression of competitors through the Winners and Losers brackets. Brackets contain a dynamic list of Rounds.
- *Round*: Rounds contain a dynamic list of Matches.
- *Match*: Matches pair the strongest and corresponding weakest players in a Round according to rank. (That is, in a tournament containing n players, the top (1^{st}) ranked player will be matched with the lowest $(n-1)$ ranked player, the 2^{nd} ranked player will be matched with the $n-2$ ranked player, etc.)

1.2 Justification of Architectural Style Choices

1.2.1 Object-Oriented Architectural Style

As described above, a Double Elimination Tournament mobile management application must maintain a set of well-defined entities (i.e. Tournaments, Brackets, Rounds, and Matches) with predetermined relationships. For example, given n competitors, a correct double-elimination tournament will contain $\lceil \lg n \rceil$ rounds in the Winners bracket and $\lceil \lg n \rceil + \lceil \lg \lg n \rceil$ rounds in the Losers bracket [2]. Also, the progression of competitors can be calculated at the creation of a tournament, and handling this progression follows a deterministic approach (e.g. The winner of Match 1 of Round 1 in the Winners Bracket will always progress to Match 1 Round 2 in the Winners Bracket - see Figure 1 for an illustrative example).
Therefore, to encourage an efficient decomposition of the algorithm and entities associated with Double Elimination Tournament management, we decided to model the architecture of `Brackit` using an **Object-Oriented** (OO) architecture. Specifically, we chose to model each of the components of our application as objects. This allowed us to encapsulate the expected behaviour of each of the tournament objects while maintaining a valid separation of concerns. To explicate the validity of the choice of an OO architecture for `Brackit`, consider the dynamic nature of Tournament creation.
A tournament bracket acts as a container for rounds, which themselves act as containers for matches. To handle tournament progression, the data associated with each match (i.e. which competitor won or lost) should be self-contained within the match object instantiation, but also must be accessible through attributes of that object.

Defining the Match construct as an object enables self-contained class methods and attributes that achieve these intended behaviours. At the same time, setting the Win / Loss attribute of a match as publicly visible allowed us to develop querying strategies that can provide score summaries for participating users.

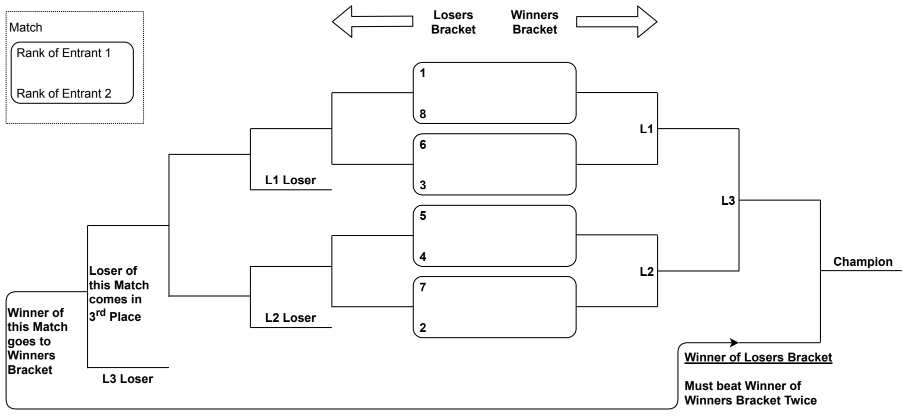

Figure 1: Seeded Double Elimination Tournament Chart for 8 competitors. (Adapted from [3])

1.2.2 Client-Server Architectural Style

A **Client-Server** architecture dictates that a server host, deliver, and manage most of the resources and services that clients can consume. Client-server architectures are used in situations where separation of concerns exists between the client and the server. The modularity inherent in this style lends itself to the development of scalable software [4]. We decided to follow the guidelines set out by the Client-Server Architectural Style because we believed `Brackit` should provide the following core features: Double Elimination bracket algorithm execution, tournament management, and user registration and maintenance. Since we committed to providing these functionalities to both registered and guest users, it was natural to model the users of our application as clients that interact with the available features. Since we did not intend on developing a platform where the communication between users is central (for example, in the form of chats, or message sharing, between users of `Brackit`), designing `Brackit` as a Restful-API [5] Client-Server model was sufficient to provide the functionalities we committed to.
Finally, using a client-server model will allow planned revisions and upgrades to Tournament Management and Creation to be easily integrated into the codebase (See Section 2.5 for a detailed description of what we plan next for `Brackit`). Once a feature and the corresponding frontend client code are tested, they can be incorporated via an update on our users' Android devices. This will allow users to access the new feature(s). That is, using a Client-Server model allows us to ensure that `Brackit` will continue to innovate, progress, and scale by providing users with new and exciting functionalities and behaviours.

1.3 Architectural Diagrams

1.3.1 UML 2 Architectural Component Diagram

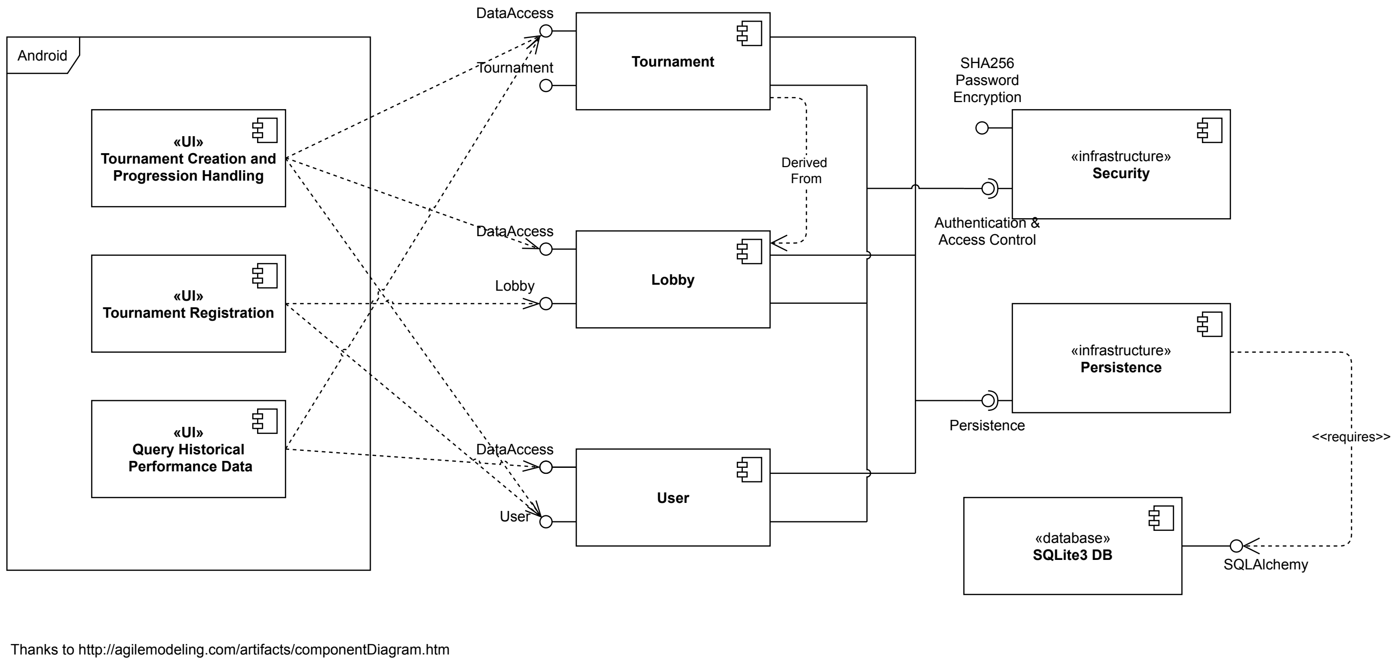
Android
«UI» Tournament Creation and Progression Handling
«UI» Tournament Registration
«UI» Query Historical Performance Data
DataAccess
Tournament
Tournament
DataAccess
Lobby
Lobby
DataAccess
User
User
Derived From
SHA256 Password Encryption
«infrastructure» Security
Authentication & Access Control
«infrastructure» Persistence
Persistence
<<requires>>
«database» SQLite3 DB
SQLAlchemy
Thanks to http://agilemodeling.com/artifacts/componentDiagram.htm

Figure 2: `Brackit - ` UML 2 Architectural Component Diagram

1.3.2 ER Diagram

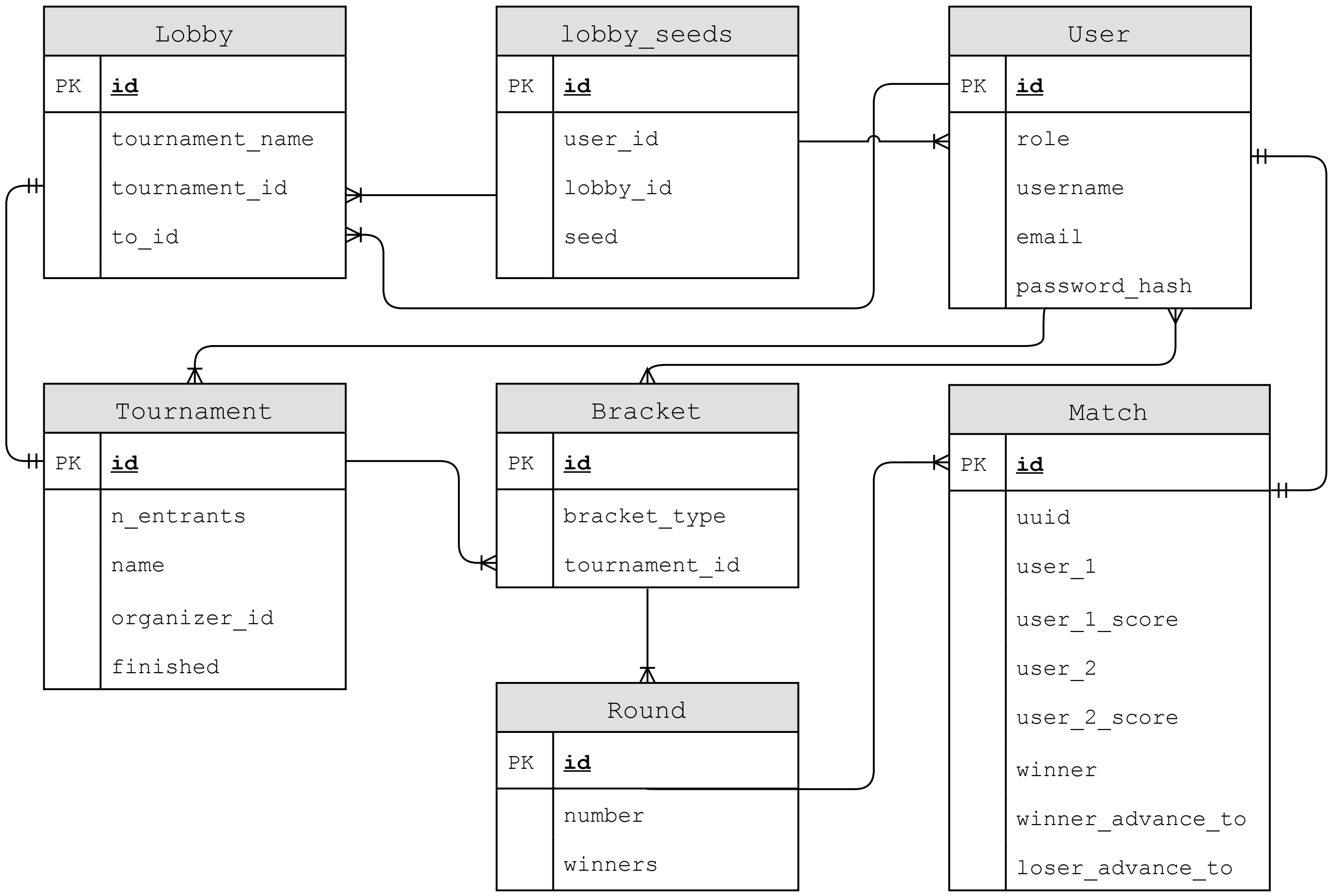

Figure 3: `Brackit -` Entity-Relationship (ER) Diagram

Design

2.1 Description and Justification of Design Patterns

1. *Facade*

   The Facade pattern intends to provide a unified interface to a set of interfaces in a subsystem. For our backend, the API endpoints contained in the `routes` module constitute our Facade. Each endpoint provides a URL which the user can access to invoke all necessary interfaces to execute the necessary code in the backend.

   This pattern provides a singular hub through which frontend clients can access backend data. Modelling our server as a Facade centralized all exposed endpoints and allowed for ease of use and extensibility.

2. *Singleton*

   The Singleton pattern intends to ensure a class has only one instance and to allow global access to that class. The `Flask` app object leverages the Singleton pattern in its design. It is instantiated only once (when the server is started), and is globally accessible.
   A Singleton design pattern for the app object is an appropriate choice because when the app class is instantiated, it acts as a representation of the `Flask` server at compile-time. This representation encapsulates all current server modifications and customizations and specifies the database schema, API endpoints, as well as any backend refactoring. Correspondingly, this guarantees that the app is only instantiated once. This maintains consistency throughout a session and mitigates the possibility that multiple clients create duplicate endpoints, which would result in unintentional behaviour such as inconsistent method invocations.

2.2 Frontend Frameworks Rationalization

To develop our mobile application, both Android and iOS platforms could have met our needs. iOS was a strong candidate since it has a large market audience and it would have allowed us to build the app in a fast and easy way. The disadvantage is that we are more restricted with writing Swift code and building up components for specific devices. Developing using Android allows for more flexible feature implementation using an open source platform that could be adapted to cater to the specific needs of `Brackit`. The downside is that developing Android apps could be more complex since there is no standardization for devices and components.
We decided to proceed with Android since more team members are comfortable with the Android environment. This gave us the flexibility to run an Android environment on both Mac iOS and Windows operating systems. For Frontend development, we used Kotlin to integrate with Android Studio. Although Java seems to be a popular choice for Android app developers, Kotlin, as a relatively new open source language, has been gaining popularity for several reasons. Kotlin is based on the JVM (Java Virtual Machine) and it is fully interoperable with Java. It can be seamlessly integrated with Android Studio. And, Kotlin is more lightweight than Java and can therefore significantly reduce the overhead associated with the project.

2.3 Implementation Analysis

The `Brackit` backend was designed primarily to marry extensibility with the correctness of our relatively complex domain. The main challenge of the backend's design is the creation of correct brackets, as well as the maintenance of brackets as they progress to completion. When a user creates a Tournament, `Brackit` initializes a `tournament` object in the backend, which itself creates the `bracket` objects. Depending on the bracket type, the appropriate number of rounds are created, with each round containing the corresponding number of matches. This is handled automatically because the number of rounds is determined by the number of entrants and the type of bracket (See Section 1.2.1).
Our backend uses the `Flask` python package to expose our bracket and user information to the frontend, as well as SQLAlchemy to manage this information in the database. SQLAlchemy provides a `Model` base-class that allows us to declare the tournament objects as database tables and a runtime interface by which we interact with our `SQLite3` database. We model each class in our class diagram in the `models` module, which creates tables for each class and defines the table relationships in an object-oriented style. This allows us to easily and safely query the database when invoking APIs and enables retrieval of the specific object being requested. Additionally, these models allow for seamless SQL querying for user data, such as users' cross-tournament wins and losses.

2.4 Integration and Coupling Challenges

As a consequence of utilizing SQLAlchemy model our classes in the database, we produced three layers of abstraction in our application:

- the `Routes` layer, i.e. our API Facade,
- the `Model` layer, and
- the `Backend` layer, where bracket generation and progression handling logic is stored.

This created some significant challenges when designing for decoupling. When we query tables for requested objects in the `Routes` layer, we return the `Model` objects and not the `Backend` objects. Accessing `Backend` objects after instantiation in the `Routes` layer became a significant challenge. We addressed this by moving the logic for inputting match results directly into the `Model` layer, as that is the layer responsible for the representation of the objects in the database.
Also, we experimented with each class in the `Model` layer containing its corresponding `Backend` object as an attribute. However, because the `Tournament` object in the `Backend` instantiates its child objects such as the `Bracket`, `Rounds`, and `Matches` during initialization, it became impossible to map one to the other.
The final challenge we'll discuss relates to the encapsulation of user data within match objects (i.e. handling how matches "know" which entrants are participating in said matches). This is known as progression handling. For matches in the initial rounds, progression handling and placement are trivial since the entrants are placed into the matches in the first round when the Tournament is instantiated. However, as the tournament progresses, the client must POST win, loss, and progression data to the backend. When the results of a match are reported, `Brackit` must determine where the winner and loser progress to.
To address this, we explored the use of pointers that correspond to each entrant in subsequent matches. However, it became challenging to ensure that as the tournament progressed and pointers were instantiated with user data, the database reflected these changes. Ultimately, we decided to store references to two other match objects within a `Match` (`loserPlaysInMatch` and `winnerPlaysInMatch`). These attributes reference the matches that the loser and winner progress to. This approach simplified database representation and enabled intuitive bracket navigation on the frontend. That is, by inspecting a match's attributes, one could find subsequent matches in the tournament.

2.5 Suggestions for Future Work

So far, `Brackit` only supports double elimination brackets, but this can be expanded by the addition of special cases in the `Bracket` constructor. An alternative approach would be to create an abstract `Bracket` class with each bracket type as an extension of the abstract `Bracket` class. This may allow for code that is readable and iterable and would be worth the refactoring time if this project were to be expanded.
We also plan to streamline user entry to tournaments via their pre-registered `Brackit` accounts using a QR code for a given tournament that, when read, automatically enters the user into the tournament lobby. Our choices of the Facade Design pattern coupled with our Object-Oriented and Server-Client Architecture will facilitate the integration of this new feature. Specifically, by modelling a Tournament as an object, we can add an additional `uuid` attribute that enables unique QR code generation and matching. Once this feature is supported, further frontend functionality will be needed to display a QR code from the `/api/lobby/<int:lobby_id>/add-user/` endpoint, as well QR Code recognition from an additional API endpoint.

2.6 Design Diagrams

2.6.1 Backend UML Class Diagram

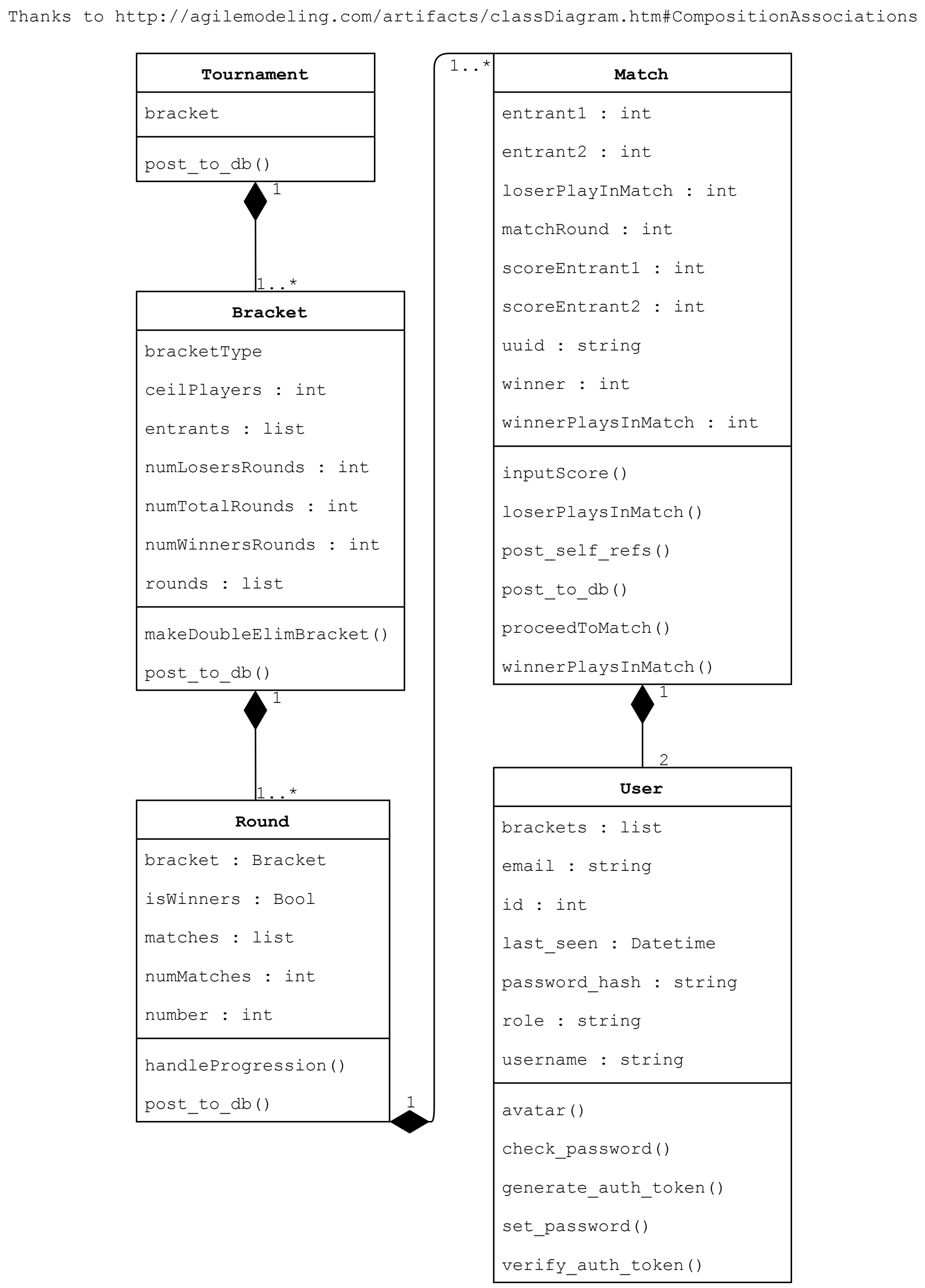

Figure 4: **Brackit** - Backend UML Class Diagram

2.6.2 Frontend UML Class Diagram

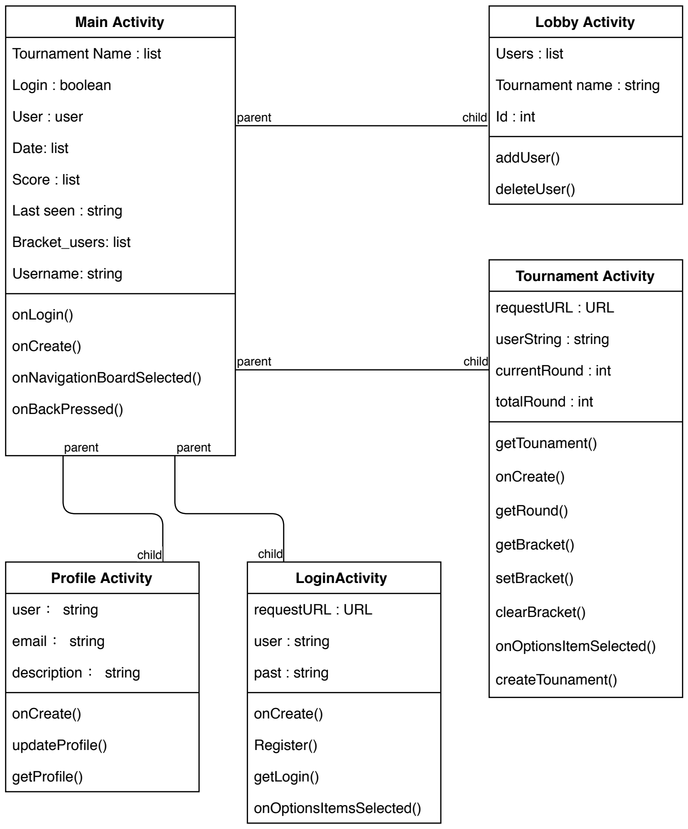

Figure 5: `Brackit -` Frontend UML Class Diagram

2.6.3 `Create Tournament` Sequence Diagram

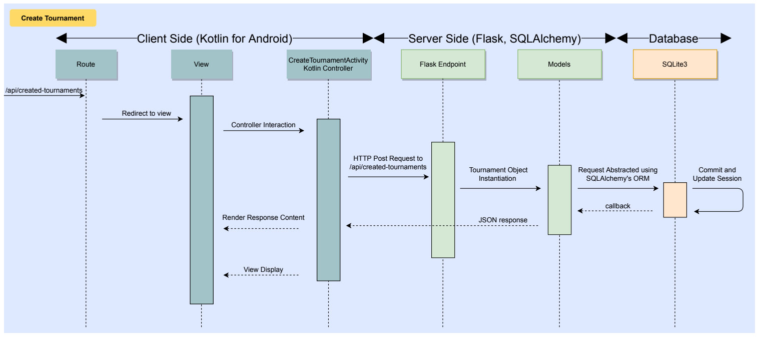
Create Tournament
Client Side (Kotlin for Android)
Server Side (Flask, SQLAlchemy)
Database
Route
View
CreateTournamentActivity Kotlin Controller
Flask Endpoint
Models
SQLite3
/api/created-tournaments
Redirect to view
Controller Interaction
HTTP Post Request to /api/created-tournaments
Tournament Object Instantiation
Request Abstracted using SQLAlchemy's ORM
Commit and Update Session
callback
JSON response
Render Response Content
View Display

Figure 6: `Brackit:` `Create Tournament` Sequence Diagram

Assignment of Tasks

- Jaime Herzog, 101009321:
  - Report:
    - Sections 2.1, 2.3 - 2.5
  - `Brackit`:
    - `Backend` Layer
    - Algorithm for generation of double elimination brackets
    - Progression Handling
    - `winsAndLosses` endpoint - Performance Metrics Endpoint
- Suohong Liu, 101002340:
  - Report:
    -
  - `Brackit`:
    - Asynchronous information processing with the server
    - Progression handling - frontend implementation
    - Endpoint testing
- Xiyi Liu, 101004577:
  - Report:
    - Section 2.2
    - Figure 5
  - `Brackit`:
    - User interface design for frontend
    - Endpoints testing
    - Implementation of user activities
- Alex Trostanovsky, 100984702:
  - Report:
    - LaTeXCompilation
    - Figures 1-4, 6
    - Sections 1.1-1.2
  - `Brackit`:
    - `Models`, `Routes` Layers
    - SQLite3 Database Initialization, Migration
    - Mock data population scripts
    - Endpoints testing and implementation

References

[1] Wikipedia. Double-elimination tournament — Wikipedia, the free encyclopedia, 6-October-2019. [Online; accessed 14-March-2020].

[2] gottfriedville. Double Elimination - How many rounds ? Blog Post. [Online; accessed 15-March-2020].

[3] candied-orange (https://softwareengineering.stackexchange.com/users/131624/candied orange). Tournament bracket algorithm. Software Engineering. [Online; accessed 14-March-2020].

[4] Mei Nagappan, Achyudh Ram Keshav Ram, Aswin Vayiravan, Wenhan Zhu. Lecture notes in software design and architectures, July 2019.

[5] REST API Tutorial. `https://restfulapi.net/`. Accessed: 2020-03-15.