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 Description

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.)

2 Justification of Architectural Style Choices

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. a 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]. In addition, 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.

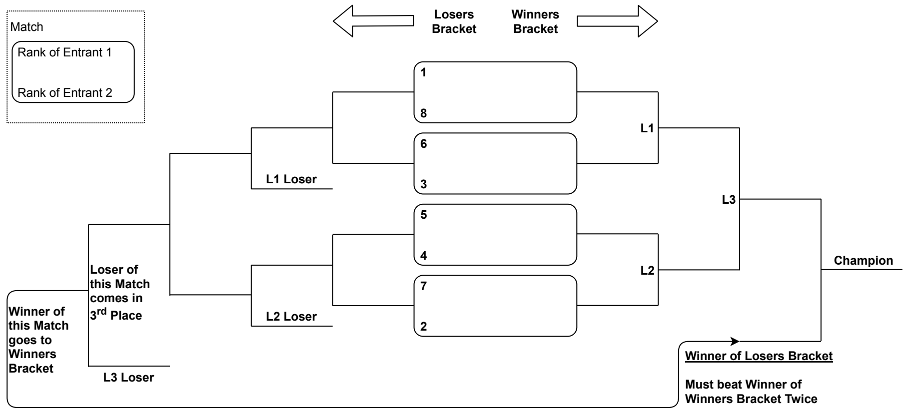

Figure 1: Seeded Double Elimination Tournament Chart for 8 competitors. (Adapted from [3])

2.2 Client-Server Architectural Style

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

3 Architectural Diagrams

3.1 UML 2 Architectural Component Diagram

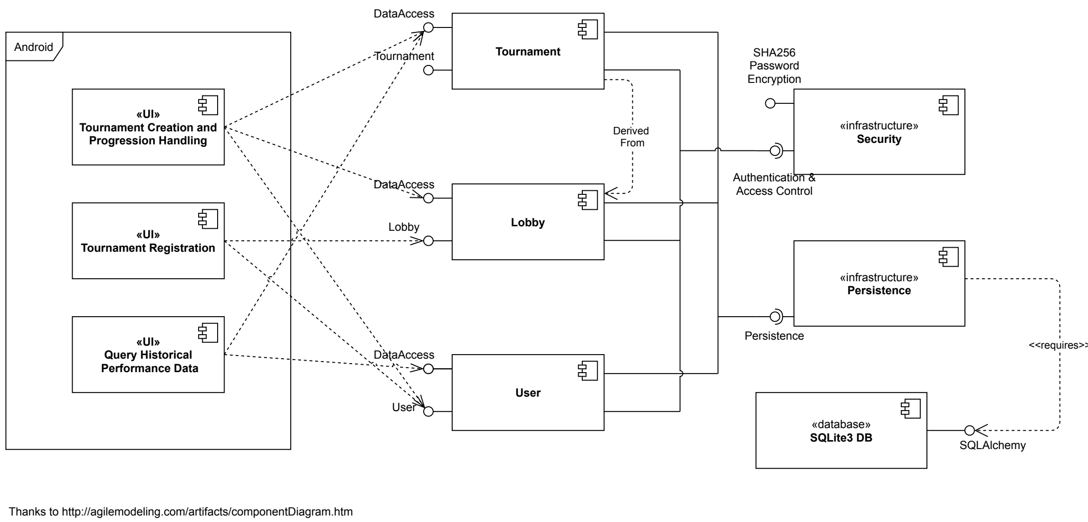

Figure 2: `Brackit` - UML 2 Architectural Component Diagram

3.2 ER Diagram

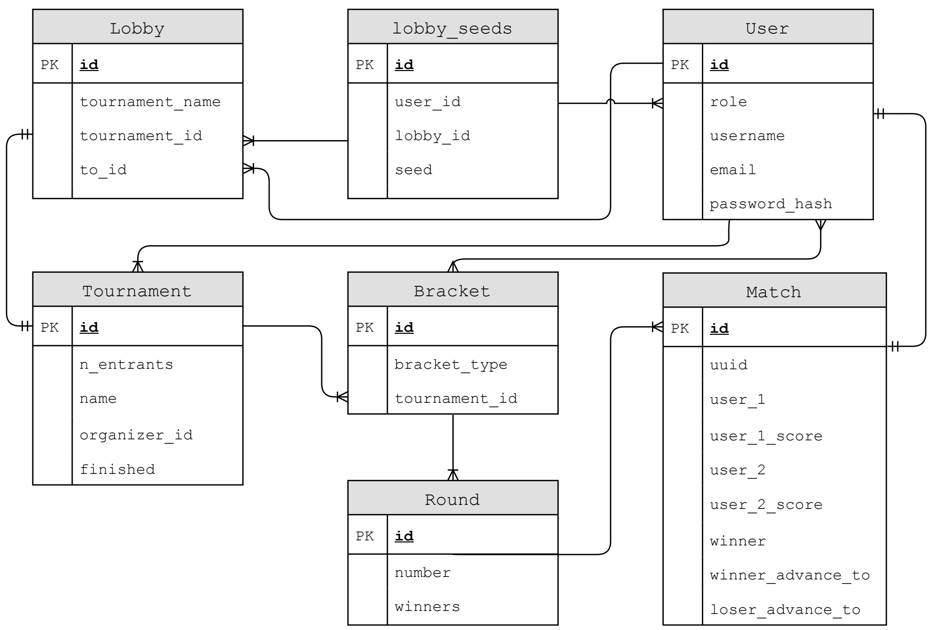

Figure 3: `Brackit -` Entity-Relationship (ER) Diagram

Design

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 accessed globally throughout the backend.
   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 by specifying 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 the inconsistent method invocations.

2 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 deterministic with respect to the number of entrants and the type of bracket (See Section 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` baseclass 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.

3 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 the 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 fully. This problem would be another

area of future evolution for our system.
A specific implementation challenge is handling how each match knows who is the entrant which is playing in said match. This is called progression - for matches in the initial rounds, this is trivial as the entrants are simply placed into the matches when the Tournament is instantiated, but as the tournament progresses, the client must send data to the backend via a POST request to progress the tournament. When the results of a match are reported, the system must somehow know where the winner and the loser progress to. One potential solution we explored was to use pointers for each entrant in subsequent matches, where each pointer points to the winner attribute of the previous match which progresses to the subsequent match. However, this solution ran into problems with the SQLAlchemy model, as it became challenging to ensure that as the tournament progressed and the pointers became instantiated with non-null data, that the database reflected these changes. Ultimately, we decided to have two references to two other match objects, loserPlaysInMatch and winnerPlaysInMatch, which reference the match that the loser and winner progress to respectively. This approach simplified database representation, as well as made navigating the bracket easier on the frontend since one could find subsequent matches from a previous match just by inspecting that match object's attributes.

4 Suggestions for Future Work

So far, `Brackit` only supports double elimination brackets, but this can be expanded easily 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.

5 Design Diagrams

5.1 UML Class Diagram

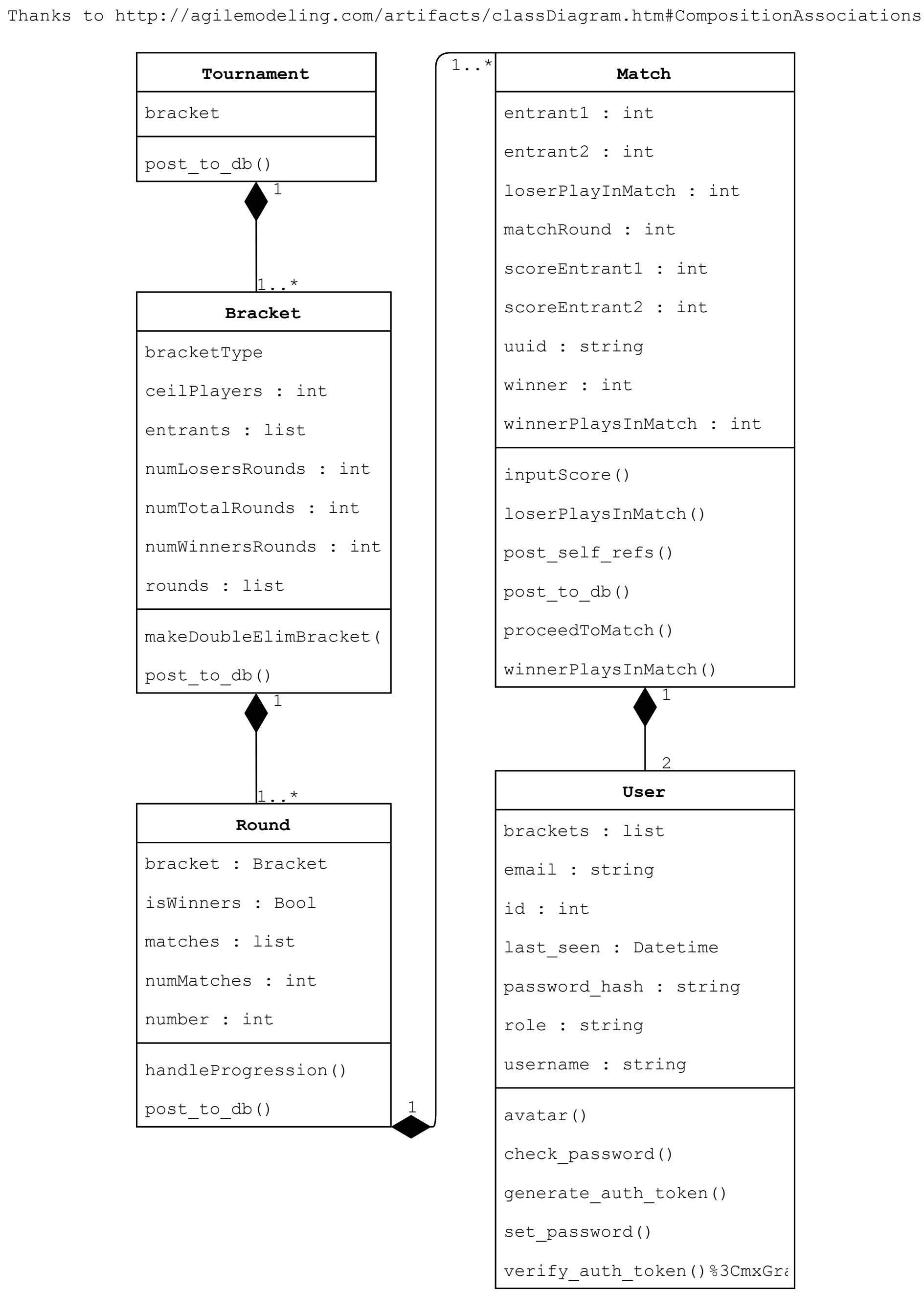

Figure 4: `Brackit` - UML Class Diagram

5.2 `Create Tournament` Sequence Diagram

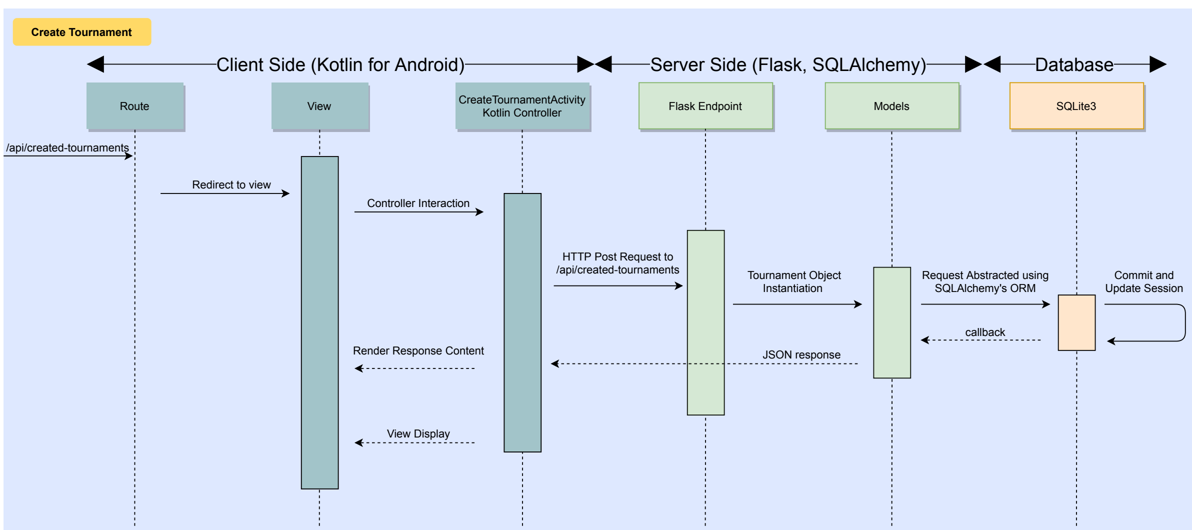

Figure 5: `Brackit: Create Tournament` Sequence Diagram

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].