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

In developing `Brackit`, we set out to address an urgent need by tournament attendants and organizers 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. Registerd `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' history:
    - Matches won/lost
    - Tournaments entered/created

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

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

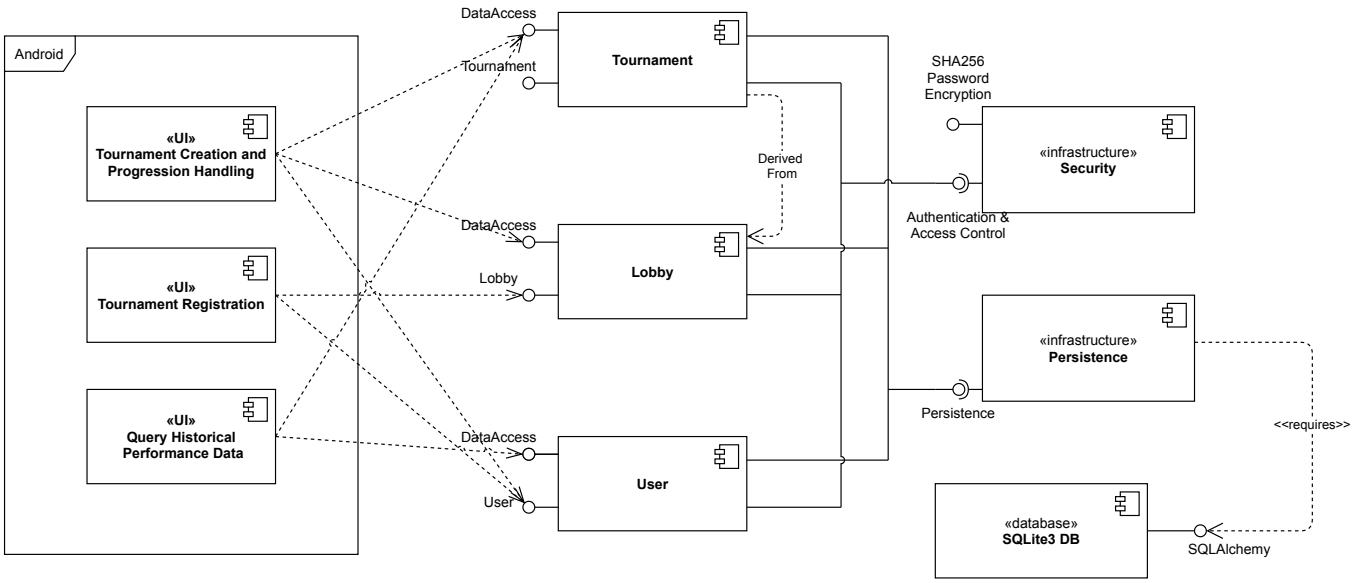
Conceptually, `Brackit` needed to support the creation and maintenance of the following *components*:

- *Tournament*: The highest level abstraction utilized in Bracket creation. A tournament acts a *container* for brackets.
- *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

2 Justification

To deve

3 Architectural Diagrams



Thanks to http://agilemodeling.com/artifacts/componentDiagram.htm

Figure 1: `Brackit` - UML 2 Architectural Component Diagram

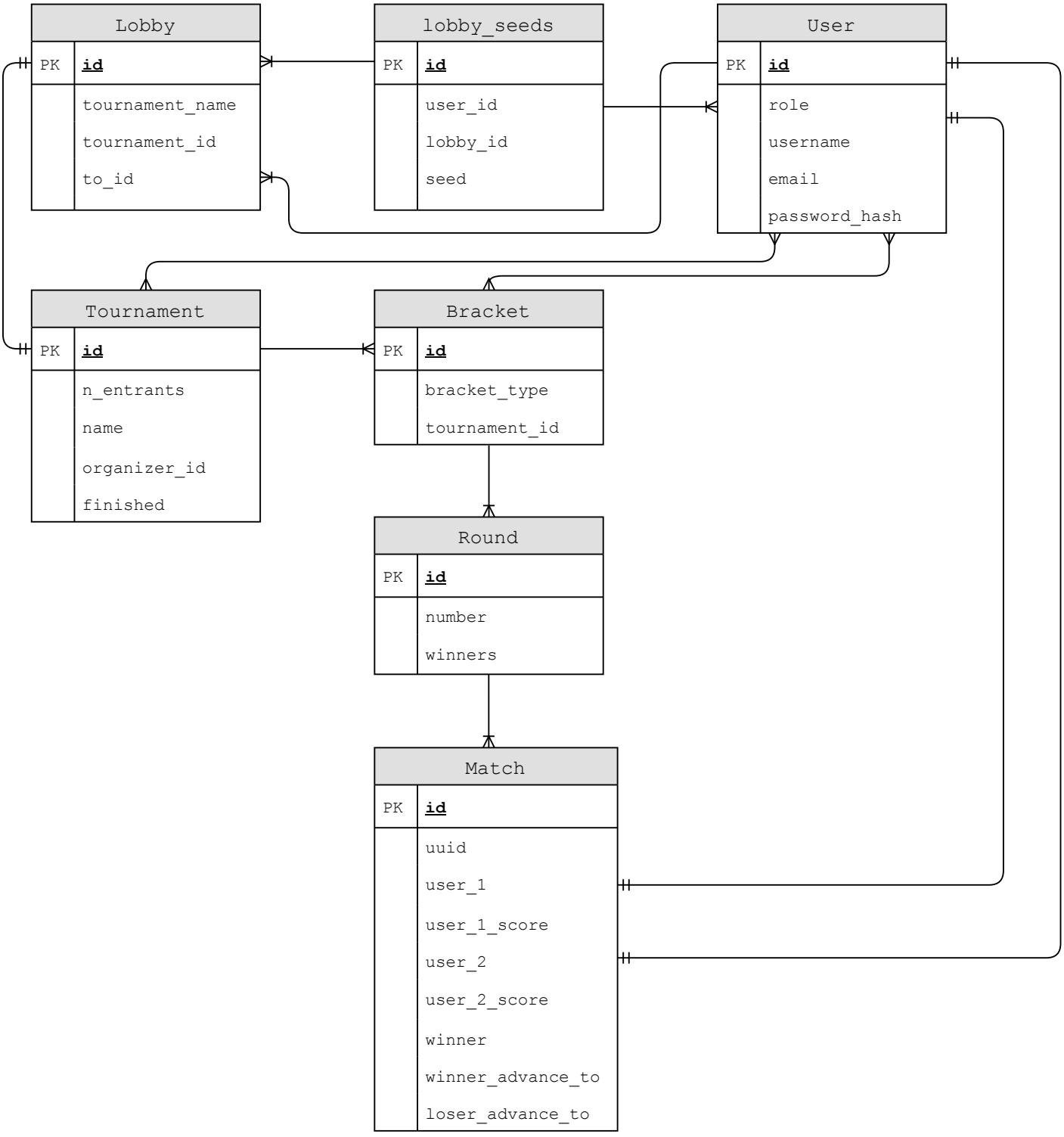


Figure 2: `Brackit` - Entity Relationship (ER) Diagram

Design

1 Description and Rationalization

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 we create a Tournament, we create a tournament object in the backend, which itself creates the bracket. Depending on the bracket type, it creates the appropriate number of rounds, with each round having its appropriate number of matches. This is handled automatically because the number of rounds is deterministic to the number of entrants and the type of bracket. See the discussion on the justification of the object-oriented architecture. So far, we only have the double elimination bracket type implemented, but this can be expanded easily by just adding additional cases in the Bracket constructor. An alternative approach would be to create an abstract Bracket class with each bracket type as an implementation of the abstract Bracket class. This may allow for code that is easier to read and iterate on, and would be worth the time spent refactoring if this project were to be expanded on.

A specific implementation challenge is hangling 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,

Our backend uses the Flask python package to expose our bracket and user information to the frontend, as well as SQLAlchemy to manage our information in the database. SQLAlchemy has a Model package which models our objects in the table, and provides the interface from which we interact with our database at runtime. We model each class in our class diagram in models.py, which creates tables for each class and defines the table relationships in an object oriented style. This allows us to query the database itself easily and safely when the APIs are invoked, enabling easy retrieval of the specific object being requested. Additionally, these models allow for easy sql querying for user data, such as users cross-tournament wins and losses.

2 Design Patterns

1. Facade Pattern

   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.py constitute our Facade, as each endpoint provides a centralized and easy to use endpoint which the user can access and which invokes all necessary interfaces to execute the necessary code in the backend.

   This pattern facilitates a single way in which frontend clients can access backend data, as well as centralizes all exposed endpoints on the server side, allowing for ease of extensibility.

2. Singleton Pattern

   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, as it is instantiated only once when the server is started, and is accessed globally throughout the backend.

   The justification for the use of the Singleton pattern for the app object is because the app class itself, when instantiated, is the representation of the Flask server instance at compile time, and is how we modify and customize our server, i.e. specifying the database schema, or specifying the API endpoints. Additionally, ensuring the app is only instantiated once maintains consistency throughout the session, as multiple apps could potentially create multiple duplicate endpoints, which would create unintentional behaviour such as method invokation duplication.

Design Diagrams

Thanks to http://agilemodeling.com/artifacts/classDiagram.htm#CompositionAssociations

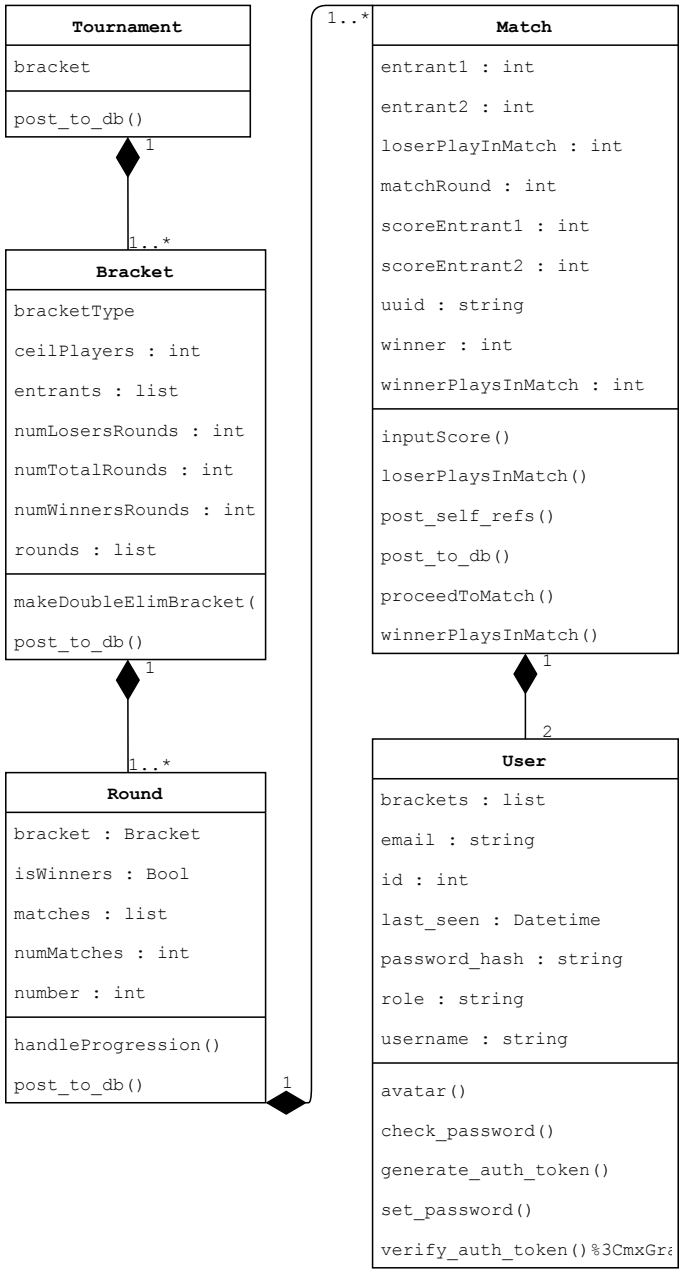


Figure 3: `Brackit` - UML Class Diagram