Web Mighty

Design and Planning Document - Team 2

Revisions

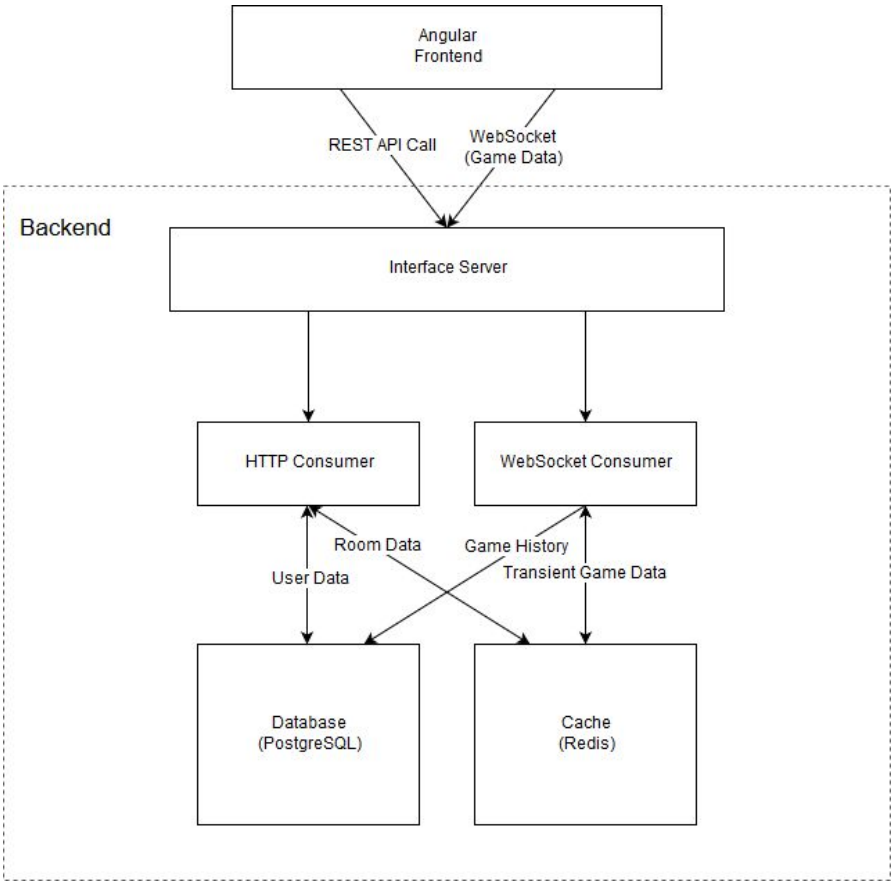
2017-11-19 - Sprint 3
2017-11-04 - Sprint 2
2017-11-02 - Initial Version

Changes

Diagrams for frontend architecture has changed. Changed parts are marked as blue.

System Architecture

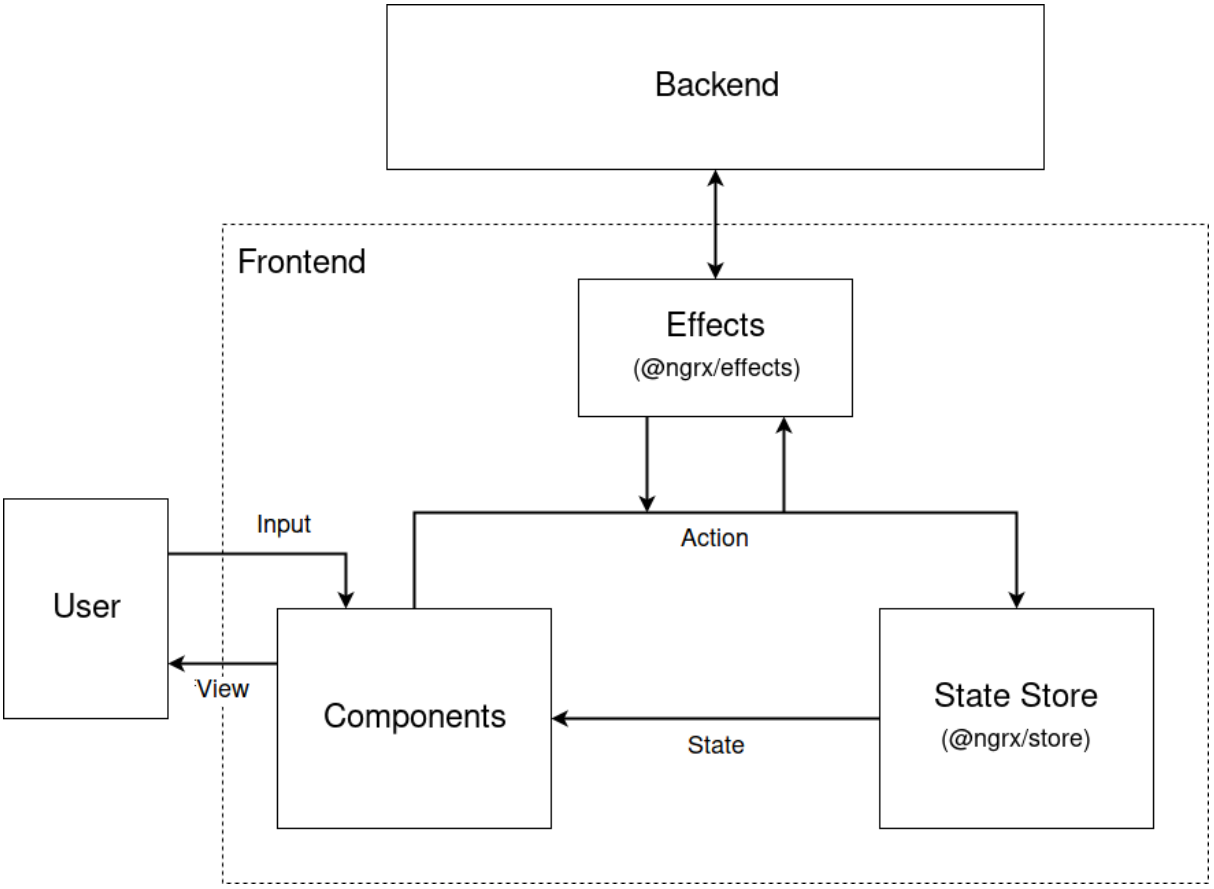
<Backend Architecture>



Since Django can only able to handle plain HTTP request and response, we decided to use Django Channels, which is the official extension project for providing arbitrary types of protocol. By using this architecture, we can handle HTTP and Websocket protocol packet consistently, without reinventing the wheel.

For reliable database performance, we decided to use PostgreSQL for DB. Django supports PostgreSQL perfectly, thanks to psycopg2. SQLite3, which is the default configuration of Django database setting, is convenient to use, but its performance is quite bad in concurrent job and speed aspects.

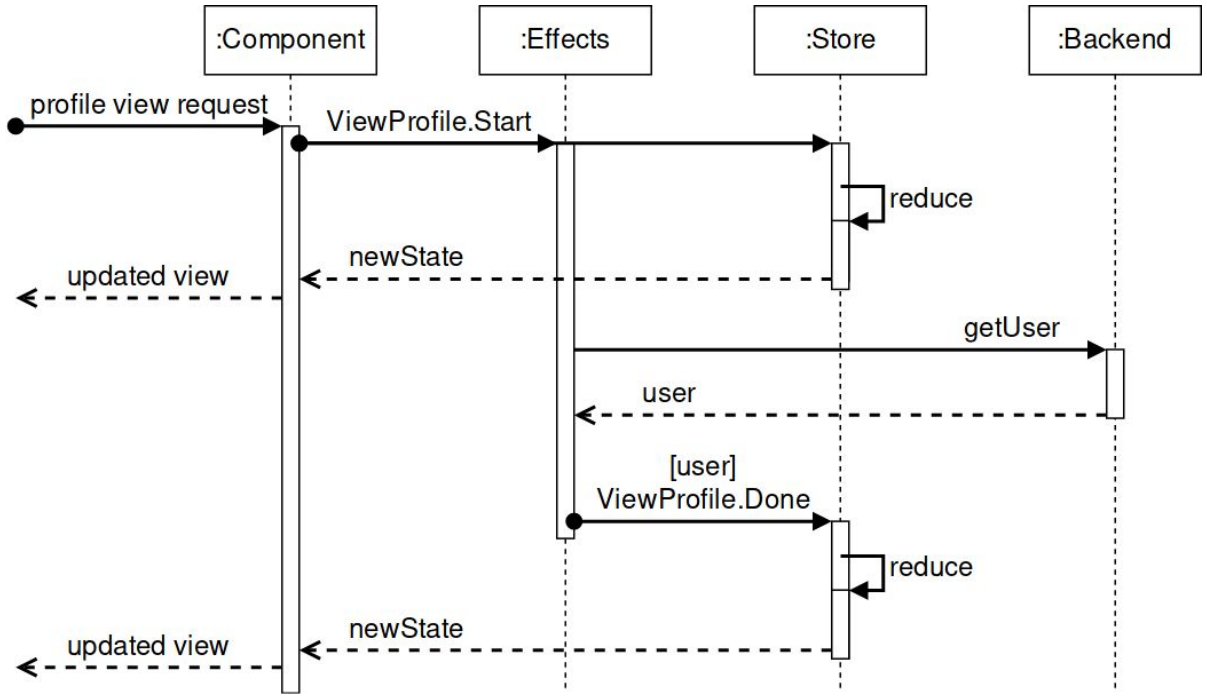
<Frontend Architecture>



Because we are making web powered game, we will use WebSocket to communicate with server and client. For Angular WebSocket support, we will use angular2-websocket package.

To manage the state properly, we will use ngrx. With @ngrx/store, we can manage states of the application using event-driven structure. State store will receive “actions” from components, and it will change its state using its “reducer,” which is a function from its current state and the dispatched action to the new state. Components will subscribe to the store and reflect state changes to views. This model can be thought as functional reactive.

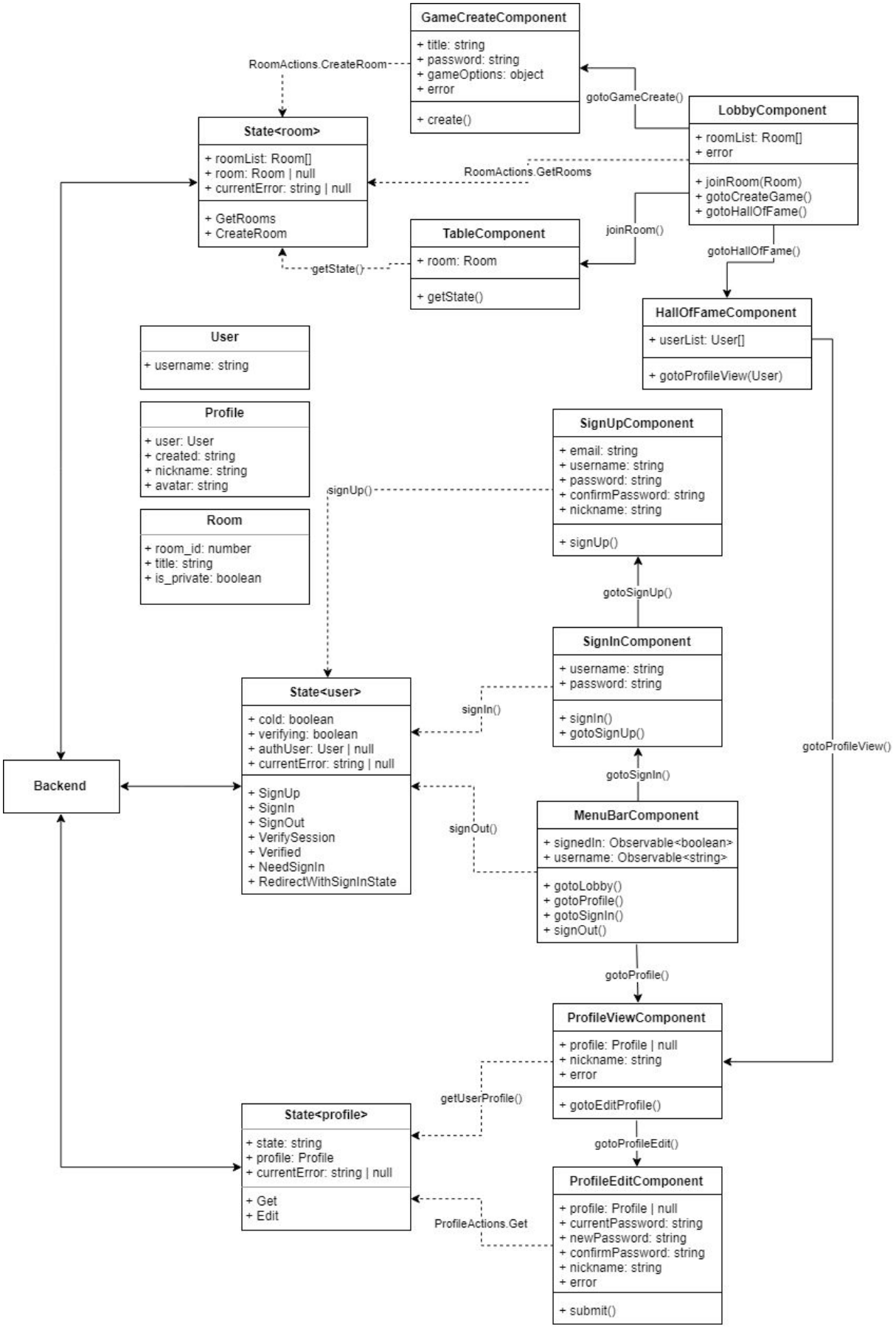
<Example of communication between modules>



The figure above describes a communication sequence between modules when the user visits a profile page. Here, we can think Effects as a ordinary Angular service; it will listen to actions -- similar to procedure calls of Angular services -- and make side effects like communicating with the backend. However, there is a significant difference between Services and Effects: Component does not directly receive results from Effects. Instead, Effects sends the result to Store, and Component receives data from it. This way, components can focus on displaying current state, effects communicating with the backend, and the state store updating its state from dispatched events.

Design Details

<Frontend>


GameCreateComponent
+ title: string
+ password: string
+ gameOptions: object
+ error
+ create()
RoomActions.CreateRoom
gotoGameCreate()
LobbyComponent
+ roomList: Room[]
+ error
+ joinRoom(Room)
+ gotoCreateGame()
+ gotoHallOfFame()
State<room>
+ roomList: Room[]
+ room: Room | null
+ currentError: string | null
+ GetRooms
+ CreateRoom
RoomActions.GetRooms
joinRoom()
gotoHallOfFame()
TableComponent
+ room: Room
+ getState()
getState()
HallOfFameComponent
+ userList: User[]
+ gotoProfileView(User)
User
+ username: string
Profile
+ user: User
+ created: string
+ nickname: string
+ avatar: string
Room
+ room_id: number
+ title: string
+ is_private: boolean
SignUpComponent
+ email: string
+ username: string
+ password: string
+ confirmPassword: string
+ nickname: string
+ signUp()
signUp()
gotoSignUp()
SignInComponent
+ username: string
+ password: string
+ signIn()
+ gotoSignUp()
State<user>
+ cold: boolean
+ verifying: boolean
+ authUser: User | null
+ currentError: string | null
+ SignUp
+ SignIn
+ SignOut
+ VerifySession
+ Verified
+ NeedSignIn
+ RedirectWithSignInState
signIn()
signOut()
gotoSignIn()
gotoProfileView()
Backend
MenuBarComponent
+ signedIn: Observable<boolean>
+ username: Observable<string>
+ gotoLobby()
+ gotoProfile()
+ gotoSignIn()
+ signOut()
gotoProfile()
ProfileViewComponent
+ profile: Profile | null
+ nickname: string
+ error
+ gotoEditProfile()
getUserProfile()
gotoProfileEdit()
State<profile>
+ state: string
+ profile: Profile
+ currentError: string | null
+ Get
+ Edit
ProfileActions.Get
ProfileEditComponent
+ profile: Profile | null
+ currentPassword: string
+ newPassword: string
+ confirmPassword: string
+ nickname: string
+ error
+ submit()

Components

- LobbyComponent
- GameCreateComponent
- TableComponent
- HallOfFameComponent
- SignUpComponent
- SignInComponent
- MenuBarComponent
- ProfileViewComponent
- ProfileEditComponent

State Store

Actions

+ RouterActions
    + GoByUrl
    + Go
    + Back
    + Forward
+ UserActions
    + SignUp
    + SignIn
    + SignOut
    + VerifySession
+ ProfileActions
    + Get
    + Edit
+ RoomActions
    + GetRooms
    + CreateRoom

Reducers

+ user
+ profile
+ room
+ router

Effects

+ RouterEffects
+ UserEffects
+ ProfileEffects

+ RoomEffects

Classes

- Room
  + room_id
  + title
  + rule
  + is_private

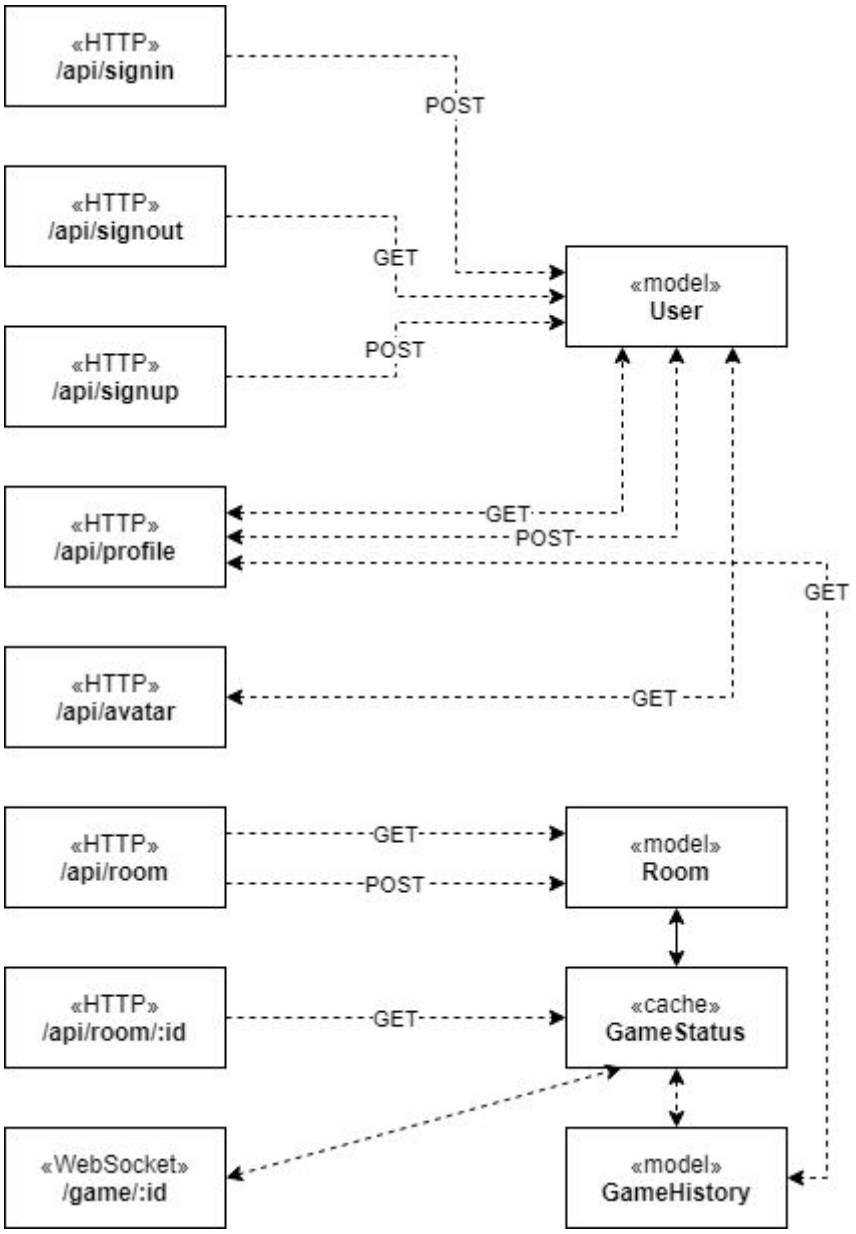
- GameStatus
  + rule
  + dealer
  + player
  + giruda
  + bid
  + president
  + friend
  + round

- User
  + username

- Profile
  + user
  + created
  + nickname
  + avatar

- Card
  + suit
  + rank

<Backend>



Model Specification

- User
  + username: CharField
  + email: EmailField
  + password: CharField
  + avatar: ImageField
  + created: DateTimeField

- Room
  + room_id: CharField
  + title: CharField
  + is_private: BooleanField

+ password: ChardField
+ created: DateTimeField

- GameHistory
  + play_date: DateTimeField
  + players: ManyToManyField(User)
  + president: ForeignKey(User)
  + friend: ForeignKey(User, null=True)
  + giruda: SmallIntegerField
  + bid: SmallIntegerField
  + score: SmallIntegerField

Game data and Room data will be stored in cache, and will be communicated with WebSocket. For example, the play of card will be dispatched from Angular to Django, and Django will handle this data and change the data which reside in cache, and broadcast to other WebSockets to make game flow.

Implementation Plan

Feature: Unregistered users can sign up to create new account
Scenario: Sign up
Iteration: Sprint 2

Implement Sign Up feature of UserService
-> Implement SignUpComponent
Difficulty: ★★☆☆☆
Estimated time: 4 hrs. Email verification logic and API call construction task will be divided to two people.

Feature: Users can login to play Mighty
Scenario: Login
Iteration: Sprint 2

Implement Sign In feature of UserService
-> Implement SignInComponent
Difficulty: ★☆☆☆☆
Estimated time: 2 hrs. Frontend component writing will be done with one person.

Feature: Users can see the profile page
Scenario: Display profile

Iteration: Sprint 3

Implement User Lookup feature of UserService
-> Implement ProfileViewComponent
Difficulty: ★★☆☆☆
Estimated time: 5 hrs. Frontend component writing will be done with two people, one for API and model, and another for component writing.

Feature: Users can edit their own profile
Scenario: Edit personal information
Iteration: Sprint 3

Implement User Profile Edit feature of UserService
-> Implement ProfileEditComponent
Difficulty: ★★☆☆☆
Estimated time: 4 hrs. Frontend component writing will be done with two people, one for API and model, and another for component writing.

Scenario: Upload new avatar
Iteration: Sprint 4

Implement Upload Profile Picture feature of UserService
-> Implement ProfileEditComponent
Difficulty: ★★★☆☆
Estimated time: 6 hrs. File handling for frontend will be done with two people, and another one people will deal with Django's image processing module.

Feature: Users can join or search room on Lobby
Scenario: Join a room
Iteration: Sprint 4

Implement Room List feature of GameService
-> Implement LobbyComponent
Difficulty: ★★☆☆☆
Estimated time: 3 hrs. Two people will design the layout of lobby, and implement the component.

Scenario: Search or filter room
Iteration: Sprint 4

Implement LobbyComponent

Difficulty: ★☆☆☆☆
Estimated time: 1 hrs. One person will implement room search filter with already written search API.

Feature: Users can see their profile from the Lobby
Scenario: Navigate to the profile page
Iteration: Sprint 3

Implement User Lookup feature of UserService
-> Implement MenuBarComponent
Difficulty: ★★☆☆☆
Estimated time: 3 hrs. One person will deal with CSS about menubar, and one will write component template.

Feature: Users can see rankings on the Hall of Fame
Scenario: Display the Hall of Fame
Iteration: Sprint 4

Implement Ranking feature of UserService
-> Implement HallOfFameComponent
Difficulty: ★★★☆☆
Estimated time: 7 hrs. Model filtering and caching will be deal with one person, while others will deal with frontend component.

Feature: Users can create a game from the Lobby
Scenario: Display game creation popup
Iteration: Sprint 3
Difficulty: ★★☆☆☆
Estimated time: 5 hrs. One person will deal with CSS for popup (or modal), and others will deal with API and backend cache for room data.

Scenario: Create a game
Iteration: Sprint 3

Implement Room Create feature of GameService
-> Implement GameCreateComponent
Difficulty: ★★★☆☆
Estimated time: 5 hrs. Two people will deal with backend game data storage, which is reside in cache, and construct the API for it. Other one person will write frontend component.

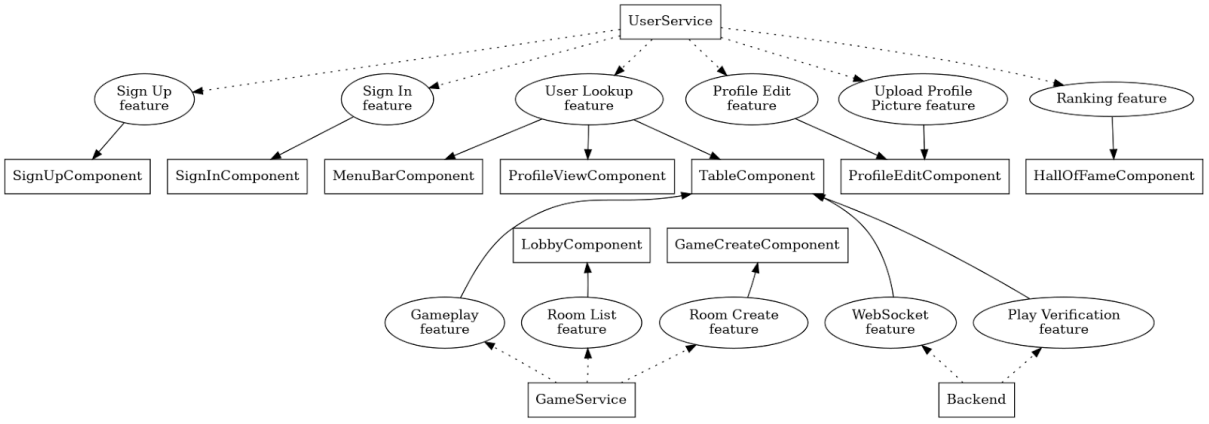
Feature: Users can play game

Scenario: Play Mighty
Iteration: Sprint 5

Implement User Lookup feature of UserService,
Implement Gameplay feature of GameService,
Implement WebSocket feature of backend,
Implement Play Verification feature of backend
-> Implement TableComponent
Difficulty: ★★★★★
Estimated time: cannot estimate. All of the member will implement Mighty game logic, AI, GameStatus model for game session data, which is saved in cache, and frontend component communicating with backend.

Scenario: Leave a game
Iteration: Sprint 5
Difficulty: ★★☆☆☆
Estimated time: 2 hrs. This will be also implemented as the above plan.

<Work dependency diagram>



There can be many risks during the project. We are anticipating that the Mighty AI implementation will be the main risk. To resolve this risk, we will endeavour to make real user and AI player with no difference at the backend logic through the abstraction. To successfully implement this, a lot of structural design planning will be required.

Also, since we are building the concurrent and asynchronous game system, testing and integration can be hard. To resolve this risk, we will use cache backend to make game consistent.

Testing Plan

We will write test alongside the implementation. All of the implementation plan in a sprint will be tested by the corresponding test code.

Unit Testing

We will use built-in Django testing tool to write unit test. As we have done in Homework 3, we can write tests to each unit, like REST API call.
We will use Jasmine to test Angular components. We will use `TestBed` to create and set the default variables, and run tests.
We will test all of the module by unit testing, for example SignInComponent will be tested by calling inner method (e.g. gotoSignUp()) and assert the following effects(e.g. navigation to SignUpComponent).

Functional Testing

Functional testing is available through built-in Django testing tool. Since it is REST API server, we can just provide example mock database data, feed input, and examine output.

/api/signin, /api/signout, and /api/signup will be tested with mock database and mock cache, whether the cache data, database and session is properly set by calling these APIs.

/api/room, /api/room/:id will be tested with mock cache and mock Clients, whether the room cache data is properly set by calling these APIs.

WebSocket protocol will be also tested through functional testing. We will create WebSocket object with Python (or JavaScript maybe?), and test the functionality of game flow.

For Angular, we will use Jasmine to do functional test in Angular. We can provide the input through `TestBed` or mock, and examine the output of component's method or services.

Acceptance & integration testing

We will use Karma to do end-to-end test for Angular. Since we use WebSocket to communicate, we will not provide mocks, but will test frontend Angular and backend Django together. Test will be done as functional testing, which the mock is not exist.