Cupet Phase 0 Progress Report

Group Members (Group 18):

Andrew Qiu
Daniel Baek
Fangyi Li
Harry Xu
Katherine Jelich
Kenneth Tran

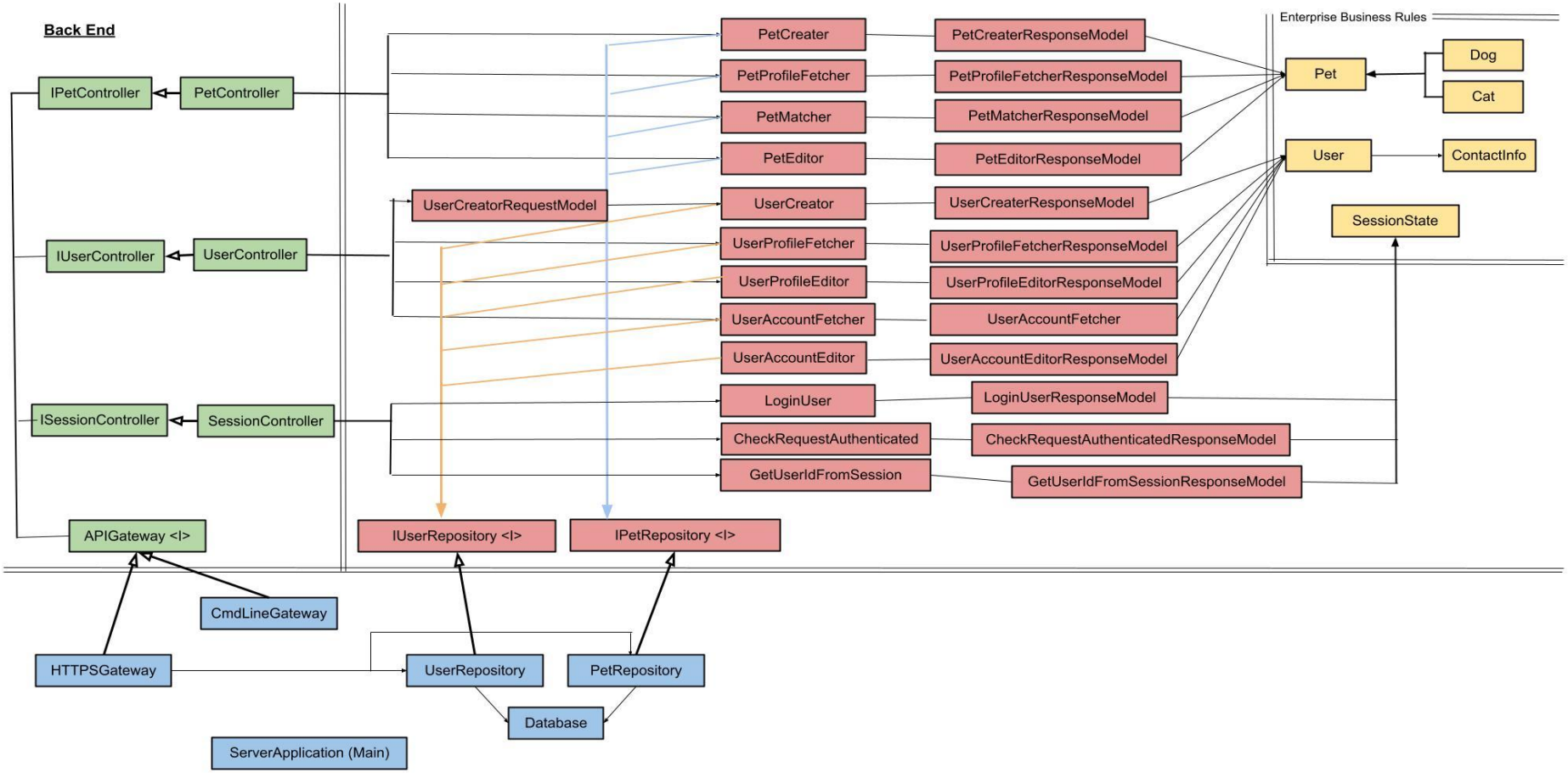
Summary of specification:

Cupet is an Android app with two parts: a front-end and a back-end. The front-end is where a user interacts with the app. The back-end is responsible for storing, processing, and relaying data from and to the front-end. Users of the app can register and log in to the app using their email and password. To register, a user needs to provide details, including their full name, email address, and home address. Each user has their own user profile.

Each user can have multiple pets, which each have their own pet profile. For each of their pets, the app displays to the user a list of other pets—or, potential matches for that pet. The potential matches are displayed based on location, as well as the preferences that the user has selected (ex., types of animals they want their pet to be matched with). Users are able to select which pets they are interested in matching with from this list.

If two users have selected each other's pet, they have matched with each other, and their pets are added to each other's match list. The two users are then able to view each other's contact information on each other's profiles.

Summary of CRC model: (as a diagram)



Filled in arrows represent dependencies and hollow arrows represent extension/implementation. Some interfaces and classes were omitted to avoid crowding the diagram.

Important CRC cards:

- APIGateway<I>: Defines all the methods that the endpoints should implement
- HTTPSGateway: Implements APIGateway. Connects endpoints to HTTP requests
- CmdLineGateway: Implements APIGateway. Connects endpoints to command line input
- IUserRepository<I>: Defines all the methods that the UserRepository driver class should implement
- IUserController<I>: Defines all the methods that call the uses cases that manipulate user data

Summary of scenario-walkthrough:

Scenario: Create a new User (Signup)

1. User launches front end application
2. User performs signup action
3. Input data is sent as an HTTP request to the backend API endpoint (HTTPSGateway)
4. In HTTPSGateway, a RestController which implements the endpoints defined by the interface APIGateway, the HTTP request is processed and sent to the UserController
5. UserController, which is responsible for managing all the use cases, calls the relevant use case
6. The use case, which has an implementation of IUserRepository injected in its constructor, executes the action and pushes to the repository, then returns a UserCreaterResponseModel object back to UserController
7. UserController takes the UserCreatorResponseModel, formats it into a JSON String and returns it back to HTTPSGateway
8. HTTPSGateway returns this JSON string as a response to the initial HTTP request
9. Front end application receives this JSON string response and performs any necessary actions (i.e. update UI)

Summary of skeleton program:

The skeleton program consists of the backend application. It runs on Spring Boot, using Spring JPA to connect to a remote MySql database. It establishes endpoints for HTTP requests with which users can interact with the server (e.g. fetching a user, creating a new user), as well as terminal commands that allow you to perform the same actions, but directly from the command line.

Open questions:

- General implementation questions about the front end:
    - How will we design the UI?
    - How will implementing clean architecture work out when we're already using a highly structured framework? (Android)
- Are test cases necessary for extremely simple classes and methods? For instance, use case classes like FetchUserAccount just re-package data from IUserRepository/UserRepository
- How will we implement retrieving longitude and latitude from a user's home address? Is there an API we can use for this purpose? What happens if there is a typo in a user's address/we cannot find a longitude and latitude?
- Since our project is split into two separate programs (front-end and back-end), how can we manage two repositories in GitHub Classroom?
- Adhering our program to clean architecture as much as possible, we have lots of interfaces, and classes whose only purpose is to re-package data into new objects. This adds a lot of boilerplate code to our project. Doesn't this contribute to runtime issues and make our project significantly more complex than it needs to be

What has gone well

- Clean implementation of clean architecture
    - We spent a lot of time ensuring that our design conformed to the clean architecture principles, and this design has simplified the process of expanding upon the program. Clean architecture has worked very well for the purpose of building a larger program with several contributors.
    - Though it took more time to implement more interfaces and classes that helped ensure the separation of layers, as we move further into the project, we are seeing how abstraction can make code more “hot-swappable”. It’s easier to expand on previous code by adding new implementations of an interface for example.
- CRC model diagram
    - The CRC diagram as well as the CRC cards helped us to really solidify and understand our design and communicate that to each of our group members.
    - It also helped us better visualize the flow of control through our program in potential scenarios. The diagram also allows us to easily visualize how different classes in our CRC model interact, and identify any areas of violation of clean architecture.

- Doing lots of design prep beforehand helped us:
    - Better understand how our program is structured. The CRC model diagram in particular helped us figure out what a class should import and how data is passed down and up between layers through dependency inversion.
    - Figure out the purpose of classes. The separation of classes into layers identifies what happens to data when passed into a class method. For instance, we know that calling a controller method like createUser would somehow pass that data to a use case. By having a consistent naming scheme (ex. Identifying controllers in their class name), it is easy to understand what a class’ methods look like.

Summary of group member work distribution:

Kenneth
Katherine
Fangyi
Harry
Daniel
Andrew

This was the task list we used to manage our work:

1. Write entities
    - User
    - Pet
    - ContactInfo
2. Write interfaces in use cases layer
    - Input port for UserCreator (UserCreatorRequestModel)
    - Output port for UserCreator (UserCreatorResponseModel)
    - IUserRepository
3. Write interfaces in controller layer
    - IUserController
    - APIGateway

- Write implementations in Use Cases layer
    - UserCreator
- Write implementations in Back end layer
    - UserController
- Write implementations in Details layer
    - CmdLineGateway
    - UserRepository
    - CmdRunner

Next Steps

- Start implementing the front end (Android App, UI)
- Determine and start learning about which frameworks we might need
- Continue implementing backend features:
    - Use cases (e.g. MatchPets, CreatePetProfile)
    - Controllers (e.g. PetController, MatchController)
    - Database repositories (e.g. PetRepository)
- Refactoring old code

The current plans for how the work will be distributed is as follows: Andrew will work on implementing and integrating the backend features. Kenneth plans to continue to work on building the Database repositories. Fangyi plans to continue to work on implementing the backend features such as the use cases. Daniel will work on Backend features including the controller classes. Harry and Katherine will start working on implementing the front end, such as the Android App and the UI. All members are working on learning about the different frameworks that might be needed, and will collaborate on refactoring the code that has already been written. As the tasks become more clear during the implementation of the code, the group may redistribute based on what is appropriate.