Design Specification : Milestone 1
for Fantastic Beasts And Where To Find Them
Version 1.0
By Chau khanh Bui (ckb62), Paviya Tojaroon (pt335), Rebecca Liu (rll249), Zhuangjin (Jane) Du (zd53)

1. System description:

1.1. Core Vision

Our plan is to make a game themed after *Fantastic Beasts and Where to Find Them*. Two or more players enter the Harry Potter Universe. The story begins when Newt Scamander discovers that his magical beasts have escaped and are running loose. Each player's task is try to win by either defeating their opponents or collecting the most beasts. When time is up, the player with the most points wins the game. Players will encounter timed day mode and night mode; players collect items and spells in day mode, and battle beasts and each other in night mode.

1.2. Key Features

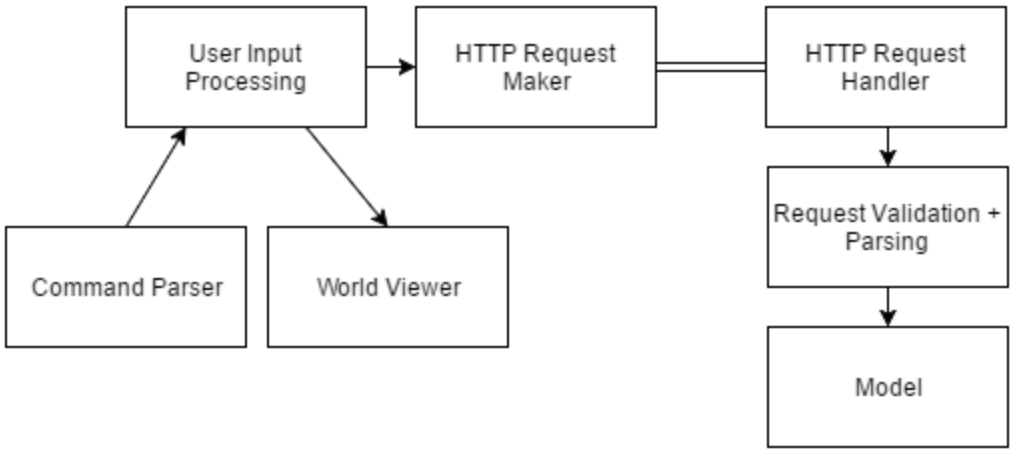
- Server-client architecture for multiplayer: each player is on their own computer on a client application, and the server will contain the map model and control turn-taking.
- Keyboard input for spells and other commands (including moving to rooms, taking items, retrieving items from inventory)
- Two main game states: day mode and night mode. In day mode, a player will be able to collect items and spells used in battle encounters. In night mode, players battle with monsters.
- A game map with three type of battle encounters. After losing all HP (health points), the player will lose points and respawn on their next turn in the same location.
    - Spells and items are used to attack. These can lower the HP of both the caster and the opponent.
    - Battles against other players; winning gains points
    - Battles against "fantastic beasts" (up to 10 species) and police officers:winning gains points

1.3. Narrative Descriptions

The gameplay will take place on a 50 by 50 room map. Players can enter rooms adjacent to their current rooms. Players take turns, where they make at most 5 moves (we hope to implement more asynchronous gameplay, but this is the minimum). Moves can be either stepping into a room, collecting something, or fighting something. The game switches between day mode and night mode every 40 or so turns; during the day, spells and items will be randomly placed in rooms, which players can collect. During the night, beasts and Ministry of Magic police officers will appear; players can fight these using their spells (permanent) and their potions (one use). The player earns points for collecting beasts and defeating agents or other players. The player has a given amount of health points, and will suffer various penalties for losing to different opponents.

2. Architecture:

The main feature of our architecture is a server-client structure. There are two separate applications that use the cohttp library; the client sends request, and the server handles them. Right now, the user input from the command line interface determines when the client makes requests, starting off the pipeline inside the client. The command, if valid, gets processed and sent to the server via HTTP. The server then uses the request to update its model, or procure a diff to send back to the client so it can update itself. The user can also view the world, which is a "print to the command line" side effect in the client application.



3. System design:

3.1. Important modules

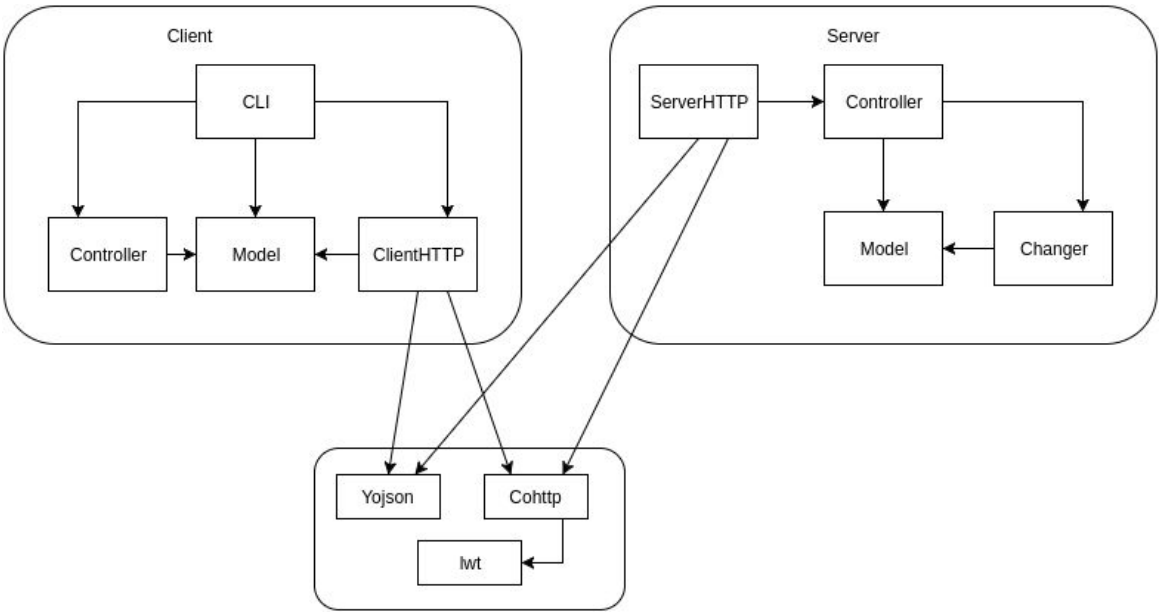
Server:

- Model: Keeps all the information necessary to capture
- Controller: Keeps the server state, which keeps a model and how far behind each server is; it keeps a list of differences each client has with the most current game state. It also validates moves against the current game states (makes sure that a change to the game state is legal) and calls changes to be made to the module.
- Changer: A module containing functions that apply diffs to the model,
- ServerHTTP (main): translates JSON to diffs and vice versa. Communicates with ClientHTTP.

Client:

- Model: represents the game state (not necessarily the most up to date game state), which can be updated via the Controller.
- Controller: updates the user's game state by applying the diff to the game state the user is seeing; keeps track of the time steps the user is currently at.
- CLI (main) : parses users' inputs and displays information about the game to users. This is currently more heavyweight than we would like, since it takes care of the logic of calculating world diffs based on user input. We would like to consult with course staff on event handling in OCaml so that more logic can be placed in Controller.
- ClientHTTP: translates diff into a json and vice versa. Also sends the json to the ServerHTTP.

3.2. Module Dependency Diagram



4. Module design:

(see .mli files submitted, and 3.1 for description of purpose)

5. Data:

5.1. Internal Data

Our data structures focus on separating static and dynamic elements of the game.

- For static elements, like effects of items or room descriptions and parts of animals, rooms, potions, spells, police, and players, we created libraries of items in the game. The immutable data are stored in a JSON file, and gets parsed to a record of the initial game state, which is represented by “world”.
- Dynamic elements of the game (e.g. other players) will store an identifier that can fetch its static information.
- The server keeps track of changes in the game by storing “diffs”, changes occurred in a move. World gets updated every time the server receives a validated change; when this happens, the up-to-date version of the world is modified, and the server keeps a copy of the diff so that the next time a client requests an update, the server can send that diff. Each diff has a timestamp. When a client asks to update, the server sends the list of diffs accumulated from the last time the client requested an update.

5.2. Communication

Two main communications are first, between the server and the clients and second, within server/client modules. The most important communication will be between different applications:

clients will communicate with servers with HTTP by sending JSON files of diffs over. Within the client/server itself, functions from other modules will be called directly (see system design for more details).

The following is a mockup of the API we will be implementing. For more details on the diff, please read the Model.mli file.

- POST /login: body containing username and password. If the password matches the game password set in the server, then the client receives a token that they need when making any other request during the game. Returns status code 200 if successful, and adds player to game.
- GET /update: The client requests to update itself to the server's most recent version. The server will then return a json file that contains a list of diffs necessary to make the client up-to-date. Requires valid sessionid.
- POST /move: body contains a diff. The client makes a move that will change the game state. The server will return status code 200 if the diff is valid: this means that it was the player's turn, and the diff makes sense (e.g. not walking off the edge of the world, using a nonexistent inventory item, etc.)

5.3. Data Structures

Libraries of items are stored in JSON file and are parsed to world records in model.ml. All data structures are used immutably to prevent the asynchronous or threading problems. We have variants to hold types of each item. Libraries, in particular, are passed into Map since it uses less asymptotic time access. Values in libraries are stored in lists since there are only small numbers (e.g. items in one room, items in the inventory, etc.), or in some cases, we need to iterate through all the values anyway.

6. External dependencies: What third-party libraries (if any) will you use?

We are currently planning on using the following:

- Cohttp: to help build the http server/client
- Yojson: to parse game state into a json file that can be sent over the server/client, as well as to store basic game state in a save file.
- lwt: a dependency of cohttp. Also makes it possible to have concurrent game play and game updating on the client

7. Testing plan:

7.1. Testing system throughout development

Throughout development, we will be testing our methods individually. For methods such as command parsing, we will test corner cases, such as: completely bogus commands (such as “cry”), commands that lead to no changes in state (such as “look” or “inventory”). We will do this by generating random inputs and writing specific corner cases (for black box and glass box testing) by hand. We will also create a test suite for all of our methods. We want to make sure that the game does not crash if the user inputs a command that does not cause any changes. We also want to make sure the model is updated accordingly after each command.

To test our server (serverhttp), we will use our browser or a specific request-maker to send a uri over so that the server sends back a json that represents the changes we want to make. We will then analyze the json to make sure that it is correct. To test our client (clienthttp), we will make a “dummy” server that essentially only reflects the json that the client sends to the server in order to make sure that the jsons the client sends to the server are correct.

We will also make a test suite to test our controllers and models, and to make sure the uri string we return when the client and server are communicating with each other is the right one.

7.2. Committing to follow testing plan

To commit to our test plan, we will write the test cases before and as we implement each method to ensure that we understand what our code is supposed to do and that it will be correct. Whoever implements a certain section will be responsible for writing the test cases and test harness for those sections, and will also be responsible for debugging. However, in the end, we will also try to look over each other’s test cases to make sure we did not miss certain corner cases. We will all “play” our final product to test the server’s performance under concurrency as well.