MS 2: Scrabble

Authors: Justin Parratt, Brian Shi, Kirk Thaker, Ram Vellanki
NetID's: jwp258, bcs84, kt485, rsv32

System Description

We developed a multiplayer (distributed) Scrabble game that can be played with either human or computer players complete with the following features.

Key features:

- AI
- Web GUI
- multiplayer (distributed)
- security with generated API keys (detailed in the **Additional Information** section)
- Chat (Instant Messaging)
- Keep-Alive mechanism for Server Sent Events (detailed in the **Additional Information** section)
- Logging mechanism for viewing server data (formatted as a quasi-HAR file and detailed in the **Additional Information** section)
- English (not OCaml) dictionary manipulation using a Trie

We created a Scrabble game which allows users to play Scrabble, but with added features such as detecting if words are valid or not. We also implemented an AI to play words that maximize score according to tile/word bonuses and point values of letters. Please note that the AI is intentionally delayed so as to allow human users to view the AI moves sequentially.

The scrabble dictionary was implemented by using a prefix tree because the efficiency of the data structure is especially important for the AI.

For the server-client interface, we leveraged several OCaml packages (listed in the **External Dependencies** section of this document) to persist data across multiple players and multiple instances of games as well as provide an HTTP interface for multiplayer functionality over the internet. Our server is exposed over a public IP address such that any system on Cornell's network can access and play our game via our web application. The API for this server is documented on http://docs.scrabble1.apiary.io.

For the user interface, users are able to view their current available letters as well as the current board and are able to perform moves.

With regards to the comment on the usefulness of a trie - we felt that a trie was necessary because the AI would be doing lookup of many words when evaluating potential moves. A trie also allows the AI to quickly determine what words can be made from an existing word on the board by adding some additional tiles to the end of the word. This is elaborated in more detail in the **Data** section of this document.

Please note that we provided extensive detailed design information in additional sections at the bottom of this document (this organizational suggestion of adding additional sections was given to us by Professor Clarkson via Piazza)

Citations

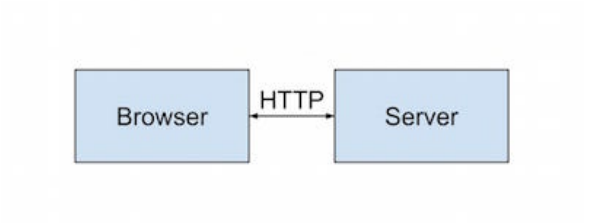
- OCaml documentation http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html
- Js_of_ocaml documentation https://ocsigen.org/js_of_ocaml/
- Js_of_ocaml source code https://github.com/ocsigen/js_of_ocaml/

- Ocamlbuild documentation https://github.com/ocaml/ocamlbuild
- Cohttp source code and documentation https://github.com/mirage/ocaml-cohttp
- Lwt source code and documentation https://github.com/ocsigen/lwt
- MDL documentation https://getmdl.io/components/
- JavaScript documentation https://developer.mozilla.org/en-US/docs/Web/JavaScript
- *The world's fastest scrabble program* https://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/scrabble.pdf
- API Key Security Guide http://billpatrianakos.me/blog/2013/09/12/securing-api-keys-in-a-client-side-javascript-app/

Architecture

There are two primary components (units of execution) in this system.

- Server - provides an interface for accessing/modifying game state over HTTP
- Client (Browser) - provides an interface for playing a game through a GUI

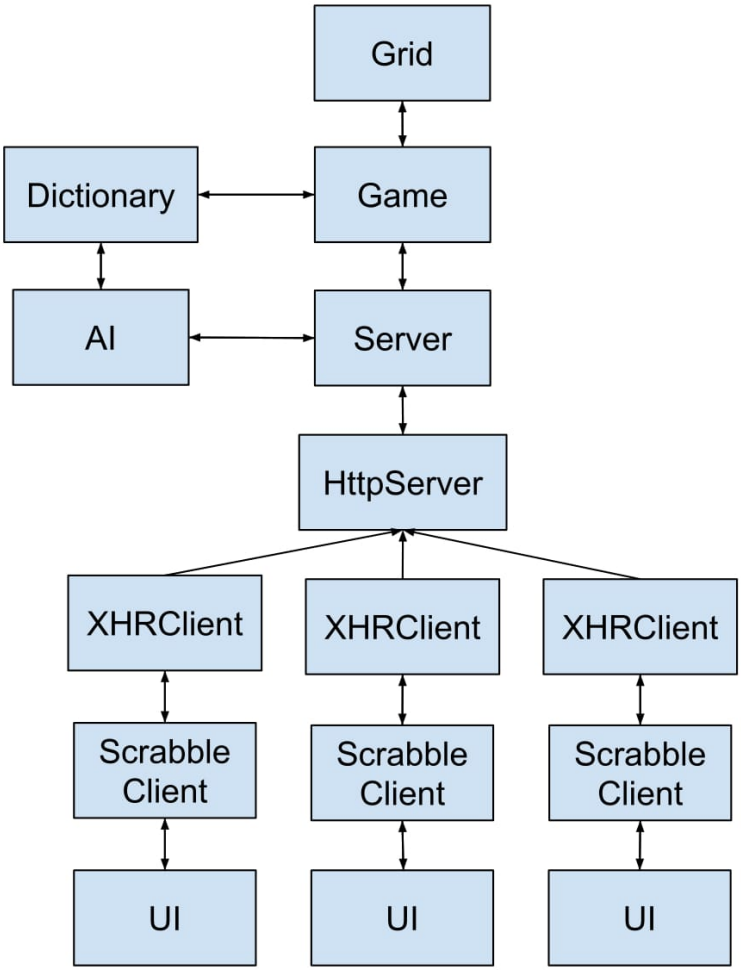


System Design

Please see the Module Dependency Diagram on the next page.

Grid - 2-D representation of board characters (including coordinates of score tiles)
Dictionary - English dictionary stored in a data structure (implemented as a Trie)
AI - logic to generate a move based on an input state (uses **Dictionary** to find a valid move)
Game - logic to perform actions in a game, such as adding/removing players and performing a move (using **Dictionary** to validate)
GUI - converts GUI input (from the browser) to moves, uses the Client to process those moves, and reflects changes in the GUI by updating a graphical scrabble board (no .mli file provided because it is the entry point of the application)
HttpServer - an abstraction of **cohttp** for simpler development of HTTP server APIs
XHRClient - an abstraction of **cohttp** for simpler development of XML HTTP requests
Server - accepts moves in the form of HTTP requests then uses Game to produce a new game state, returns the changes, and saves changes to the database, this module issues Server-Side Events to clients as a means of more efficient updates for game state
ScrabbleClient - communicates with the Server's HTTP API to join games, create games, retrieve the changes in game state etc. (i.e. serves as an abstraction to make the separation between client and server seamless); this module also leverages Server-Side Events for more efficient updates of game state

In order to provide a more comprehensive design of our HTTP API (specifically the **Server** module above), we have developed an outline of our API: http://docs.scrabble1.apiary.io



Module Design

The .mli files are provided in the code submission.

Detailed module design descriptions can be found above in the **System Design** section.

Data

The system maintains persistent data for game information including its players, remaining tiles, turn counting, and of course the current board state. Here are some of the more intricate details of the data representations and transfer.

- OCaml Memory - information such as game state and players etc. are stored in memory in OCaml List structures (this data representation suffices for our efficiency constraints given that we are unlikely to scale beyond 10,000 simultaneous games)
- HTTP API - data transferred as JSON
- Dictionary - a trie contains the dictionary data (the efficiency of this data structure is particularly useful for the AI). Specifically, we insert the English dictionary into two different tries - one where words are in normal order, and one where words are in backwards order (e.g. OCaml would be inserted into the trie as l-m-a-C-O). The forwards-trie is used for word validation as well as for the AI to find words where letters can be added onto the end of an existing word. For example, adding 'd' onto the end of 'dance' yields the new word 'danced'. The backwards-trie would be used for the AI to find words where letters can be added onto the front of an existing word. For example, adding 'un' onto the front of 'balanced' yields the new word 'unbalanced'. This cannot be done with the forwards trie, since it would be very slow to figure out the words that have 'balanced' at the end.

* Note that we chose to use OCaml memory over persistent data storage mechanism such as databases with a query language such as SQL due to the ease of development, efficiency, and the stronger security guarantees due to avoiding

issues such as SQL injection. The tradeoff of this approach involved a heavy use of mutable features to maintain the state on the server.

External Dependencies

- ounit - for unit testing
- lwt (**2.6.0**) - for concurrent programming
- cohttp - for the HTTP client and server
- yojson - for serializing/deserializing data to and from JSON
- js_of_ocaml - for developing a browser-compatible GUI
- str - for easier string processing
- NodeJS (not an opam package) - for serving web pages and forwarding requests to the OCaml server

* use `opam update` and `opam upgrade` to fix unbound value compile errors

Testing

Test Plan

All the non-web modules (e.g. board, dictionary, ai, and game) are tested using black-box and glass-box unit tests. To ensure accountability, the black-box tests for a module were written by someone who did not work on it (i.e. cross-testing). The GUI was be tested by alpha and beta testing including individuals outside of CS 3110. The HttpServer and XHRClient modules and remaining web related modules (e.g. ScrabbleClient, and Server) were tested via integration to the primary system (this was done as opposed to developing mock components to test each individual web component as a result of a lack of time).

Test Results

Running `make test` in the 'server' directory indicates that all of our tests pass which lends a high confidence in the correctness of our software.

Known Problems

- only guaranteed to work on Chrome
- may experience issues with the view if the screen is too small (simple fix is adjusting the zoom on the page)
- no error handling/fault tolerance for loss of connection (e.g. WiFi dropping out) due to the nature of the `EventSource` API but the server will not crash as a result of any of these issues (this includes unclean exits from the webpage including sleeping a computer)
- some security vulnerabilities (as detailed above)
- opening multiple instances of the game on one machine within one browser (i.e. in different tabs) will **NOT** work due to the limitations of 6 `EventSource` instances per browser (may work up to 2-3 tabs at most but only guaranteed for 1 tab) and may even break the individual game attempting to be joined but will leave all other games on the server unaffected (i.e. can still create a fresh completely functional game)
- there do exist some thread safety issues in that multiple people joining a single game too quickly can lead to slight UI inconsistencies but this could easily be solved with more time by implementing a locking mechanism such that these types of requests are processed sequentially

Division of Labor

- Justin implemented the Grid and Dictionary modules (commits aren't logged correctly because he was commiting through the virtual machine). Justin spent approximately **30 hours** on the project.
- Brian implemented Game module. Brian spent approximately **30 hours** on the project. See the code/programming section for more detail on implementation.

- Ram implemented the Server-Client interface for sending data between different clients over the HTTP interface (including the chat instant messaging system). He also developed the front-end browser GUI to display the data received from the server as well as provide an interface for the end-user to interact with the game and effect changes to the game state. Ram spent approximately **150 hours** on the project.
- Kirk implemented the AI (and can also verify that Justin did quite a bit of work, git was just acting weird on Justin's VM). Kirk spent approximately **30 hours** on the project.

Additional Information

Usage Details

General

- drag tiles from the player tiles onto the board to lay out your move
- select individual player tiles (highlighted white) in order to replace those tiles

Visual Updates

- the highlighted player in the score table is the player who's turn it is
- snackbar notifications at the bottom of the screen display relevant game information such as joining and leaving players

Buttons

- Place - place the tiles you have dragged on
- Reset - reset all placed tiles from the board back to the player tiles
- Pass - pass your turn
- Replace - replace the selected player tiles (highlighted white)

Scrabble Rules

We followed most of the official rules of Scrabble (found on the official website), but made some slight modifications to the ruleset. When ending the game, we enforced that the game could be ended when 6 scoreless turns occur, rather than when "no more moves could be made". We also decided that when choosing a letter for the wildcard, you cannot choose a letter that currently is on your tile rack. For example, if your tile rack is ABCDEF?, and you want to change your ? into a F, you must first place the F on the board before changing the ? to a F. We also allowed blank tiles to be scored by the value of the tile that they transformed to, rather than making it 0.

Code Design/Programming

Types are sometimes specified to assist OCaml in overcoming the pitfalls of weak type inference with records or refs and also to improve code clarity in some cases.

Each individual module was implemented **bottom-up** while the overall project was implemented **top-down** in order to allow for parallel development of modules while stubbing out dependencies.

UI

- the AI is intentionally delayed so as to allow human users to view the AI moves sequentially
- used 'id' even for repeated tags because of limitations of `Js_of_ocaml` in searching through an HTML table
- used viewport pixels to scale well with multiple screen sizes
- newly joined users may not view older chats only newly received chats (design decision of messaging system)
- attempts to load the `scrabble.html` page without joining/creating a game will lead to an automatic redirection to the `index.html` (main) page

Server-Client

OCaml

- used association lists (as opposed to a `Map` or `Hashtbl`) as the primary data structure for storing key-value pairings despite the O(n) lookup time because all lists on the server are consistently < 10,000 in length (efficiency traded off in favor of simplicity)
- `refs` were employed extensively in order to save state on the server side because immutable abstractions are impossible when each individual endpoint callback is process independently in its own scope and own thread (alternative would be to use a database such as `sqlite3` for persistent storage but again, efficiency was traded off in favor of simplicity)

Web Security

- API keys - upon joining/creating a game an API key is generated by hashing the result of `player name + game name + current system time + secret string on the server` (all subsequent requests to modify or act as that player in that game must include this API key or else the server responds with a 401 status code indicating an unauthorized request); this approach was inspired by http://billpatrianakos.me/blog/2013/09/12/securing-api-keys-in-a-client-side-javascript-app/
- web application is only accessible on Cornell's network for security reasons
- browser CORS security protects against malicious rewritten JavaScript requests
- prevented DOM manipulation to influence HTTP requests by deterring inspect element functionality via disabling right-click and certain shortcut key strokes
- concealing other player's tiles in server-sent payloads involving game state

Security Summary:

- Parameter Attacks - **protected**: these types of attacks are infeasible (no additional query language such as SQL is used so no vulnerability such as SQL injection is present)
- Identity Attacks - **protected**: because API key is generated on creating/joining by hashing a combination of information such as the user-agent, the current time, and a secret key (also obfuscated! because of the nature of js_of_ocaml first compiling to bytecode then transpiling to JS)
- Man in the Middle - **vulnerable**: because the domain is net secured by SSL (simple improvement would be to dish out cash for this)
- DoS - **vulnerable**: no human authentication for creating new games (simply fix by integrating reCAPTCHA)
- DOM - **semi-protected**: inspect element deterrent with right click and keyboard shortcut disable

Web

- requests to the server are logged at `<URL>/server_log.txt` in a format similar to a HAR (HTTP Archive) file and game logging is saved to `<URL>/game_log.txt` (an additional improvement would be to secure these two files behind a password protected authentication system)
- there is slight duplication of files between the 'public/js' and 'server' directories due to modules that the client employs to organize information more cleanly such as using the **Grid** module to obtain the bonus tiles of the board and using the **Game** JSON serialization functions to more easily interpret game information sent from the server and send game information to the server
- certain pieces in the 'scrabble.ml' OCaml file are written with pure JavaScript bindings due to the limitations of js_of_ocaml (e.g. displaying the dialog for selecting wildcard tile values)
- due to Cornell's hosting only opening port 80, all requests to the uri "/api" on port 80 are forwarded to an internal OCaml server running on port 8000
- used SSE (Server-Sent Events) as opposed to polling techniques (for fast downstream game updates and messages) and implemented a keep alive mechanism that pings clients every ~ 15s to maintain wakefulness of client EventSources that wait to read data from these Server-Sent Events
  * chose not to use WebSockets because of the minimal support for OCaml
  * still lacks EventSource reconnection (mentioned in the known problems)
- used `localStorage` to transfer data between the two pages of the web application

- used 200 HTTP status code for all successful operations for simplicity and ease of development given time constraints (as opposed to employing other more descriptive status code such as 201)
- `HttpServer` and `HttpClient` modules provide clean interfaces that abstract what would've been duplicated code between HTTP endpoints/requests such that a callback function simply needs to accept a request record and produce a response record

`HttpServer` Usage:

```
HttpServer.add_route (<HTTP_METHOD>,<ROUTE_ENDPOINT>) <CALLBACK>
e.g. HttpServer.add_route (`GET,"/api/") callback
```

Dictionary

- used a `Map` to represent the children of a trie node to increase simplicity and speed of word lookups
- `refs` were used while building the tries to increase ease of use of the file reading mechanism

Grid

- `lists` were used to represent the grid and bonus tiles for simplicity (list sizes are small so efficiency loss is not drastic)

Game

- used records to represent a player, game diff, move, and game state.
- state and player are mutable records because the game information is not stored in a database (e.g. SQL) but rather in memory. We would have preferred to do this immutably, but the difficulty in using SQL with OCaml made us decide to just have the game information stored in memory and use mutable records.

Implementing functions like add/remove player and create game were very straightforward and simple. The execute function was the main challenge. The first step of execute was to take the tiles placed and determine the direction that the tiles were placed on (horizontal or vertical). When a single character is placed, the direction that the user intended to make the new word in needs to be inferred. Once the direction was inferred, all the words formed by the placement of tiles are collected. First, all the words formed perpendicular to the direction of the main word are collected. For example, if a word is placed horizontally, the vertical words formed by each tile placed are collected. Then the main word along the tiles placed is collected. If when doing this, there is a coordinate with no preexisting tile or tile placed, then the move will fail. The total score is determined in the same way that words are collected. A move is interpreted as a pass if the tiles placed is empty, and a swap if the swaps list is nonempty.

AI

- Used for loops to efficiently and simply traverse the board (which is a 2-dimensional list). Other than that, nothing special was used (only lists, tuples, and some variant/record data types). And for loops were only used occasionally (mostly List.fold_left).

Although we used the Scrabble AI paper listed in the citations as inspiration, not all of the algorithms described in it were implemented because it was simpler and far more clear to do otherwise and remain efficient.

In fact, the AI always finds moves in sub-second times - the delay that is presented by the UI is added externally because if the AI moves at full speed we can't tell which player (in the web UI) makes which moves!

The key insights to building the scrabble AI were as follows:

- Scrabble moves have to be adjacent to an existing word (except if it's the first move)
- The limiting factor in generating move permutations is both our tile set and the efficiency of dictionary lookups.

The scrabble AI works as follows: 1. Identifies all "slots" on the board. Slots are simply (row,col) coordinates that are adjacent to an existing word.

1. For each slot, figure out what characters from our tile list simply cannot go there because they form invalid cross words.
2. The generated association list of (row, col) paired with the valid chars that can go there is called an anchor list.
3. For each anchor in our anchor list, the AI attempts to build a word in all 4 directions. It uses the anchor list to cut down on permutations early and uses the function Dictionary.has_extension (or Dictionary.has_back_extensions if building backwards) to further cut down on permutations.
4. The final list of possible moves is accumulated and ranked based on tile values and board bonuses found in Game.
5. The best move is selected and returned, or if no move is possible, a GameOver exception is raised.
6. It is worth noting that raising a GameOver exception does not necessarily mean the game is over. It could also be the case that our AI in particular simply has no more moves to make because it has a "bad" tile list. The functions that call Ai.best_move account for this peculiarity.

Dictionary

There are two dictionaries in this project, (the official scrabble dictionary, and a dictionary of all words backwards). The trie was implemented as follows:

- Each node represents a character in a word, a Map of all characters that can come next in a valid word, and a boolean which determines whether or not the current character marks the end of a valid word

Adding a word to a trie works as follows:

- In the case that the first character of the word does not exist in the children of the root, the word is recursively added letter by letter to the trie
- In the case that the first character already exists in children of the root, the rest of the word is added to the children of that node recursively (for example if the trie contains h-e-l-p and h-e-l-l-o is added, the word e-l-l-o will be added to the 'h' node) until there is a difference between the words at which point the new character is added to the map of children nodes (continuing our example the first 'l' in hello now has children 'l' and 'p').

The boolean in each trie node exists to determine whether the current letter marks the end of a valid word, even if it is not a leaf node (for example in the word "racecar" the 'e' and second 'r' character both mark the end of a valid word)