CS5003 — MASTERS PROGRAMMING PROJECTS

CS5003 PRACTICAL 2: GIN RUMMY

Deadline: Friday the 19th March 2020 at 21:00

Credits: 33% of the coursework (and the module)

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

AIMS

The main aim of this project is to write a complete web application including both client and server components. More specifically, it will involve designing and testing an API, implementing the front-end and back-end in JavaScript which communicate through this API, and choosing the right data representation.

Your client-side JavaScript should be written in modern JavaScript (e.g. ECMAScript version 6+) with appropriate HTML, CSS and optionally VueJS. Your server-side should be written in Node.js, using specified packages such as Express. You will be provided with a package.json file specifying which packages are permitted. Do **not** install and use any other packages without asking the lecturer first. Do not use any other language, JavaScript dialect or templating language (e.g. TypeScript, CoffeeScript, HAML, Pug, React, ejs, etc.).

The use of git version control system is mandatory for this practical. You will be expected to use it throughout the development to keep track of progress.

OVERVIEW

You will create an online game that allows different users to connect and to play against each other. You will implement a simplified version of *Gin Rummy*, in which players take it in turn to draw cards from the *stock* or *discard pile* in an attempt to complete *melds*. The game is played with a standard 52-card deck, using the standard ranking of cards from Ace (low) to King (high). There are a number of different variations on the classic Gin Rummy game (https://en.wikipedia.org/wiki/Gin_rummy), but you will implement a custom variation for this practical. You will start with a simple two-player, win-lose version and expand from there. You can find a list of terms and the detail of the game rules for this practical in the Game section.

Working in pairs, you will implement both the client side (HTML+CSS+JS) and the server side (JavaScript based on Node.js), The server side will implement a RESTful API for exchanging data with the client. Your webpage will contain client-side JavaScript which makes HTTP calls to the API, and exchanges data with the server using JSON.

The server should maintain the stock of cards, discard pile, and whose turn it is to play (to prevent clients cheating). The API should provide services including, but not limited to:

- starting a new game (including allowing players to join, shuffling the deck, and dealing hands to the players)
- drawing the top card from the deck
- adding a card to a player's hand
- discarding a card from a player's hand to the discard pile
- declaring Gin (including submitting melds)
- configuring the game (player names, number of rounds, whether knocking is allowed or not, whether aces can be high or low, etc.)

The client should provide a user interface allowing the user to connect to a game, draw and discard cards, form melds, and declare Gin.

GAME

TERMS

Deadwood
: Cards in a player's **hand** that have not been put into a **meld**.

Discard pile
: Cards that players have discarded from their hands. The top card (AKA the **upcard**) can bee seen by all players. During their turn, players can draw the upcard into their hand.

Gin
: A declaration made by a player when *all* their cards are in valid **melds**. Gin signals the end of the game. A player who incorrectly declares Gin loses the game.

Hand
: The cards belonging to a player. For a two-player game, a hand comprises 10 cards each. For a multi-player game, a hand comprises 7 cards each.

Knocking
: A declaration made by a player when *some of* their cards are in valid **melds**. Knocking signals the end of the game. The player with the highest score at the point of knocking wins the game. (Used in intermediate requirements).

Meld
: A run of 3 or more consecutive cards of the same suit, or a set of 3 or more cards with the same rank.

Scoring
: Ace is worth 1 point, face cards are worth 10 points, all other cards are worth their numerical value. A player's score if the sum of the values of their **deadwood** cards. (Used in intermediate requirements).

Stock
: Cards that are not in players' hands or the discard pile. Stock cards are not visible to players. During their turn, players can draw the top card from the stock into their hand.

GAME PLAY

The basic game is played as follows:

- Once enough players have joined a game, all players are dealt a hand of cards.
- The top card in the stock is moved to the discard pile becoming the upcard.
- The player whose turn it is can either:
    - declare Gin and submit their melds. At this point the game is over.

      OR

    - draw a card into their hand (either the upcard or the top card from the stock) and discard a card from their hand (this could be the card they just drew or any other card in their hand). Once the player has discarded a card, play moves to the next player.
- Play continues until a player has declared Gin.

The Intermediate and Advanced requirements add to the basic game play, and details can be found in the relevant subsections of the Requirements Section.

REQUIREMENTS

Your application should provide the functionality described below. Many of these requirements are open to interpretation and there are many different ways to fulfil them. You are encouraged to speak the lecturer if you have any doubts or questions about any of the requirements.

You should make sure you have completed all the requirements in each section before moving onto the next.

BASIC

Your application should provide the following:

- Allow a player to start/join a game.
- Allow a player to see the current state of their game, including the cards in their own hand and the upcard.
- Allow a player to draw a card into their hand (either the upcard or the top card from the stock) and discard a card from their hand.
- Allow a player to declare Gin and submit their chosen melds.
- Reveal all players' hands once the game is over and give an appropriate win/lose message.
- The server should ensure that players cannot 'cheat', either by drawing cards when it is not their turn, drawing cards that are not avaiable, or claiming to have cards that are not in their hands.

HINTS:

- You can assign each game and each player in the game a universally unique id. The ids can be used by the server to identify the player/game. To give a more user-friendly experience the ids could be hidden from the user.

- Clients will need to repeatedly *poll* the server to get updates when it is not their turn. Promises, timeouts and async/await can be very useful to achieve this. To get started you can check https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

INTERMEDIATE

Your application should provide all the requirements described in the Basic Section, plus the following:

- Allow a player to enter their name. Note: you can assume the name is lost when the user refreshes the page if you wish.
- Keep track of a player's wins and losses and display the current user's statistics to them.
- Allow aces to be high *or* low within the game, meaning [A♠ 2♠ 3♠] and [Q♠ K♠ A♠] are valid runs. Runs "around the corner" are **not** allowed, meaning [K♠ A♠ 2♠] is not a valid run.
- Allow a player to *knock* and submit their chosen melds. The opposing player must then submit their chosen melds. The player with the *lower* score wins the game. If both player's have the same score, the defending player (who did not knock) wins the game.
- Detect if a player has been inactive for a set period of time during their turn (e.g at least a minute). If they have, assume the player has left the game, and make the remaining player the winner regardless of the score.

ADVANCED

Your application should provide all the requirements described in the Basic and Intermediate Sections, plus the following:

- Introduce multi-round games where the goal is to reach a specific score (e.g. 100). The first player to reach (or exceed) that score wins the game. At the end of each round (i.e. after a player Knocks or declares Gin), players' scores are calculated and added to their running total for the game. If a no player has reached the prescribed number of points, the deck is shuffled, hands are dealt, and a new round begins. Bonus points are awarded as follows:
    - If a player successfully declares Gin, they get a bonus 25 points.
    - If a knocking player gets a lower score than the defender (the one who did not knock), the defender gets 25 bonus points.
- Allow a player to chose the number of players in a game. If there are more than 2 players, each player should have 7 cards in their hand rather than 10. Make sure you consider the implications on win/lose scenarios, and players leaving games early.
- Allow games to be recorded and replayed.
- Allow players to register/login/logout. Any new games they play should be added to their all-time score record. You can assume all records are lost when the server is reset.

REPORT

You should write a short **joint** report (approximately 2000 words) detailing the design of your solution. You should include the following sections:

- **Overview** — a brief description of what has been achieved in the submission. This should include a list of which requirements have been met and to what extent, including the ID of the person who was primarily responsible for each feature .
- **Technologies & Resources** — a brief description of what external technologies and resources you used during the development of your solution and what you used them for. This should include references to any external CSS libraries, tutorials, forums, websites, books, etc. that you used to produce your solution. It should also clearly state what browser(s) were used to run and test your solution, and what version of NodeJS was used for the development.
- **Design** — a brief discussion of any interpretation of requirements and any design decisions taken. Focus on the *reasons for your decisions* rather than just providing a description of what you did.
- **Team Work** — a brief description of your approach to team work, including what tools and techniques you made use of (e.g. Trello, pair programming, etc), how you allocated and tracked work within the team, and any up-skilling of team members.
- **Evaluation** — a brief reflection on the success of your application. What went well and what could be improved? What might you do differently next time or if you had more time?

You would also write a short **individual** report (around 1000 words) describing your own contribution and your own challenges.

DELIVERABLES

A single `.zip` file must be submitted electronically via MMS by the deadline. It should contain:

- The entire working copy of your repository, containing the source code and revision history. Note: the node-modules folder should **not** be included in this.
- Your **joint** report in **PDF** format.
- Your **individual** report in **PDF** format.
- A short README file describing how to run your server.

Submissions in any other formats may be rejected.

MARKING CRITERIA

Marking will follow the guidelines given in the school student handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback. html#Mark_Descriptor

Your submission will not be evaluated based on aesthetic appeal (this is not a visual design course), but use of CSS and DOM scripting which enhances the experience and interactivity will be rewarded.

The mark will be based on the quality of the final application and joint report. This mark may be adjusted for each member of the group based on their individual contribution to the project. Individual contributions will be evaluated using the individual reports and git repository, and will take into account the amount and complexity of features tackled by the individual and their participation in team-work.

Some specific descriptors for the **group component** of the assignment are given below:

- A **poor implementation in the 0 – 7 grade band** will be missing nearly all required functionality. It may contain code attempting a significant part of a solution, but with little success, together

with a report describing the problems and the attempts made at a solution.

- A **reasonable implementation in the 8 – 10 grade band** should provide some of the functionality described in the Basic Section, and demonstrate reasonable use of HTML, Javascript and Node.js. The code should be documented well enough to allow the marker to understand the logic. The report should describe what was done but might lack detail or clarity.
- A **competent implementation in the 11 – 13 grade band** should provide all the functionality described in the Basic Section, demonstrate competent use of HTML and CSS (e.g. for layout and positioning), allowing players to play a complete game against an online opponent. The code should be documented well enough to allow the marker to understand the logic and should have a modular design. The report should describe clearly what was done, with good style.
- A **good implementation in the 14 – 16 range** should provide all the functionality described in the Basic Section and some or all of the functionality from the Intermediate Section. It should demonstrate good code quality, good comments, modular design, and proper error handling. All of the JSON objects passed through the API must be checked and validated. Good use of CSS for layout and styling is expected for a submission in this range (not unmodified defaults). The report should describe clearly what was done with some justification for decision, with good style, showing a good level of understanding.
- An **excellent implementation in the 17 and higher range** should provide all the functionality described in the Basic and Intermediate Sections, and some or all of the functionality from the Advanced Section. It should demonstrate high-quality code and be accompanied by a clear and well-written report showing real insight into the subject matter.

Note: For this practical you do not need to invent your own extensions. Concentrate on providing high quality, sophisticated implementations of the requirements in this specification and writing an insightful report demonstrating understanding.

It is anticipated that in most pairs, individuals will receive the same mark. However, where there is a substantial difference in contributions marks may be adjusted up or down. The following are examples of when the mark may be adjusted.

- A **poor contribution** will have
  - a missing or weak individual report, demonstrating confusion and misunderstanding of the topic. Little or no contribution to the final software, adding little to no value, focussing on only one part of the program functionality.
  - OR a sizeable contribution to the code but a failure to engage in the teamwork process.
- An **excellent contribution** will have an excellent individual report, regular and sizeable code contributions throughout the project, contribution to many important parts of the project, and clear engagement in the team working process.

WORD LIMIT

An advisory word limit of approximately 2000 words for the group report and 1000 words for the individual report, excluding references and appendices, applies to this assignment. No automatic penalties will be applied based on report length but your mark may still be affected if the report is short and lacking in detail or long and lacking focus or clarity of expression.

A word count must be provided at the start of the reports.

LATENESS PENALTY

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

GOOD ACADEMIC PRACTICE

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/

HINTS AND SUGGESTIONS

Start by spending some time on design – what needs to be done to meet the basic specification. What type of data needs to be represented and where (e.g. client or server). What is the best way to represent the data – objects, classes, JSON objects, etc. Write a basic outline of your object/class structure and RESTful API on paper and think whether it makes sense. It pays to spend time with your partner and discuss this in detail before starting to code.

Divide the work into small chunks and decide how to split the work between the two of you. Agree on a rough work plan, and meet regularly to see how the work is progressing – sometimes you will need to adjust because things turn out to be more difficult than planned.

Have the basic requirements working fully before venturing into intermediate or advanced requirements. A clean, fully functional and well-written basic implementation is better than a buggy mess with extensions.

Avoid splitting the work into parts that are mostly separate (like only client-side and only server-side) and working on them independently. Integrating such work at the very end is likely to be difficult. Try to work on related aspects – like splitting the API between the two of you, or working on different aspects of the user interface, or participating in pair-programming. Integrate work at regular intervals so you can look at each other's code from time to time. It will help you understand your partner's thinking, help spot problems, and help you pick up good habits and tricks from each other.

Make use of existing resources where available – but make sure that there are no licensing issues and that you credit external sources. A very useful resource for graphics is https://openclipart.org/.

FINALLY

Don't forget to enjoy yourselves and use the opportunity to experiment and learn! If you have any questions or problems please let me know (ruth.letham@st-andrews.ac.uk) — don't suffer in silence!

Author: Dr Ruth Letham

Module: CS5003 – Practical 2: Gin Rummy