CS5003 — MASTERS PROGRAMMING PROJECTS

CS5003 PRACTICAL 1: MEMORY GAME

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

Credits: 22% 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 clearly structured single-page web application. You will practice using APIs to access remote services, manipulating json and arrays, and incorporating the results into your web application.

DETAILS

In this project, you will create a web version of a single-player pair-matching memory game, similar to the card game Concentration, in which the player tries to find matching pairs of images.

Your single-page web application will present the player with a number of ‘boxes’. The player will repeatedly ‘open’ pairs of boxes, revealing the images inside. If the images match, the boxes remain open. If the images do not match, the boxes are ‘closed’. The game is over once all matches are found and all boxes remain open.

Your application should consist of client-side JavaScript, using appropriate API calls to provide suitable images from https://docs.thedogapi.com. You will need to investigate and choose which endpoints and parameters you will need. You will use version 1 of the API, using the base URL **https://api.thedogapi.com/v1/** for all api calls. Please note that the Dog API has a number of API endpoints that require an API key, however, you can fully complete the practical **without signing up** for one.

Your submission should be written in modern JavaScript (i.e. ECMAScript version 6+) with appropriate HTML and CSS. You may use pure CSS frameworks to improve the look of your web application, but you must not use any JavaScript libraries or any other language, JavaScript dialect or templating language (e.g. jQuery, TypeScript, CoffeeScript, HAML, Pug, React, Vue, ejs, etc.).

REQUIREMENTS

Your application should provide the functionality described below. You must make sure you have completed all the requirements in each section before moving onto the next.

BASIC

Your application should provide the following:

- Implement a basic pair-matching game comprising 12 ‘boxes’:
    - Fetch 6 random images from the Dog API and randomly associate each of these images with two of your boxes (Figure 1).
    - Allow the player to ‘open’ two boxes (e.g. by clicking on them) and reveal the image that is associated with each (Figure 2).
    - After the second box in the pair is opened, detect whether the images match — if they do not ‘close’ both boxes (by hiding the image).
    - Allow the player to repeatedly ‘open’ pairs of boxes until all matches have been found (Figure 3).

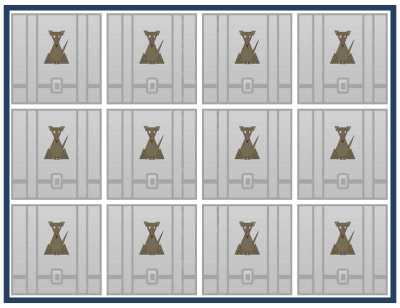
Figure 1: Basic game with all boxes closed.

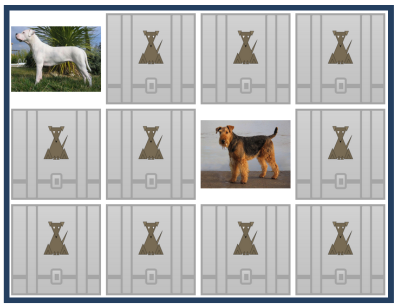
Figure 2: Basic game with two non-matching boxes open.

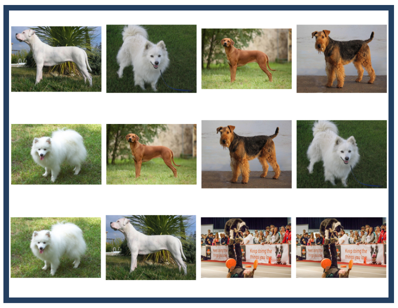
Figure 3: Completed basic game with all boxes open.

- Keep track of how many pairs of boxes the player opens in a game.
- When a game is over, let player start a new game (with a new random set of dogs) without refreshing the page.
- Track and display the player's ‘best’ score so far (i.e. the lowest number of pairs they needed to open to find all matches). The score should persist until the player refreshes the page.

INTERMEDIATE

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

- Let the player select the ‘difficulty’ of the game, with higher difficulty levels having more boxes to match — e.g. easy = 8 boxes (4 dogs), normal = 12 boxes (6 dogs), hard = 18 boxes (9 dogs). Make sure you keep track of the player's best score for each difficulty level.
- Before a game begins, list all available breeds (fetched from the API) and let player select which breed they want the game to feature. If they player has selected a breed, only include dogs of that

breed in the game. If the player has not selected a breed, include a random selection of any dogs in the game.

  - Note: some breeds do not have enough images to build a full game. Make sure your program checks for this and copes gracefully with it (i.e. do not start a game with too few dogs).

- Paginate the breeds list so the player can only see 10 breeds at a time. Allow the player to load the previous and next pages of breeds.
- Set and display a time limit on a game. If the player does not make all the matches within the time limit the game is over and they do not get a score. Remember to increase the time limit for more difficult games.

ADVANCED

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

- Allow the player to select multiple breeds to be included in the game. Make sure your program copes gracefully if the number of breeds does not match the number of dogs. For example:
  - If the player selects 2 breeds for a game that has 6 dogs – i.e. 12 boxes – you should include 3 of each breed.
  - If the player selects 4 breeds for a game that has 6 dogs you should include 2 dogs of each of the first 2 breeds and 1 dog of each of the remaining 4 breeds.
  - If the player selects 8 breeds for a game that has 6 dogs you should include 1 dog of each of the first 6 breeds and 0 dogs of each of the remaining 2 breeds.

  Remember to cope gracefully if there are not enough dogs across the selected breeds to build a full game.
- After a match has been found, let the player click on the image and see the breed information.
- Research and use *localstorage* to persist high scores when the page is refreshed.
- Research and use closure, modules and/or classes to improve code structure (and prevent players cheating using the browser console!). Think carefully about the major elements in your program (e.g. the timer, the cards/dogs, the breed list, the game itself) and define classes or closures to represent these distinct concepts, including their related data and operations.

REPORT

You should write a short report (approximately 1000 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.
- **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.

- **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.
- **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?

DELIVERABLES

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

- The source code for your application, including your CSS, HTML and Javascript.
- Your report in **PDF** format

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. Some specific descriptors for this assignment are given below:

- A **poor implementation in the 0–7 mark 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 mark band** should provide some of the functionality described in Section Basic, and demonstrate reasonable use of HTML and JavaScript. 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 mark band** should provide all the functionality described in Section Basic, demonstrate competent use of HTML and CSS (e.g. for layout and positioning), allowing a player to complete multiple games and see their best score during the current session. The code should be documented well enough to allow the marker to understand the logic and should be contained in JavaScript files that are separate from the HTML code. The report should describe clearly what was done, with good style.
- A **good implementation in the 14–16 mark band** should provide all the functionality described in Section Basic and some or all of the functionality from Section Intermediate. It should demonstrate good code quality, good comments, and proper error handling. 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 mark band** should provide all the functionality described in Sections Basic and Intermediate, and some or all of the functionality from Section Advanced. 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.

WORD LIMIT

An advisory word limit of approximately 1000 words, 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 report.

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/

Author: Dr Ruth Letham *Module*: CS5003 – P1: Memory Game