The Happy Cow Game

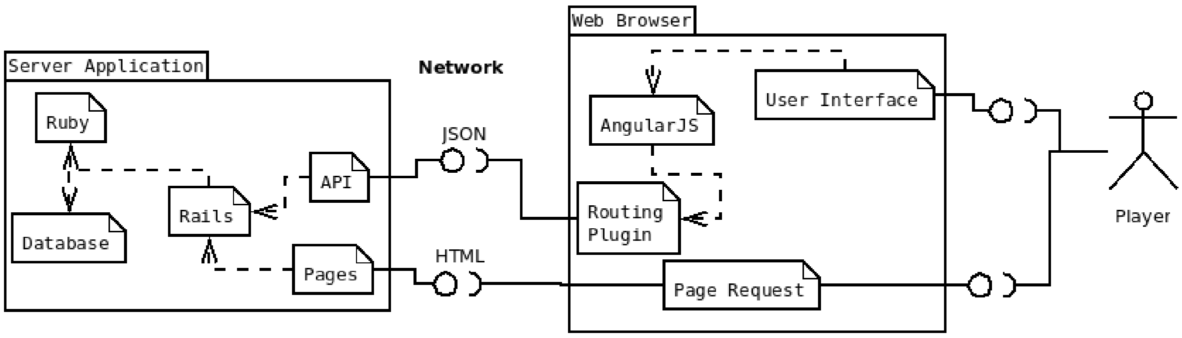
Report Name	Design Specification
Author (User Id)	Simeon Smith (sis17)
Supervisor (User Id)	Amanda Clare (afc)
Module	CS39440
Degree Scheme	G401 (Computer Science (Inc Integrated Industrial And Professional Training))
Date	February 17, 2015
Revision	0.1
Status	Draft

1 Introduction

The purpose of this document is to consider the project as an entire working system, and then break down the detailed aspects into easily manageable sections. The requirements will be transformed into the necessary data and client side structure, facilitating the creation a working application.

2 Game Applications

This design outlines two applications necessary for this project. Using two applications separates game logic and data persistence from user interaction and interface desgin.



2.1 The Server Side

The server side application includes data persistence and the logic necessary to constrain user actions. The second is the client side application. Using a RESTful API provided by the server, this will represent the list of possible resources and actions as a Single Page Application (SPA), making it easy for users to make game choices and see the game react.

2.2 Interoperability

A RESTful API was chosen to improve interoperability, making the game deployable on various client side devices. This project will focus on a web-application, to be rendered in a web browser. However, once in place, native applications or other client side implementations could be created to render the game on a host of various devices.

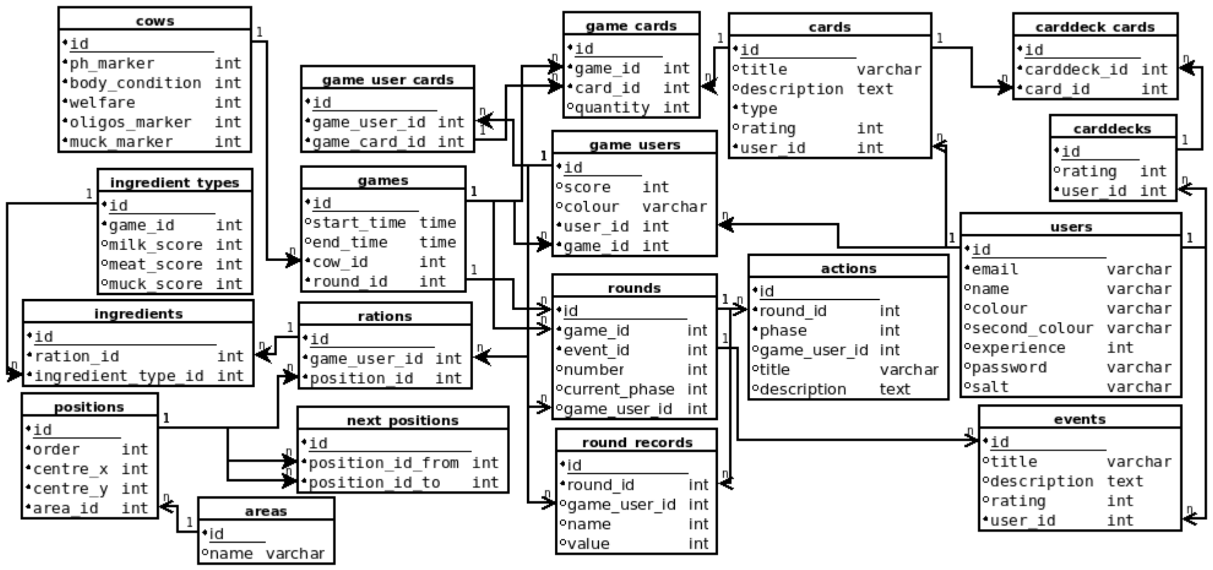
In accordance with a RESTful API, resources will be represented with unique URIs. HTTP actions will be used to access, update or delete the data. JSON will be the default language to send data between the server and client.

2.3 The Client Side

The client side application will use data from the server to represent a game board and controls to the user. When users are authenticated by the system their actions will update resources on the server that relate to their account and games they are involved in. So data will be retrieved from the server, and occasionally updated by the client.

3 Database Structure

3.1 Main Database Structure



3.1.1 Conventions

This design matches the Ruby On Rails Active Record convention. Tables names are plurals of the objects they contain. Primary keys are simply named 'id', and foreign keys are singular items [3].

3.1.2 Tables

users The **users** table stores details of players which allow them to login and store a few personal details. Their email, password and salt are to do with validation. Once logged in, players will be referred to by their name, and sometimes game pieces will be identified by a colour. The second colour is used in case two players in the same game have the same colour. A **users** experience will increase the more games they play.

game_users This is a table to join games with players in a many-to-many relationship. So it can be thought of as details that refer to a player and are specific to a game. A score and the determined colour (out of two possible) are needed per game the player is involved in.

games One record is kept in this table for each **game** that is played. A start time and end time is recorded, and a reference to information in the **cow** table and the current round.

cow The **cows** table shares a one-to-one relationship with the **games** table, only one cow is needed per game. Here values are recorded to persist the current value of markers which tell players what the state of health of the cow is in their game.

rounds Rounds must be recorded, as when the game is created, the number of rounds is determined, and an event chosen for each round. So rounds identify themselves to a game, they have a number which indicates the order they appear in. The current phase is a number from one to four, relating to the possible phases. The active player is also recorded, if applicable. When the round is in the last phase, no player is active. Rounds have an event assigned when they are created, at the start of a game.

actions These are created as the game progresses. They provide a record of what has happened in a round, used in the final review phase. The round and phase is necessary in an action, where as an action may be performed by a player, or may not. An example of an action is title: 'Made a ration', game_user: 'player1', and description: 'Spent a water and energy to make a ration'. They are designed to be human readable, rather than machine readable.

round_records This table exists in order to record values, such as player's scores, from round to round. The 'name' field can be 'score' or 'cards', and the value indicates the number. This table is used for review and game statistics only.

rations Each **game_user** can have up to four rations. These are capsules for ingredients that have a set position, which can be updated by the owner in the movement phase.

positions This table is necessary in order to tie the rations to the abstract shape of the game board. Positions are pre-set, and a single list is used for all games. Each position has an x and y position, which identifies where it will appear on the game board. It also has three images to represent it's shape. These are so that the area on the board can light up different colours depending if a player can move their ration to the position or not. Finally positions have an area and an order. This is so that the leading or rear ration can be determined. The leading ration is the one with the greatest area, and then the greatest order within that area.

next_positions This table defines a graph of positions. Positions can be considered nodes, and the next_position records can be considered to be one-directional connections. So having a position it is possible to look up all the positions that lead to it, and all the positions that lead from it. This will confine where a user is allowed to move their rations. In the rumen, bi-directional connections are allowed, so two records will be needed to join neighbours.

areas These correspond to the areas of the board: trough, oesophagus, rumen and intestines. They help order positions.

ingredients and ingredient_types A ration can have up to four ingredients. They are created by using cards with the corresponding ingredient, however, an ingredient record is only created when a ration is. Ingredient types store data about the number of points scored for each ingredient, as these are subject to change.

cards These are all possible cards, for all possible games. Each card includes a title (eg: Medicine), a description (eg: Cow welfare +1), a rating which can be given by people who finish playing a game with that card, and the creator of the card is automatically assigned.

Another diagram exists describing the possibility of cards performing automatic effects, defined by users when they are created. But that may be implemented in a later phase, and so has been left out of this diagram for simplicity.

card_decks and card_deck_cards These tables organise the cards into decks, creating a many-to-many relationship between the **cards** table and the **card_decks** table. When creating a game, a user can choose an existing card deck, or create a new one. To do this they choose the cards they want to include as well as the quantity of that card. The quantity defines how likely the card is to come from the deck, if card A has a quantity of 1, and card B a quantity of 4, card B is four times as likely as card 1, as with a real deck of cards. Card decks have a rating by those who have played a game using it.

game_cards As described above, a user chooses a deck of cards when creating a game. Because a deck could be edited by it's creator, these decks are copied to several other records, which define the unchangeable cards and quantities that will be used during the game.

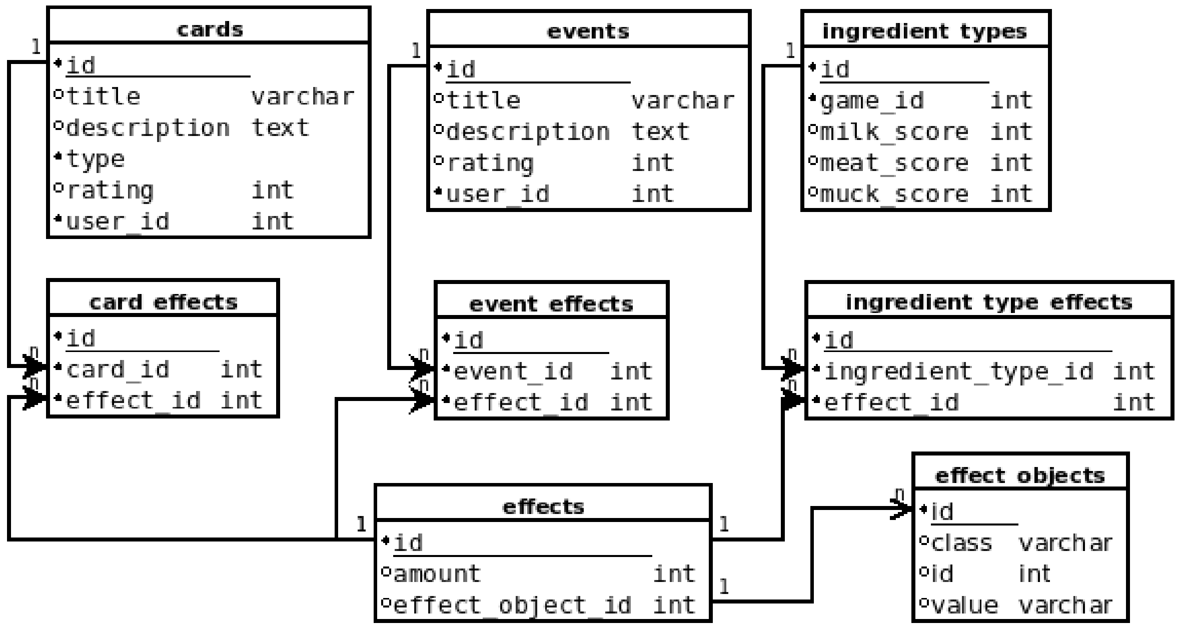
game_user_cards Within a game, a player accumulates a hand of cards. This table records this information by providing a many-to-many relationship between the **game_users** and **game_cards** tables.

events Possible events that can happen in a game are stored in this table. The event has a title, description, rating and image. All describe the event.

round_events When a game is created it's number of rounds is determined, and an event given to each round. This table joins the rounds to events.

3.2 Extra Database Structure

If the project reaches the final phase, it will be necessary for cards and events to be created with automatically assignable actions that get carried out during games. To make that possible, the following database structure must be added to the existing one above.



The **cards**, **events** and **ingredient_types** tables are from the existing database. They each have a sub table which links certain instances of cards, events and ingredient types to the **effects** table. An effect is something that happens to another object in the database. The class field indicates which table that object is in, the id field picks out the row to effect, and the value says what the new, or relative value of the row should be.

An effect engine will need to be put in place to carry out these operations, but given this structure it should not be complicated to perform. If other effects are needed later, such as creating or removing an instance, they can be added as other tables that are linked to the **effects** table.

4 RESTful API

This section describes how the data in the previous section will be accessed via the server API. A strict standard of a RESTful API has been chosen in order to increase the organisation of interoperability between the server and client. The notation '[id]' has been used to represent a number that can be used to represent a resource.

4.1 HTTP Operations

Unless stated otherwise below, the HTTP operations used for URIs will follow the pattern described in this section.

4.1.1 /resources

This will return a collection of all the resources with the name 'resources', if the user has permission to access them. The following HTTP operations are allowed for this URI:

- **POST** - creates a new 'resource'.
- **GET** - returns a list of all instances of 'resource'.

4.1.2 /resources/[id]

This will deal with a specific resource with the given id, if the user has permission to access it. The following HTTP operations are allowed for this URI:

- **GET** - returns a single resource's details.
- **PUT** - replaces a resource's details.
- **PATCH** - updates part of a resource's details.
- **DELETE** - removes a resource.

4.1.3 /resources/[id]/otherresources

This will return a collection of all the resources with the name 'otherresources' that are related to the specific 'resource' identified by the id. This is only allowed if the user has permission to access them. The following HTTP operations are allowed for this URI:

- **POST** - creates a new 'otherresource', but fills in the relation to the parent 'resource' automatically.
- **GET** - returns the 'otherresources' that are related to the 'resource', eg: are owned or created by it.

The same is true for further nested resources.

4.1.4 /resources/[id]/otherresources/[id]

This will deal with a specific resource with the given id, if the user has permission to access it. The following HTTP operations are allowed for this URI:

- **GET** - returns a single resource's details. Resources at this level are read-only. To update or delete the resource, it can be accessed via it's base URI **/otherresources/[id]**.

4.2 HTTP Data Types

The API is designed to provide a service for a client. It should be possible to play a game by giving the correct values to the API without a client. However, this is not the intended use, therefore the default data type of the application will be JSON format. This is all that will be implemented for the core functionality.

When the game is fully functional, there is a chance it will be used with other clients that may demand XML or HTML data. Ruby on Rails makes it easy to return different data types, based on the requested file type. These data types can therefore be offered at a later stage.

For addressability, as far as possible resources will provide links to related resources dynamically in the data that is sent to the client.

4.3 Resources

4.3.1 Users

- **/users** - a list of users.
- **/users/[id]** - a specific user.
- **/users/[id]/games** - games related to a specific user.
- **/users/[id]/games/[id]** - a specific game related to a specific user.

4.3.2 Games

- **/games** - a list of games.
- **/games/[id]** - a specific game.
- **/games/[id]/cow** - a cow that is related to this game. It is a singleton.
- **/cows** - a list of all cow details, should you wish to know it.
- **/cows/[id]** - a specific cow's details, without going through the game.
- **/games/[id]/cards** - cards used within a game.
- **/games/[id]/users** - players of a game.
- **/games/[id]/users/[id]** - a specific player of a game.
- **/games/[id]/users/[id]/cards** - a list of cards belonging to a player.
- **/games/[id]/users/[id]/cards/[id]** - a specific card belonging to a player.
- **/games/[id]/users/[id]/rations** - a list of rations belonging to a player.
- **/games/[id]/users/[id]/rations/[id]** - a specific ration belonging to a player.
- **/games/[id]/users/[id]/rations/[id]/ingredients** - a list of ingredients in a specific ration belonging to a player.
- **/games/[id]/users/[id]/rations/[id]/ingredients/[id]** - a specific ingredient in a ration belonging to a player.
- **/games/[id]/round** - the current round in a game.
- **/games/[id]/rounds** - the current and previous rounds in a game, unless the user has permission to see all the rounds in a game.

- **/games/[id]/rounds/[id]** - a specific round in a game.
- **/games/[id]/rounds/[id]/actions** - a list of the actions that have happened in a round.
- **/games/[id]/rounds/[id]/actions/[id]** - a specific action within a round in a game.
- **/games/[id]/rounds/[id]/event** - the event that happened in a round. It is a singleton.

4.3.3 Cards

- **/cards** - a list of all cards.
- **/cards/[id]** - a specific card.
- **/users/[id]/cards** - a list of cards the player has created.
- **/users/[id]/cards[id]** - a specific card that a player has created.
- **/decks** - a list of all decks.
- **/decks/[id]** - a specific deck of cards.
- **/decks/[id]/cards** - a list of all cards in a deck.
- **/decks/[id]/cards/[id]** - a specific card in a deck.
- **/users/[id]/decks** - a list of decks the player has created.
- **/users/[id]/decks[id]** - a specific deck that a player has created.

4.3.4 Rounds and Events

- **/rounds** - a list of all the rounds. Rounds are not created by the client, so are read only.
- **/rounds/[id]** - a specific round.
- **/actions** - a list of all the actions.
- **/action/[id]** - a specific action.
- **/rounds/[id]/actions** - a list of actions that happened within a round.
- **/events** - a list of events.
- **/events/[id]** - a specific event.
- **/rounds/[id]/event** - the event that happened in a round. It is a singleton, and is read only.
- **/users/[id]/events** - a list of events created by this user.
- **/users/[id]/events/[id]** - a specific event created by this user.

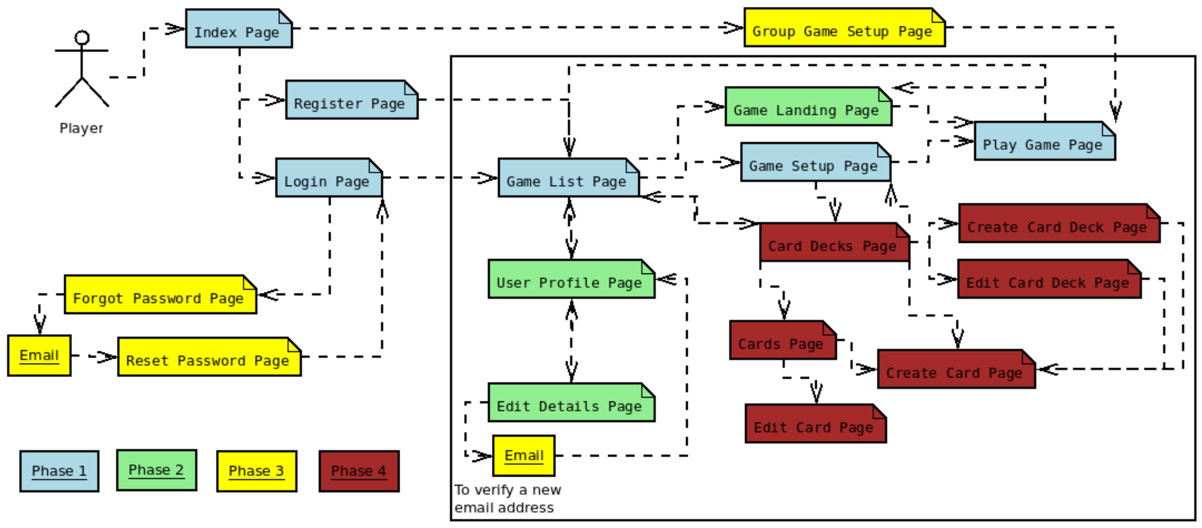
4.3.5 Positions and Areas

Positions and areas are read only. They are the same for all games. Rations are updated with positions.

- **/positions** - returns a list of all positions, in order.
- **/positions/[id]** - a single position's details.
- **/positions/[id]/next** - a list of all positions from this position.
- **/positions/[id]/previous** - a list of all positions going to this position.
- **/areas** - a list of all areas, in order.
- **/areas/[id]** - a single area's details.
- **/areas/[id]/positions** - a list of positions in this area, in order.
- **/positions/[id]/graph/[number]** - uses the positions like a graph. A number must be supplied. The result will be a recursive list of positions, each with an array of their next positions. The depth is the number supplied.

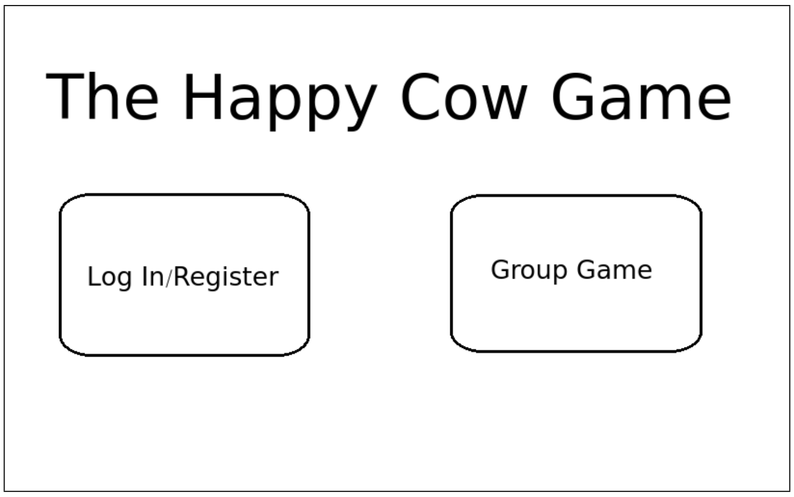
5 Client Structure

5.1 Pages



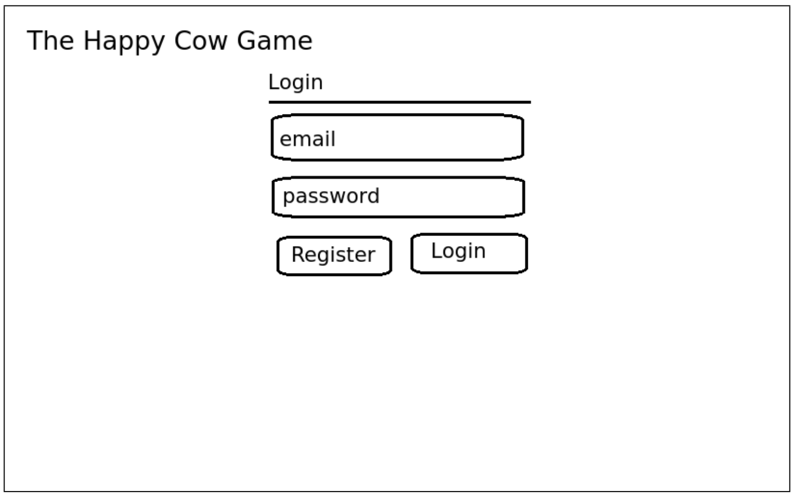
This diagram shows the pages needed for the application. Four phases are shown, indicating which pages are required for which phases of development outlined in the requirements analysis. The following description focuses on the first phase, the pages necessary for creating a working application.

5.1.1 Index Page



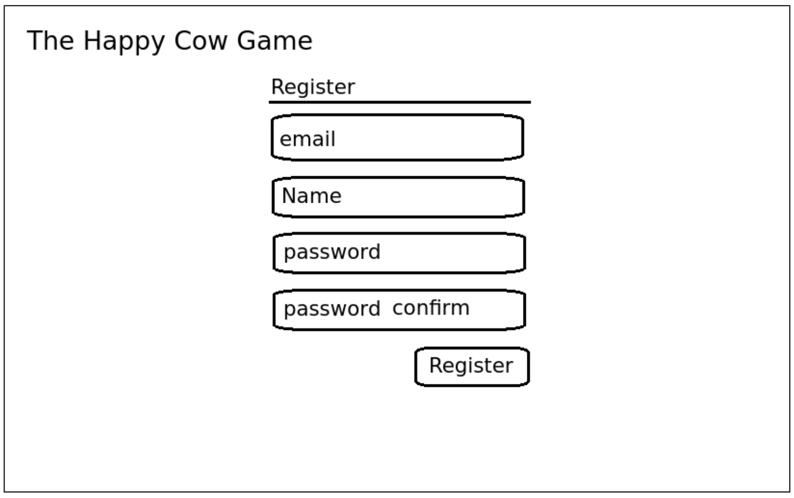
This is the index page, the first page a user sees. It gives the options of loging in and registering, or creating a group game. A group game will only be possible in a later phase of development.

5.1.2 Login Page



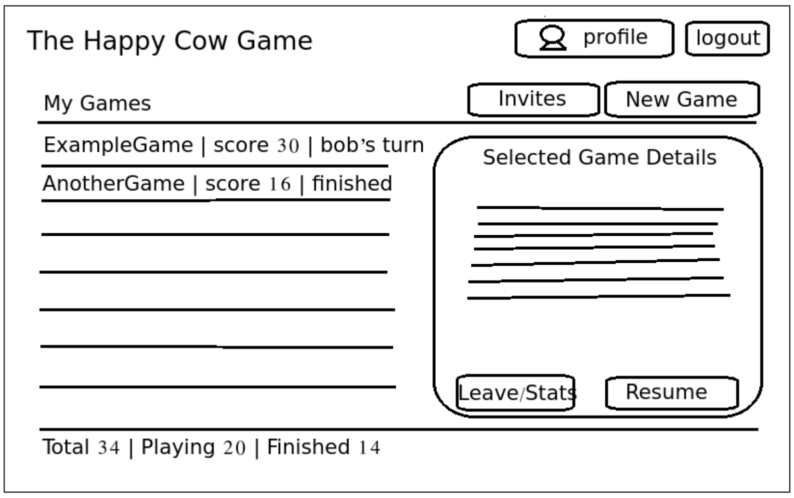
The login page allows a user to authenticate themselves with their email and password. There is also an option to register, if the user is new to the site.

5.1.3 Register Page



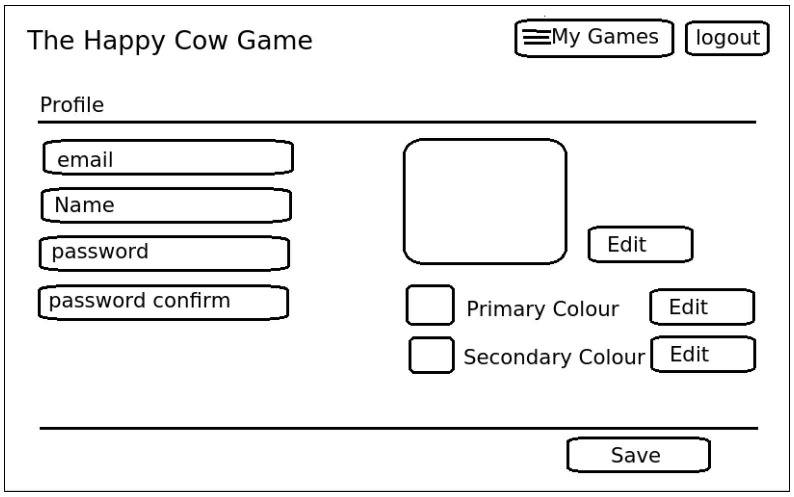
The register page is designed to look similar to the login page, but with a few more feilds. This means that when choosing to register instead of logging in, the page updates and simply asks for a name and confirmation of the password as well.

5.1.4 Games List Page


The Happy Cow Game
profile
logout
My Games
Invites
New Game
ExampleGame | score 30 | bob's turn
AnotherGame | score 16 | finished
Selected Game Details
Leave/Stats
Resume
Total 34 | Playing 20 | Finished 14

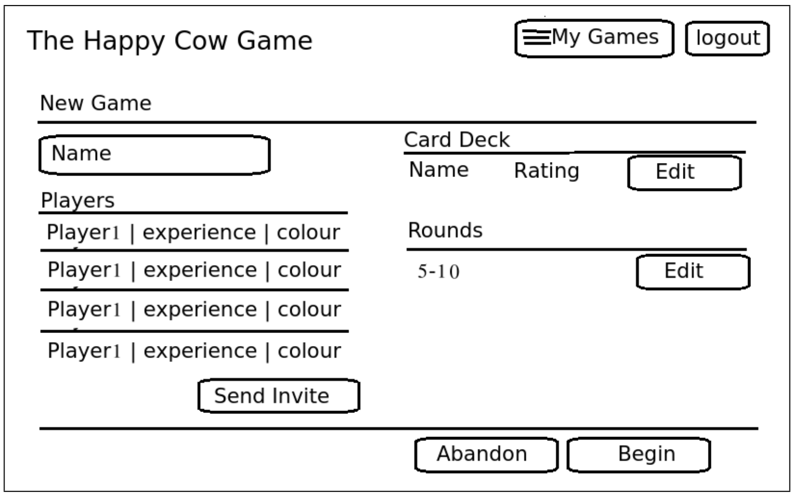
The game list page is the entry point for games, and other options available to a user. From here a user can see different games they are involved in, and a few details of each. Options allow a user to resume a game, or leave it if they don't want to play anymore. Other menu buttons allow a user to begin a game and view their profile.

5.1.5 Profile Page



This page allows a user to view and edit their personal details. The details needed to register are on the right. When updating a password or email, a confirmation email will be sent. The details on the right are to do with how users are represented in a game.

5.1.6 Game Setup Page


The Happy Cow Game
My Games
logout
New Game
Name
Card Deck
Name
Rating
Edit
Players
Player1 | experience | colour
Player1 | experience | colour
Player1 | experience | colour
Player1 | experience | colour
Rounds
5-10
Edit
Send Invite
Abandon
Begin

This page allows a user to create a new game. They can choose a name, a deck of cards to use, and the length of the game. They can also send invites to other players. At first these will be a link allowing a player to join a game themselves, but later if time allows, will be a message within the game.

For players who aren't creators but are joining a game, the edit buttons will not be available. The only action possible will be 'Join' or 'Leave'.

5.1.7 Play Game Page

This page is far more detailed than those above, and is described in the following section. When players resume a game, this page is loaded and guides them through the various changes and phases of a game.

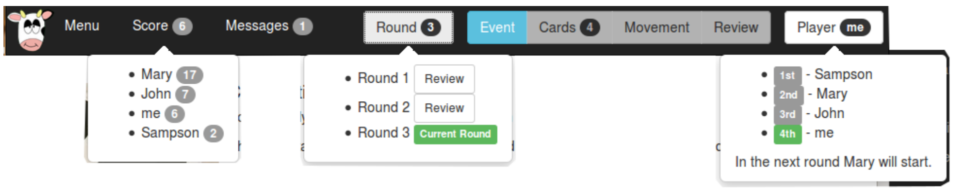
5.2 Game Play User Interface

The game play page is a single page application. That means it does not require a page load to update data on the page. Instead this is done behind the scenes via Ajax requests to the server. The methods to do this are discussed in the next section. Here the layout of this SPA is considered.

5.2.1 Menu

Basic controls of the game appear in a menu at the top of the screen. These include:

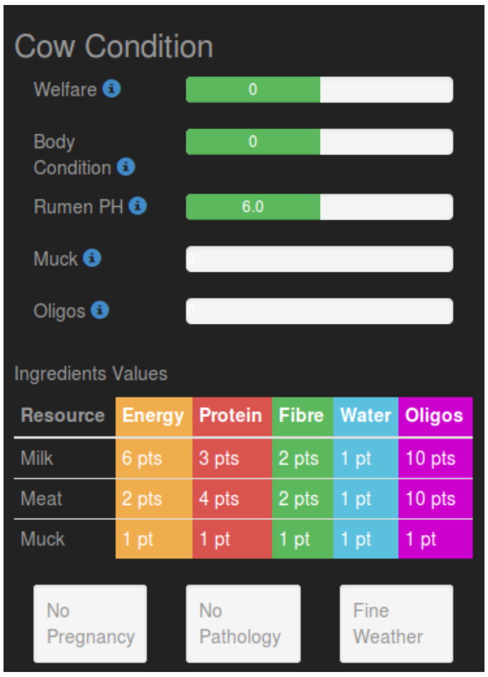
- A **Main Menu** with options to return to the game list page, or log out. More features can be added to this list if time allows, such as view statistics of the game.
- An overview of the **Score**. This shows the authenticated player's score, and when clicked, ranks the players in order of score.
- **Messages** will not be implemented in the core functionality.
- The **Round** button shows the current round, and also allows players to see the review phases of previous rounds.
- The **Phases** buttons show the current phase, but give players the ability to visit previous phases. The current phase may have a series of actions. Yet if the current phase is the cards phase, a player can still see the state of the board in the movement phase, but cannot perform any actions.
- The **Active Player** button shows the player whose turn it currently is. A drop down also shows the current order of play, as this changes from round to round.



In the next round Mary will start.

5.2.2 Cow Information

This section of the user interface, like the menu, is always visible. It shows the state of the cow.



Resource	Energy	Protein	Fibre	Water	Oligos
Milk	6 pts	3 pts	2 pts	1 pt	10 pts
Meat	2 pts	4 pts	2 pts	1 pt	10 pts
Muck	1 pt	1 pt	1 pt	1 pt	1 pt

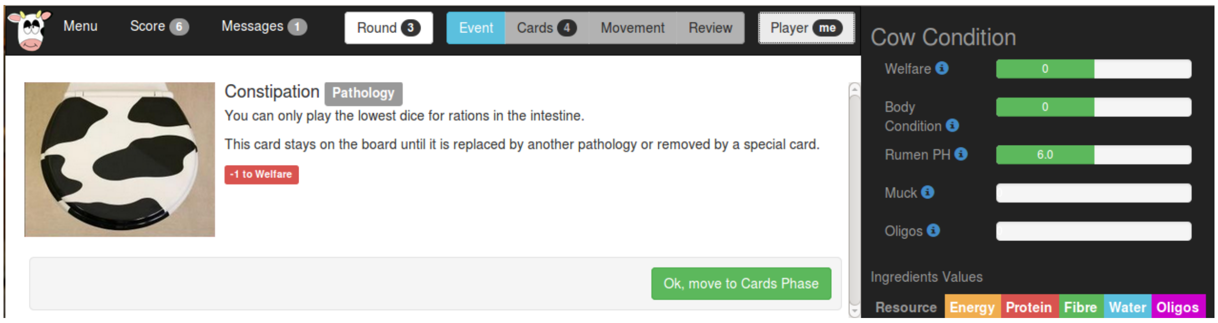
Five markers are shown: Welfare, Body Condition, Rumen PH, Muck and Oligos. These are updated as the game progresses and give an overview of various statistics.

The points that ingredients score are also shown, as these can change. This gives players an idea of what ingredients to include in their rations, and where to try and get their rations absorbed.

Finally there are a few events that remain from turn to turn. These are shown at the bottom and updated as events change.

5.2.3 Event Phase

The first of the four phases is the Event Phase. An event will usually have consequences for a player, but these happen later in the game play. The only action available to a player is to review the phase and move on to the cards phase. An example event is shown below.



Constipation Pathology

You can only play the lowest dice for rations in the intestine.

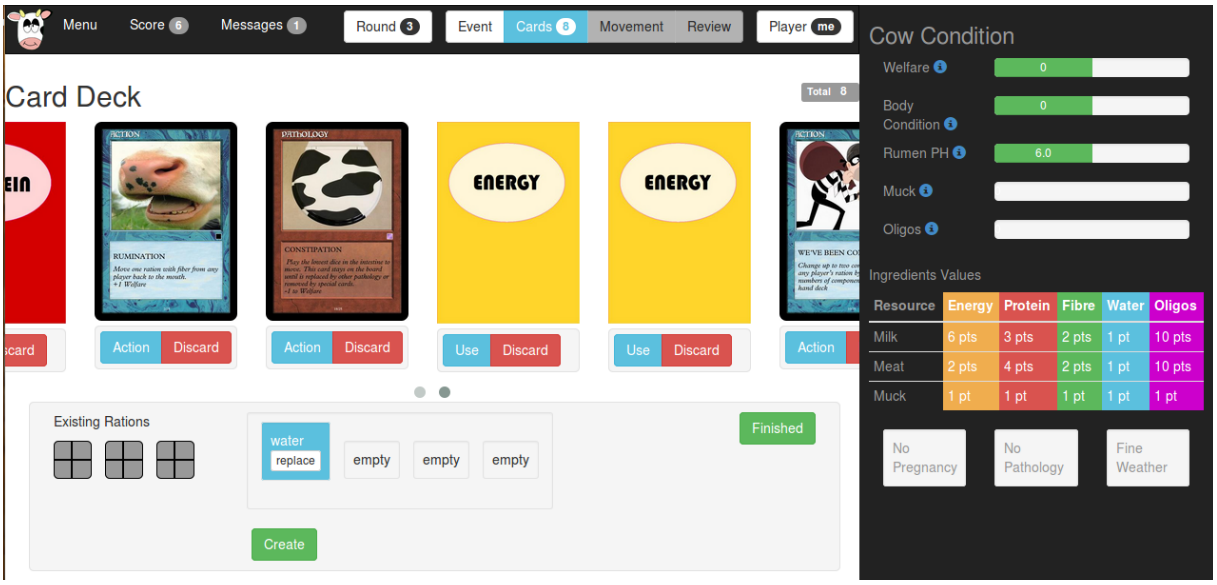
This card stays on the board until it is replaced by another pathology or removed by a special card.

-1 to Welfare

5.2.4 Cards Phase

The second phase is the cards phase. Players receive two new cards at the start of this phase, and can see their deck in a scrollable list of cards. Actions are shown below the cards, and for more details a player can click on a card.

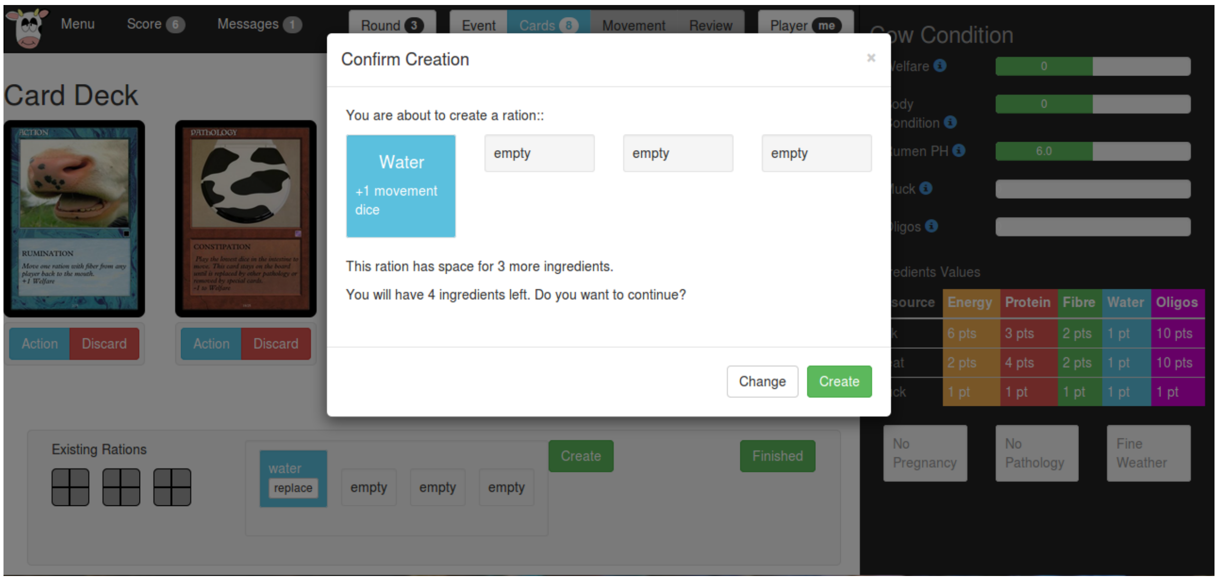
Below the list of cards a player can view their existing rations, and create a new ration if they have at least one ingredient and do not already have four rations. By clicking the 'Use' button below an ingredient, the ingredient appears in the list of ingredients ready to be made into a ration. When ready the player can click to create the ration.



Ingredients Values

Resource	Energy	Protein	Fibre	Water	Oligos
Milk	6 pts	3 pts	2 pts	1 pt	10 pts
Meat	2 pts	4 pts	2 pts	1 pt	10 pts
Muck	1 pt	1 pt	1 pt	1 pt	1 pt

Only one ration can be made per player, per round. A modal is displayed to confirm that the user wants to create a ration with the selected ingredients. If they are sure, they confirm their choice and the corresponding ingredient cards in their hand are removed.



Confirm Creation

You are about to create a ration::

Water +1 movement dice | empty | empty | empty

This ration has space for 3 more ingredients.

You will have 4 ingredients left. Do you want to continue?

Change | Create

redients Values

source	Energy	Protein	Fibre	Water	Oligos
k	6 pts	3 pts	2 pts	1 pt	10 pts
at	2 pts	4 pts	2 pts	1 pt	10 pts
ck	1 pt	1 pt	1 pt	1 pt	1 pt

The player can continue to perform actions on action cards, or discard cards. When they are ready to move on, they end the phase.

5.2.5 Movement Phase

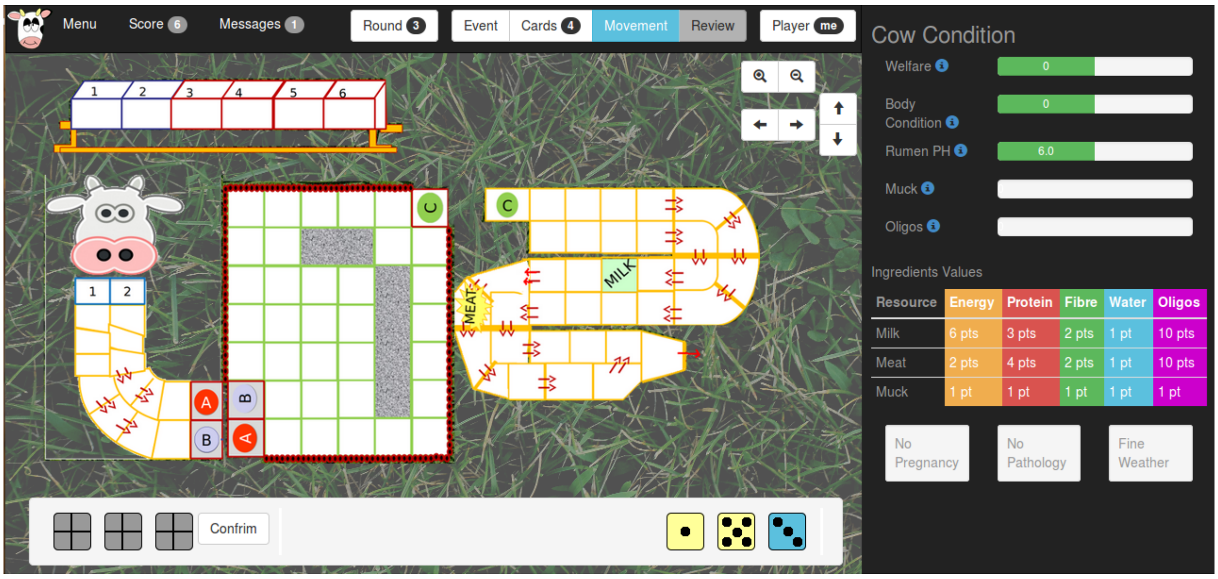
The third phase is the movement phase. In this phase the player must select a ration to move, and then select a movement amount offered by the dice. So there are 3 actions within this phase:

- Select a ration. The player can click on the rations in the bottom control bar, and the board will move so that the ration is shown in the centre. They then confirm their choice of ration to move.

- Select a movement amount. Only once a ration is selected are the movement amounts for that ration calculated and shown. The player can then select the different dice, and the squares where the ration can move are highlighted.

- Move the ration. By clicking on one of the highlighted squares, the player confirms their choice of movement. As a player clicks on a square, the ration is shown to 'hop' to that square. The player must keep clicking until all the movements are used up.

Once all the movements are used up for the selected ration, the player's movement phase is automatically over.

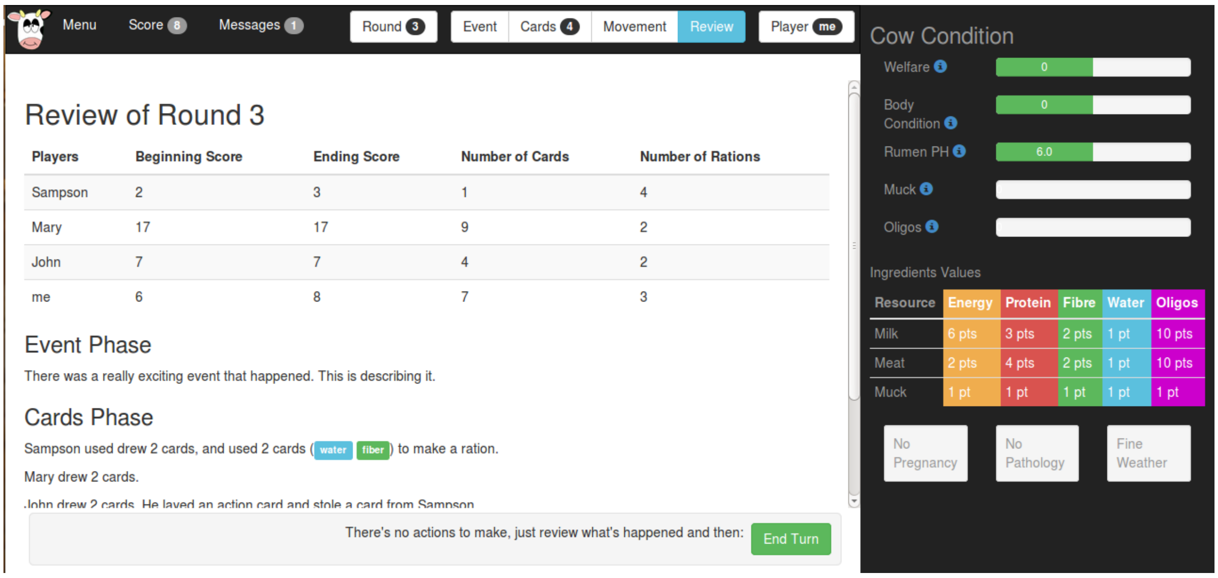


Ingredients Values

Resource	Energy	Protein	Fibre	Water	Oligos
Milk	6 pts	3 pts	2 pts	1 pt	10 pts
Meat	2 pts	4 pts	2 pts	1 pt	10 pts
Muck	1 pt	1 pt	1 pt	1 pt	1 pt

5.2.6 Review Phase

The fourth and final phase is the review phase. No actions are available, but to confirm that they are ready to move on. The player is shown a list of actions that have happened this round, and the current level of points for the round.



Review of Round 3

Players	Beginning Score	Ending Score	Number of Cards	Number of Rations
Sampson	2	3	1	4
Mary	17	17	9	2
John	7	7	4	2
me	6	8	7	3

Event Phase

There was a really exciting event that happened. This is describing it.

Cards Phase

Sampson used drew 2 cards, and used 2 cards (water fiber) to make a ration.

Mary drew 2 cards.

John drew 2 cards. He layed an action card and stole a card from Sampson.

There's no actions to make, just review what's happened and then:

Ingredients Values

Resource	Energy	Protein	Fibre	Water	Oligos
Milk	6 pts	3 pts	2 pts	1 pt	10 pts
Meat	2 pts	4 pts	2 pts	1 pt	10 pts
Muck	1 pt	1 pt	1 pt	1 pt	1 pt

When all the players indicate they are ready to move on, the next round begins.

5.3 AngularJS

This section delves into the different components that make up the game play page. AngularJS uses controllers and services to separate concerns. Controllers perform logic on data that is represented in the Document Object Model (DOM), and therefore each 'functional area' of a page (called the 'view' in Angular terms), should be operated by a separate controller [1], page 28. Services, on the other hand, can be created when needed to perform actions within the controllers. Examples are routing and data storage.

5.3.1 Controllers

The controllers needed to display and update data within the game play page are listed below.

1. **MenuController** This is a parent controller for the operations of the menu bar at the top of the page.
   (a) **MenuViewController** This controller manages a drop down menu with options to leave the game play area, logout, ect.
   (b) **ScoreViewController** This controller manages a drop down menu showing the players in order of score. It also displays the score and position of the player. Further information about scores throughout the game, and how to score points, can be accessed.
   (c) **RoundViewController** This controller manages a drop down menu showing the number of rounds played, and the current round. It offers the option to review previous rounds, allowing users to access the review information from previous rounds.
   (d) **PhaseViewController** This controller shows the current phase by highlighting the current phase and updating the phase on user clicks.
   (e) **Player View Controller** This controller shows the active player, and manages a drop down menu showing the player playing order.
2. **CowController** This is a parent controller for the cow information display, to the right of the page.
   (a) **CowStatsController** This controller manages the progress bars which show the welfare, body condition, rumen PH, muck and oligos markers. These need to be updated automatically as the game progresses.
   (b) **IngredientValueController** This controller manages the table of ingredient values, updating their values as events cause changes.
   (c) **CowEventsController** This controller manages the Pregnancy, Diseases and Weather areas that show changes to cow behaviour caused by events.
3. **PhaseController** This is a parent controller for the display of the main area of the board. It is changed as players view different phases using the PhaseViewController.
   (a) **EventController** This controller manages the template showing the event of the round.
   (b) **CardsController** This controller is a parent of those which manage the actions related to cards.
      i. **CardDeckController** This controller manages actions of cards, and also displays modals with more details of the cards and their causes.
      ii. **CreateRationController** This controller manages uses of ingredients and displays rations as they are created. A modal is provided to review a ration to confirm creation.
      iii. **ExistingRationsController** This controller shows the existing rations, and manages pop-ups showing their position. If a ration is created, it updates to show it.
   (c) **MovementController** This controller is a parent of those which control movement of rations.
      i. **BoardController** This controller manages the zooming, scrolling, and automatic changes of the board as rations are selected.
      ii. **RationMovementController** Once a ration is selected, this controller manages the actions needed to highlight it, show it's possible movements and update it's position.

iii. **RationSelectionController** This controller manages the selection of controllers and dice to decide it's movement. Rations and dice need to be disabled once one is chosen.

(d) **ReviewController** This controller manages the template showing the actions that have happened during the round.

4. **AlertsController** This controller manages the alerts created by user actions, displaying and deleting them.

5.3.2 Services

Currently known services needed are:

- An **$resource** service. This extends the $http service and is especially designed for use with a RESTful API. It will perform requests to the server. Many of the controllers will need to do this to perform actions players have requested, and update the page with the responses.

- An **$animate** service. This helps angular perform basic animations on elements within the page.

- An **Alerts** service. This will allow any controller to create alerts. The AlertsController will be integrated to display alerts that are generated by this service.

- A **LocalStorageModule** service (`https://github.com/grevory/angular-local-storage`). This will be used during development to store data instead of sending it to the server. Once the prototype is complete, this will be replaced by the **$http** service.

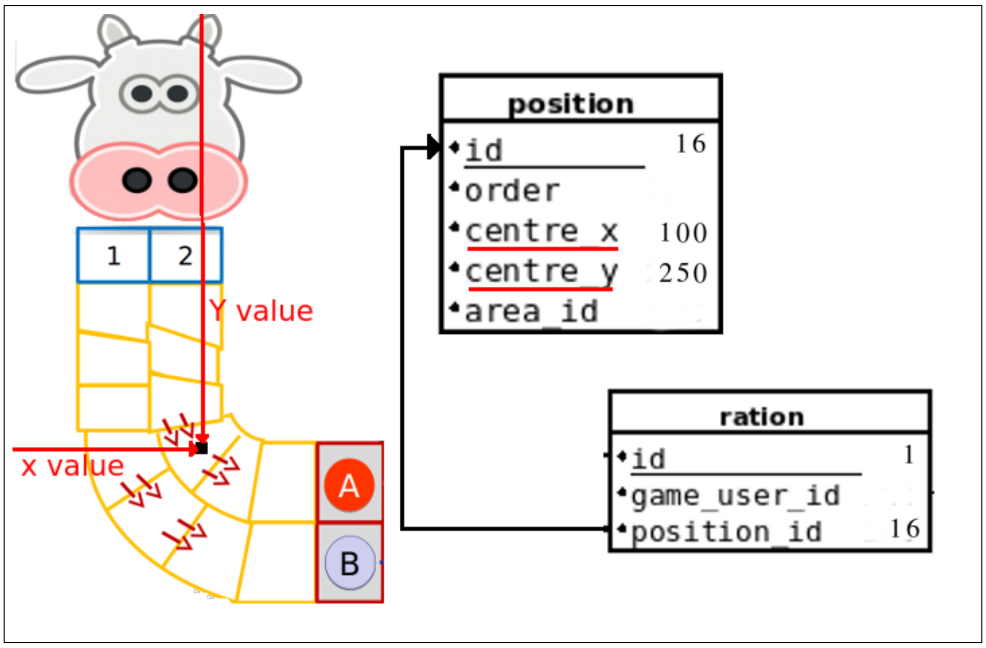
6 Detailed Design Sections

Some areas of the game will be straight forward, and simply involve implementing the API and client side structure above. Other sections require some more thought, however. The aim of this section is to identify and briefly consider these sections.

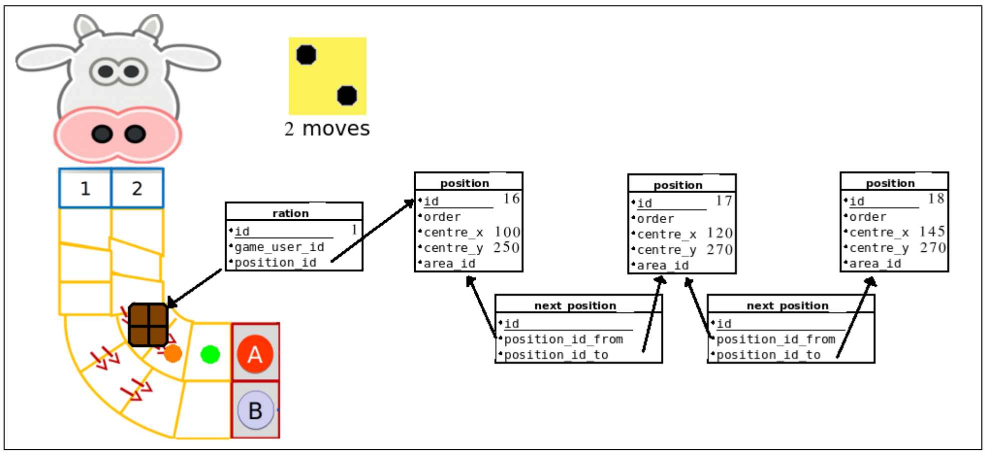
6.1 Drawing and Moving Rations

Once created, a ration has a place on the game board. This has to be both persisted in a database, and represented to the user. To do this, a table of positions is stored, the same list exists for every game instance. Positions need to know where they should appear on the game board. For this reason they have a centre position, indicated by an x and y value. These values correspond to a position on the game board, measured from the top left.

A ration has a record of it's position. It always has a position. If it is absorbed or leaves the cow, it scores points and is then removed. A ration can therefore, use it's position's coordinates to decide where it appears on the board.



As well as x and y coordinates, positions also have a concept of 'next positions'. This means that a position knows which positions a ration can move to from it, and vice versa. This creates a data structure called a graph, where nodes (positions) are connected by one-way relations. When moving a ration, it is therefore possible to look up which positions can be moved to from the current position. This is possible using the server API **/positions/[id]/graph/[number]** where [id] is the current position, and [number] is the value of the dice.

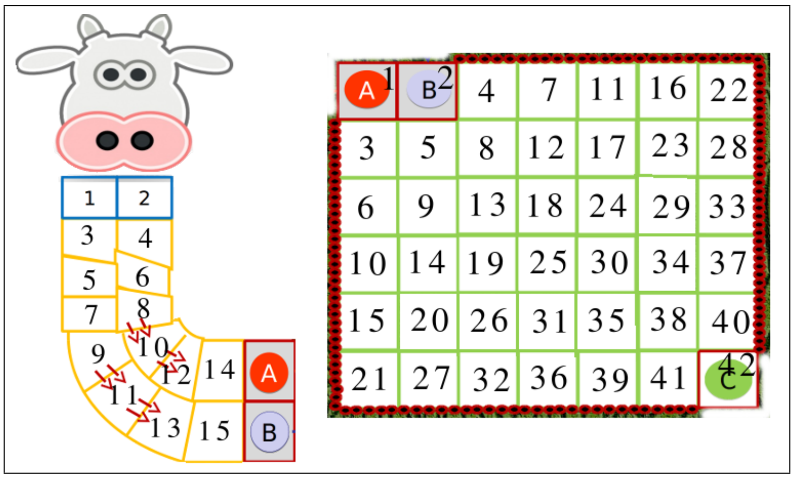


In this example, the ration can move 2 spaces. Because the ration is on a bend, the only relations between positions are forwards, down the oesophagus. The ration must move through the first position, so it is marked in orange, but can only finish in the second position, so it is marked in green. These colours help players to see where their rations are allowed to go as they select different dice, and think about which movement to use.

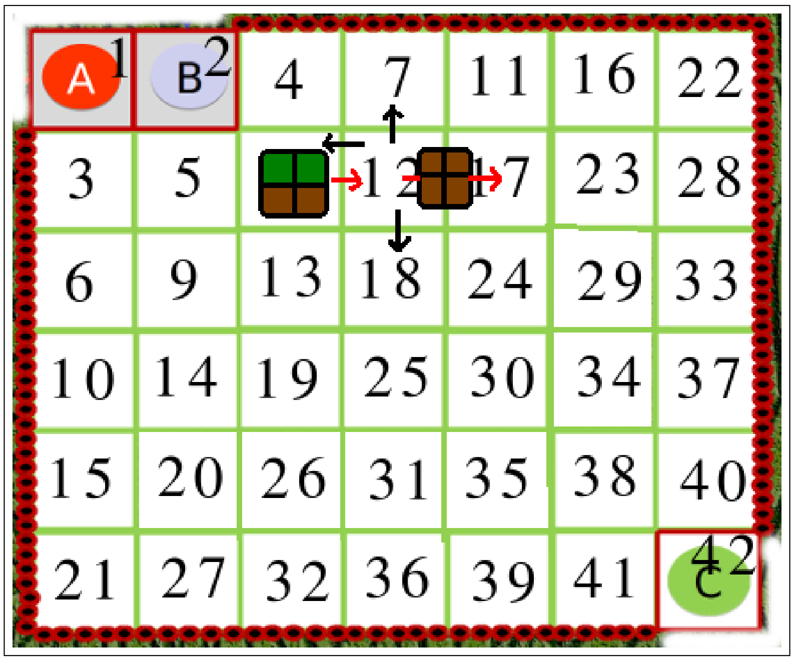
The final challenge is animating the movement. A ration moves one position at a time, as the player must choose which position to move to if there is more than one option. The animation can be done using the coordinates of the two positions, the current position of the ration and the new position of the ration. The difference of the coordinates for x and y is determined, and then divided by 10. The ration is then moved by this amount every 50 mili-seconds. After half a second the animation will be complete.

6.2 Ordering Positions

To be able to order rations based on their progress through the cow digestive system, positions need an order. When rations are displayed in the cards or movement phase, they can then be ordered by the order of their positions, without having to work out an order using the graph of positions, which would be computationally expensive. In the oesophagus and intestines the order is fairly straight forward, and can be determined based on the distance from the entry point. In the rumen, the order will be across, so the numbers increase as they go down and to the right.



As well as providing an order of rations that can be used for display, and resolving some actions, an order to positions as an added advantage when it comes to pushing rations. Rations can be pushed by other rations with more fiber. To work out the direction the attacked ration should be pushed in, the order of positions can be used. If the order of the position the pushing ration came in is less than the order of the current position, then the attacked ration must be pushed to a position with a larger order.



In the example above a ration in position ordered 8 is moving to position ordered 12 and has fiber, so can push a ration already on that position. But which direction should the ration be pushed? It is obvious to us, but not with a graph data structure. Instead of comparing coordinates, which could become very complicated in the oesophagus or intestines, the order can be used. There are four possible positions: 7, 8, 17 and 18. The pushing ration is from the second lowest position, so it pushes to the second highest position. If it came from the lowest position it would push to the highest position, and vice versa.

6.3 Invites

When a player creates a game, he will want other players to join in. This is done through invitations. One player has control over the creation of a game in order to avoid conflicts when setting a game up.

Initially, to invite another player to a game, the creator of the game will have to send an invitation as a URL to a page via some form of social media. That will allow the player they contact to login, and visit the same game creation page. The player joining will see the name and choice of cards for the game, as well as the list of players who have already joined the game, but they will only be able to perform two actions: join or leave. If they join, a **game_users** record is created for that player. If they then decide to leave the game, that record is deleted.

Messages are an extra feature of the project, to be implemented in a later phase. An invitation will be a special form of message that appears in a list of invites for each user. It is essentially an in-game message of the type above that directs a player to the game setup page. They then join or leave as described above.

Annotated Bibliography

[1] B. Green and S. Seshadri, *AngularJS*, third edition ed., M. B. S. St. Laurent, Ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media Inc., 2013.

A book introducing AngularJS. It includes helpful guides and tips for getting started and explaining why AngularJS works the way it does.

[2] C. Price and N. Hardy, "Software Engineering Group Projects Design Specification Standards," November 2010, unpublished. [Online]. Available: http://www.aber.ac.uk/~dcswww/Dept/Teaching/CourseNotes/2010-2011/CS22120/GPDocs/se.qa.05a.pdf

This was the document used to describe the contents of a design specification for the group project in year two of the course. It contains descriptions of several sections. It is a good aid for where to get started with design specification.

[3] Various, "Active record basics," accessed: February 2015. [Online]. Available: http://guides.rubyonrails.org/active-record-basics.html

An introduction to Ruby on Rails Active Record. It covers the convention of naming used in database tables.