Uno!

David Ferm
Anders Olofsson

Webbapplikationer DAT076 final project report

https://github.com/david-ferm00/WebbapplikationerDAT076

Introduction

The aim of this project is to recreate the well-known card game UNO, in a web-application format. This means that we aim to connect two players on a server and allow them to play the game together. A potential feature which we have not been able to implement would be to allow more than two players in one game, which we have also kept in mind while developing this application. The game of UNO is a simple card game with a deck that is unique to the game. Most of the cards have a colour and a value, apart from special cards which have destructive effects on the other player. The aim of the game is to get rid of one's hand, while other player's are able to force the player to pick up if they play such a card. The main catch with UNO is that once a player reaches only one card left, they need to say "uno". The obvious problem is that if you say uno it does not do anything in the application. Because of this, we have implemented a button which is to be used instead of saying "uno".

https://github.com/david-ferm00/WebbapplikationerDAT076

This project has been written using typescript for the back-end and Reactjs with the help of bootstrap for the front-end. In order to connect the two, we use axios to make requests to the server from the front-end.

List of Use cases

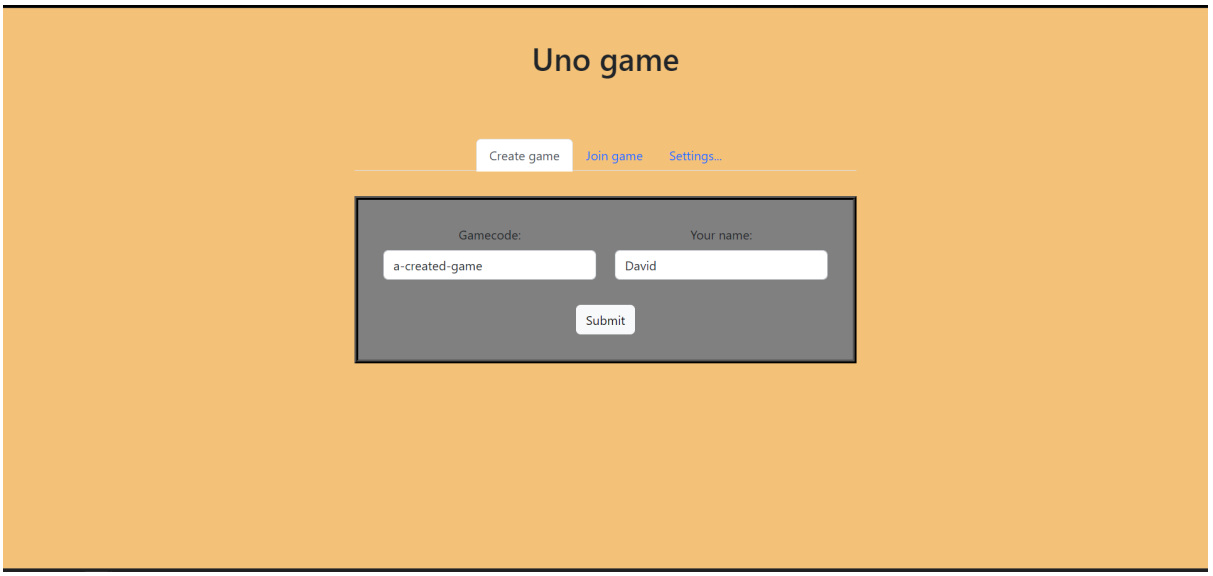
Seeing as this project is a game, it is difficult to assign use-cases in the same way one would for a more "practical application". However the list of use cases we made before production of the game can be found below.

1. The user should be able to create a game
2. The user should be able to see already existing games
3. The user should be able to join an already existing game
4. The user should be able to choose a card to place, and place it
5. The user should be able to pick up a card if the game rules requires it
6. The user should be able to call UNO
7. The user should pick up two cards if UNO is not called within a certain timeframe
8. The user should be able to finish a game (win/lose)
9. The user should be able to force the opponent to pick up a card when such a special card is placed
10. The user should be able to choose what the new colour of play is, when such a card is placed.
11. The user should be able to see which cards are in their hand
12. The user should be able to see how many cards remain in the other players hand

User Manual

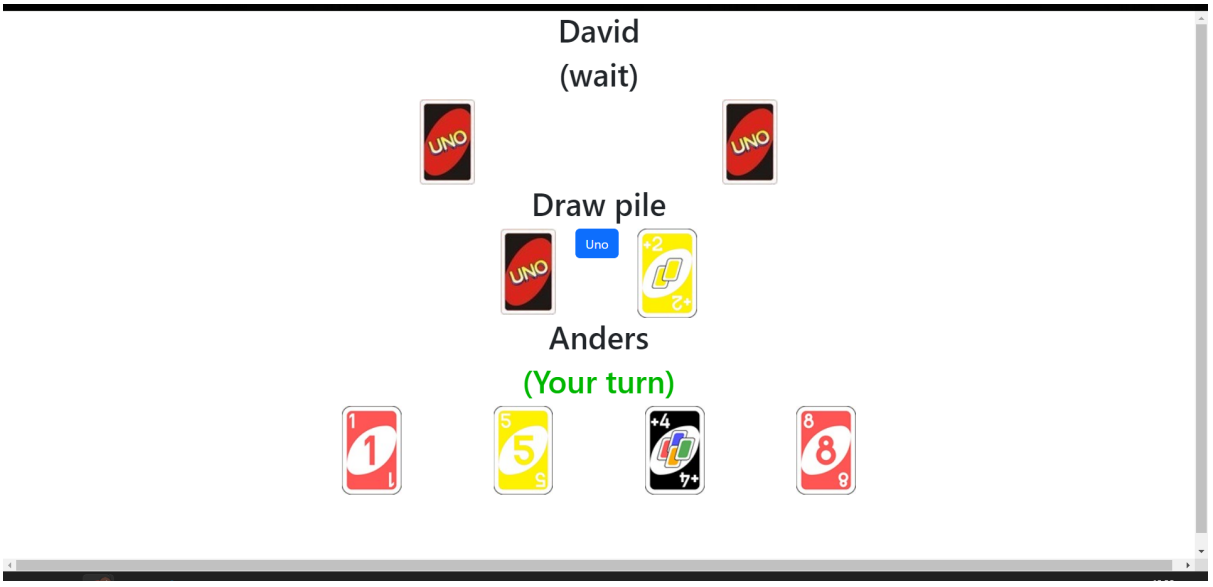
In order to install the application, you will need to clone the github repository. There are some node packages that need to be installed before you can run the project. Axios, cors and react-bootstrap must be installed for the front-end, as well as express for the back-end. Additionally, since we use react-router-dom in order to navigate between pages, this package needs to be installed as well. After everything has been installed and set-up, you need to start the client and the server separately. To start the server, navigate to the server folder in the project via terminal and enter: “npm run-dev”. To start the client, navigate to the client folder in the project via terminal and enter: “npm start”. In order to connect another “user”, you can connect to localhost:3000 in your browser. We have unfortunately not made the application able to connect over multiple devices yet.

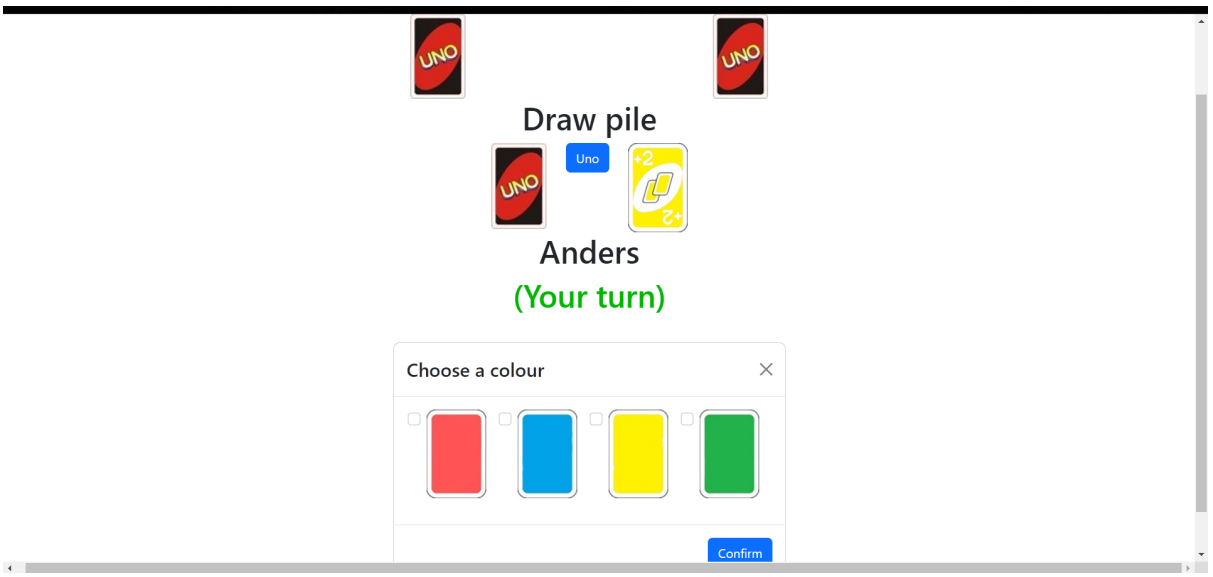
The application should be quite intuitive to use. At the start screen there are three tabs labelled: create game, join game and settings (as seen below). If there are no games to join, the user can create a game by inputting a name and a gamecode. After this is done the user is immediately sent to the game page of the game they have just created. If there is a game to join, it will be displayed in the join Game tab, where you can see the game code and the number of players currently in the game. The user is only able to join the game if they have filled in a name in the name input textbox. The reason for this is so that the server is able to tell the difference between the two players and give each user the correct information. Additionally, as the game only supports two players, if it shows there are two players in the game, the user is unable to join the game. At the moment the settings tab does not contain anything of value.

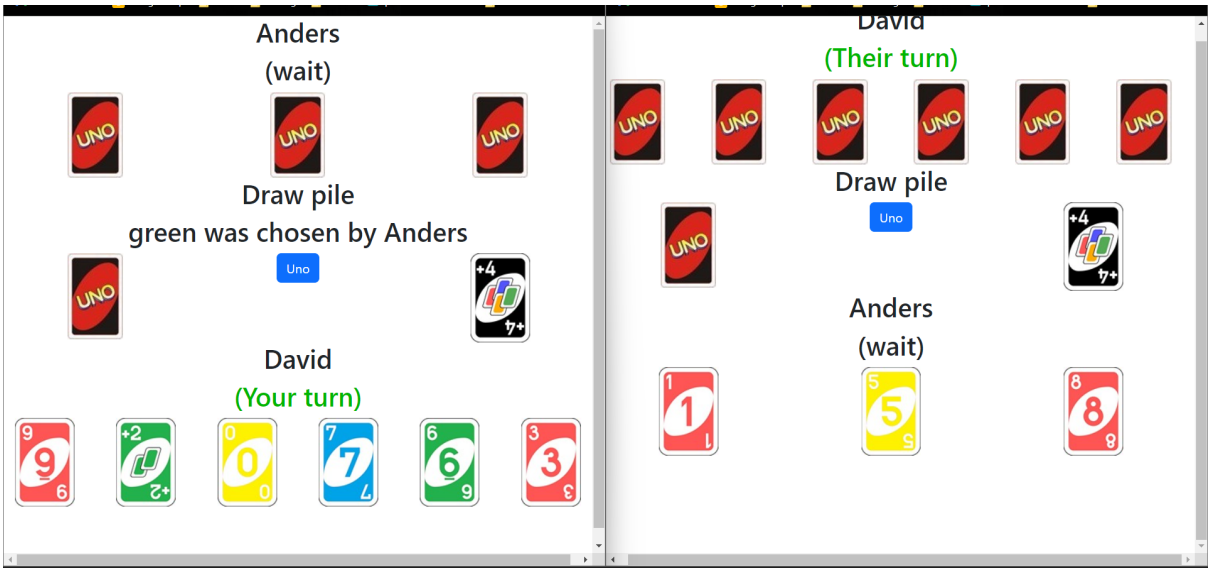


In the game page the user will see three rows of cards. The top row represents the opponent's hand, where they cannot see the actual values of the card, but how many the user has left. The middle row contains the draw and discard pile, the draw pile being where the user clicks in order to get cards, and the discard pile being where the user can place cards. In order to place a card the user simply clicks on a card in their hand, if the user is able to place that card, the card is placed in the discard pile. If the user does not have a card which they can place, they are able to pick up cards from the draw pile by clicking on it. The middle row also has the UNO button, so if the user presses it at the wrong time, they gain cards. The third and final row contains the user's hand. This will display the value and colour of the cards, so that the user can pick between them.

There are certain cards which allow the user to change the colour of play. When a card is placed that allows the user to change the colour of play, they need a way to do that. The thought was to have this component appear when such a card is placed and then the user can pick a colour and play would continue. In the screenshot below, there is such a card which allows colour changes, on the far left in the hand. When a colour is chosen, a text is displayed to the other player informing the user of which colour was chosen.







There are also cards which force the other player to pick up cards. These come in the form of coloured +2 cards, and a +4 which also allows the player to choose the colour of play. When these are placed, the server automatically takes care of that logic and adds cards to the opponent's hand. This way the user does not need to think about that at all.

Design

This application is made up of a front-end(what the user sees / what is managed on the client side) and a back-end(the logic in the server). On the client side, react and bootstrap is used to create the visuals that the user sees. We also used the react router package in order to link our two pages together. We have specific files which represent each page (MainPage and UnoGame), as well as files which contain different elements that these pages use. We also make use of css files in order to stylize the pages. The client has files which represent the model (such as Card and Pile classes), as well as image files so that we can show the cards.

In the front-end the application is built so that it simply shows the current game state. We use an interval to request the current game state from the server every 200ms. The relevant information from the game state is then sent to the rest of the components on the screen. Many of the components have onClick listeners implemented, which then in turn sends requests to the server. The gamestate is represented by a class in both the frontend and the backend, and contains only the necessary information that the front-end needs to display. The rest of the game information is stored in the back-end.

In the back-end, we have the majority of the game's logic. The back-end is also where the game's information is stored. The communication between frontend and backend occurs through a router file. This is where all requests are defined and handled. The router file contains which requests can be made and then tells the game controller what to do. For example, if the user wants to place a card, a request containing the card, the id of the player and the gamecode is sent to the server. The router file then handles this request and tells the service layer to take that card from that player's hand in that game, and place it in the discard pile.

In the service layer we have two main classes, one is the GameManager, and the other is the Game class. When a request is handled by the router it calls a function in the game manager, which then picks which game to update and calls the necessary function inside that instance of the Game class. We also have a small class which is used to represent the information used in the game list for the mainpage. This small class only contains the game code and the number of players. The reason why we have this class is because it means that we avoid sending unnecessary information to the client. One advantage of using a Game manager is that we are able to have multiple games ongoing at the same time. These instances of games are stored simply in an array. We have chosen to use an array as we do not expect a large number of games ongoing at the same time. The initial idea was to store the games in a database, however due to there being so few games, we decided it was unnecessary.

There are a number of requests detailed in the backend:

For matchmaking we have: gamelist, creategame and joingame.

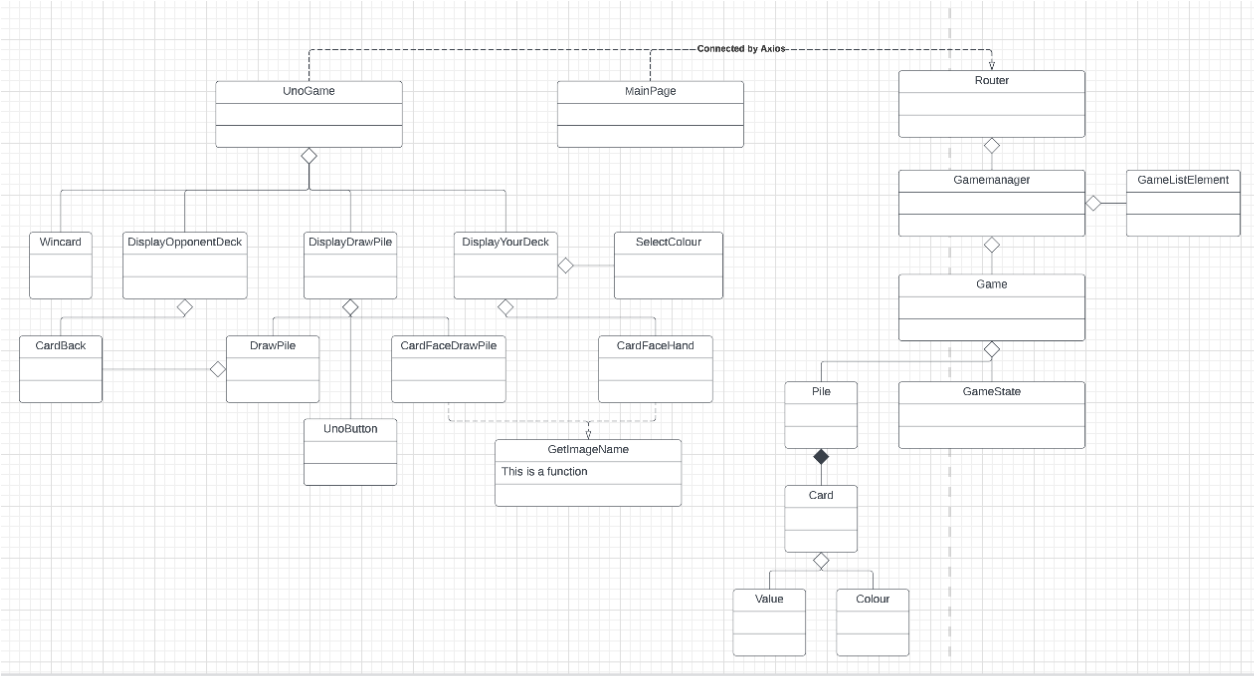
- Gamelist returns a list of currently active games. Seeing as it is a get request without any parameters, there are no invalid requests. If there are no active games, it will return an empty list.
- Creategame does exactly as expected, it creates an instance of the uno game. The parameters that the request takes are two strings: the game code and the name of the player. If either of the parameters are not of type string, an error code of 400 will be sent back. The response to this request is simply a code to signify if it worked or not. For example, if a user tries to create a game with a code that already exists, it will send error code 400.
- Joingame is very similar to creategame in respects to the parameters and the response. The way the request is handled however is slightly different. This request sets the id of player2 to whatever name is given in the parameters for the game specified through the code. If the code given does not correspond to a game, an

error is thrown and response code 500 is sent back. A similar story for when the second player tries to name themselves the same as the first player.

For the actual game logic we have: game_state, pickUpCard, select_card and say_uno.

- Game_state is a request which allows the front-end to keep up with what the gamestate actually is. The parameter for this request is the id of the player as well as the gamecode. Once again, we test the types of the parameters, and if they are not what was expected, we return an error code. If the request is valid, the server sends the gamestate back to the client, containing only information that that player should know. In the event that the gamecode does not exist we throw an error and return an error code.
- pickUpCard takes the player id which requested to take a card from the draw deck and then adds a card to the player's hand. It is similar to the request above in how the request may be invalid. The only response to this request is an OK code if the player ends up picking up a card, and an error code if not.
- The select_card request is made when a user wants to place a card onto the discard pile. There are three parameters to this request: the card to be placed, the player id and the game code. If the card is not of the type card, meaning there is no value or colour, the request is invalid and an error code will be sent as a response. Regarding the other parameters it is a similar story as above, if they do not correspond to a certain game, then an error code will be sent. If the request is valid, the game will try to place the card onto the discard pile. If the card is not of the same colour or value as the top card of the discard pile, it will simply not place the card and send an error code back.
- Say_uno corresponds to the main component of the uno game. On the client side there is a button which can be pressed at any time. It should only be pressed when the player has reached one card in their hand. If the player has not reached one card, the game will give the player two extra penalty cards. Parameters are handled similarly to the above requests. If for any reason the request fails, an exception will be thrown and an error code will be sent.

Below is a diagram showing how the different classes and components are connected to each other. On the right hand side is the server side, and on the left is the client side. The client side is missing the model (Card, Pile classes etc.) because they are used by many different components.



Responsibilities

Seeing as we only have two people in our group for the project, it is quite difficult to place responsibility for specific features or classes on one person. This is because we have worked together on the majority of classes and there is no one area where only one person worked. If we were forced to divide the responsibilities up, we would divide them so that the person who did the majority of the work there is listed.

David:

- Model in server
- Matchmaking requests
- Game manager
- Game class
- Report
- Layout mainpage
- Logic mainpage
- Linking between pages
- Connecting front and back end
- Uno button + logic

Anders:

- Model in client
- Uno game requests
- Testing
- Layout game page
- Images
- Colour changing
- Components in uno game
- Refactoring code (making the code better even when functionality is there)

Css
Gamestate logic