Technical Report - FarmClash

Camille De Vuyst, Joren Van der Sande, Thomas De Volder,
Faisal Ettarrahi, Ferhat Van Herck, Siebe Mees

April 22, 2024

Contents

Abstract

This technical report describes the development and implementation of FarmClash, a web-based idle game designed for the Programming Project Databases course at the University of Antwerp. The game integrates concepts of resource management and multiplayer interaction within a farm management context, drawing inspiration from popular titles like Clash of Clans and Hay Day.

1 Introduction

FarmClash is an educational project designed to apply database and web development skills in a practical scenario. Inspired by games like Clash of Clans and Hay Day, it challenges players to manage a farm, interact with other players, and strategize in real-time, balancing competitive and cooperative gameplay elements.

2 Project Objectives

The primary objectives of this project include:

- Develop a functional web-based idle game, inspired by titles such as Grepolis and Clash of Clans, focusing on the management of cities and resources.
- Utilize PostgreSQL to manage a robust database that handles game states, user data, and interactive features such as guilds and direct messaging.
- Implement secure and efficient server-side interactions using Flask, ensuring that the system handles real-time data exchange and multiplayer interactions seamlessly.
- Create an intuitive user interface that facilitates easy navigation and management of game tasks, utilizing technologies such as React for dynamic frontend development and CSS frameworks for aesthetic design.
- Design the game to support background updates and cooldowns effectively, representing idle game mechanics where actions continue to progress even when players are not actively engaged.
- TODO: Ensure that the application adheres to modern web standards and security measures, including the use of JSON Web Tokens for stateless authentication and secure communication.

- paragraphasize software extensibility and maintainability, preparing the system for future upgrades and scalability.
- Deploy and maintain the software system on a production environment, leveraging provided Google Cloud Platform resources for hosting.

This project is designed to meet the educational goals of learning to develop extensive software projects within a team setting, discovering and integrating new technologies, and effectively planning and distributing tasks.

3 Game Description

FarmClash allows players to build and manage their farms, defend against attacks from other players, and maximize their profits through strategic selling of farm produce influenced by dynamic market prices. Players start with a basic farm and can upgrade their facilities and defenses as they progress, interacting with other players through alliances and competitive gameplay.

4 Technologies Used

- **Flask:** Serves as the web server framework.
- **PostgreSQL:** Manages all data storage needs.
- **Redis:** Implemented for caching purposes to enhance performance (pending verification).
- **JavaScript:** Used for dynamic frontend development.
- **Tools:** Includes PyCharm, DataGrip, JIRA, and Google Cloud among others.

5 Implementation Details

5.1 Database Schema

The database schema is designed to efficiently handle game data and user interactions. It includes tables for users, game states, transactions, and market prices, reflecting the dynamic nature of the game's economy.
∗ TODO: Add more details here. e.g. image of the final db schama

5.2 Backend Logic

The backend handles requests from the client, processes game logic, and interacts with the database. It ensures that all player actions are reflected in real-time and manages the continuous accumulation of resources even when the player is offline.

5.2.1 Login and Authentication Feature

Login Feature The login functionality is critical for user authentication, allowing users to securely access their accounts.

- **Endpoint** The route '/auth/login'(methods=['GET', 'POST']) handles the login requests.
- **Form Processing** When a POST request is received, the server retrieves the username and password from the form data.
- **User Validation** The user details are fetched from the database using user_data_access.get_user(username). The password is verified using werkzeug_check_password_
- **Session Management** If the credentials are valid, login_user(user_record) is called to log the user in, using Flask-Login to handle the session.
- **Response** After successful login, users are redirected to the game interface or admin panel based on their role
- **Error Handling** If login fails, an error message is displayed.

Registration Feature Registration allows new users to create accounts by providing their username, password, and email.

- **Endpoint** The route '/auth/register'(methods=['GET', 'POST']) handles the registration requests.
- **User Creation** On receiving a POST request, it attempts to create a new user record with the provided details. The password should ideally be hashed before storage (though hashing is not explicitly mentioned in the provided code).
- **Account initialization** If the user is successfully added to the database, the system initializes the user's game environment, including creating a default map and initializing resources using GameServices.
- **Response** Successful registration redirects the user to the login page. If registration fails (e.g., if the username is already taken), an error message is provided.

Logout Feature The logout functionality securely ends a user's session.

- **Endpoint** The route '/auth/logout' logs the user out by calling logout_user() from Flask-Login.

5.2.2 Friends and Messaging Feature

The friends and messaging features are central to fostering a community environment and enabling social interactions within the game.

Friends Feature: The backend supports functionalities related to the management of friendships between users. Users can add new friends, and view their list of friends. The implementation involves:

- **Friend Request Management:** Users add friends using a specific API endpoint that updates the database to include a new friendship. This is implemented in the endpoint defined in `api.py` (see line 96). The endpoint is:

  ```
  .../add_friend/<string:friend_name>
  ```

  By utilizing the User Data Access, we retrieve data for both the current user and the prospective friend, converting this information into user objects through the get_user function. Once we have the user data, we create a new friendship by employing the add_friendship method from the Friendship Data Access class.

- **List of Friends:** Users can retrieve their list of friends through an API call, which fetches the data from the database and returns it in a formatted JSON array. This feature ensures that users can easily access and interact with their friends list. The relevant API call is implemented in `views/friends.py` (see line 43). The endpoint is:

  ```
  .../api/friends
  ```

Messaging Feature: $*$ TODO These features not only enhance user engagement but also contribute to the game's social dynamics, making the gaming experience more interactive and enjoyable for users.

5.2.3 Leaderboard Feature

The backend manages the leaderboard by fetching and calculating user scores, sorting them, and handling specific leaderboard queries. Here's how it is structured:

- **Fetch and Calculate Scores:** The route

  `'@api/leaderboard'`

  is used for handling GET requests for the leaderboard. It retrieves all users from the database via user_data_access.get_all_users(). For each user, it calculates the total score by summing up the resources associated with each user, fetched by resource_data_access.get_resources(user.username).

- **Sort and Rank Users:** Users are sorted by their calculated scores in descending order. The backend extracts the top three users to represent the leading positions on the leaderboard. Additionally, it retrieves two friends of the current user to include in the leaderboard for more personalized competition.

- **Ensure Inclusion of Current User:** The system checks if the current user is among the top users or friends listed. If not, the current user is also added to the leaderboard. It then removes duplicates and finalizes a ranked list of unique users.

- **Return Leaderboard Data:** The backend returns the leaderboard data in JSON format, including the user's rank, username, and score. This data is then displayed on the frontend for users to view and engage with.

5.2.4 Market Feature

The backend manages the market by storing the current price and adjusting it based on user sales of a given crop. This process occurs in intervals of 1 minute using past and current sales data: route :

`'@game/update-market'`

- **update_price Function:** Calculates a new price for a crop. Adjusts the price based on the current count and a random factor. Constrains the new price within a specific range. Returns the calculated new price.

- **update_market Route:** Handles POST requests for updating market data. Retrieves current market data for a specific crop. If 1 minute has passed since the last update, calculates and updates the price using the update_price function. Updates the market data with the current sales and price. Creates a new entry with a default price if no existing market data is found. Returns a response indicating the success or failure of the update.

- **fetch_crop_price Route:** Handles GET requests to fetch the current price of a crop. Retrieves market data for the requested crop. Updates the price using the update_price function if needed. Returns the current price of the crop in the response.

5.2.5 Building Feature

The backend manages the buildings by storing, updating or fatching the requested building_data: route :

`‘@game/building-map‘`

- **update_map Function:** inserts a given building json in the database. If it exists, it updates, else it creates a new entry.

- **fetch_building_information Route:** Handles GET requests to the requested building info. if the requested building does not exist, returns an error.

5.3 Frontend Interface

The frontend provides an interactive and user-friendly interface using React, allowing players to engage with the game seamlessly. It features a map view for navigating different parts of the farm and a market interface for selling produce.

5.3.1 Login and Authentication Feature

Login Page The login page offers a straightforward interface for existing users to access their accounts.

- **HTML and CSS:** Structure: The login form is embedded within a div element styled with a form-container class. It includes input fields for username and password. Styling: Stylesheets for authentication (auth.css) and footer (footer.css) are linked to maintain consistency in appearance with the rest of the application. The Press Start 2P font from Google Fonts enhances the retro aesthetic of the app.

- **JavaScript:** Dynamic Visual Feedback: JavaScript is used to provide visual feedback on the login button. It changes the button's image when pressed and released, helping confirm user interaction. Form Submission: The form posts data to the /auth/login endpoint, handling user authentication in the backend.

Registration Page The registration page allows new users to create an account by providing essential information such as username, password, and email.

- **HTML and CSS:** Extended Form: Similar to the login form but includes an additional field for email to accommodate the registration process. Consistent Styling: Utilizes the same CSS files as the login page for a uniform look and feel.
- **JavaScript:** Button Feedback: Implements the same interactive feedback for the registration button, enhancing user experience during the account creation process.

5.3.2 MAIN GAME WORLD

The *main game world* consists of several HTML canvases layered on top of each other, each with their own unique JavaScript classes to provide the right functionality. These canvases display 16x16 tiles scaled to a variable size and fill the entire screen. The player can click and drag the screen to look around the game world, interact with buildings, and use the appropriate UI to navigate different parts of the game.

After logging in, the player gets redirected to `https://"baseLink"/game/` which fetches `game.html`. This file is the most important HTML, as it connects and fetches every other file. It includes the UI HTML, the pop-up HTML, sets and positions the canvases, and links `canvas.js`. All the canvases have their own set of tiles which are pre-fetched and stored during the initialization period. This is needed for instantaneous loading; otherwise, a fetch delay is noticeable during runtime.

The HTML UI elements sit on top of these canvases and utilize CSS styling. The only technologies used are JavaScript, HTML, and CSS.

canvas.js This file is most important for initializing the game. It creates all the JavaScript classes used throughout the main game in the correct order (some classes depend on other classes). It is also responsible for correctly resizing all the layered canvases when the browser window's size is changed by the user. Additionally the initialize() functions of some classes are called. These async functions

perform all the correct fetches for the asset's. Then the resize event gets linked and the ticker class gets started.

Terrain Layer The terrain layer provides functionality for correctly displaying the terrain tiles and is the lowest layered canvas, providing as a sort of background layer. The tiles are semi-randomly generated. (TODO Faisal als ge hier nog iets extra over wilt zeggen kunt ge da hier doen). The map is stored as a 2D array of strings that refer to their tile png name. All these assets were fetched beforehand and stored in a map with the tile name as the key. The 'drawTiles()' function uses the map array together with the camera location to correctly draw the tiles on the screen. This function gets called many times throughout the code every time there's an adjustment and the tiles need to be redrawn. When dragging the screen the appropriate move functions of the map get called which change the camera location and redraw the screen.

After starting the game you may notice some tiles are being animated. This is accomplished by utilizing the tick() function and going through the terrain map array, changing specific tiles to the consequent tile in the animation and redrawing them.

Building Layer The building layer works similarly to the terrain layer in terms of rendering the tiles but there are some distinctions and additional features. The way the building map is saved differs from the the terrain layer. The buildings and building information are saved in a json structure. (opslag methode moet geüpdate worden dus hier is nog geen documentatie voor).

The building layer provides support for interacting with buildings. To move a building you can click (without dragging), after which the building is locked to the mouse. Move the mouse, then click again to place the building. When attempting to place a building on an invalid location the building simply doesn't get placed and stays attached to the mouse. Invalid locations include other buildings and water tiles. When trying to drag a building outside the window or outside the map bounds the building doesn't move any further in a way such that it is never in an impossible location. You can drag the screen while moving a building to easily move around the map without needing to let go of the building.

When moving a building it gets rendered on top of all other buildings to avoid it from going underneath other structures around the map. This would lead to a somewhat strange visual otherwise.

Building Pop-up When right-clicking on a building, a HTML pop-up shows up on the right side of the screen. This is used to display various details about the building of interest and also allows you to upgrade the building. The pop-up

contains a "Building Info" title, the name of the type of building, a small text with some information (namely the purpose of the building). On top of that there are, depending on the type of building, potentially some stats that include the level the building is at and other stats like upgrade cost, defense, eggs/hour, etc. Lastly there is an upgrade button and a close button that are both animated when clicked.

When right-clicking another building while a pop-up is already open, a switch animation is displayed to make clear that you are switching to the information of a different building. When right-clicking the same building twice this animation doesn't happen. Lastly when doing almost any other action like dragging the screen and moving a building the pop-up automatically closes.

Main UI The header and footer HTML provide the user with buttons for easy access to different parts of the game as well as displaying some useful information such as the amount of coins and produce the player has.

Ticker class This JavaScript code utilizes the observer pattern. During initialization, classes can subscribe to this class, then when the ticker starts it will call the tick() function on every subscribed class every set milliseconds (1000/24 ms to get 24 frames/second). Classes can then use this function to implement timing sensitive functionality, namely animations.

User Input Handler This is a class that has set up all the event listeners needed to detect useful user input in its constructor. Like the Ticker class it also uses an observer pattern approach, where all classes that potentially need to know about user input get added in the initialization phase of the game.

In this project a click on any of the canvases only gets registered on the mouse up event. This is because we clearly want to differentiate between a click and a drag. To achieve this the user input handler stores the mouse down location. On the mouse up event it compares the current location to the mouse down location and when this distance is smaller then 5 pixels in the x and y direction, a click is registered. This margin ensures that when the user accidentally moves the mouse a few pixels, the input still gets registered as a click.

In order to check on what canvas the user intended to click, a trickle-down method is used. The input handler first checks the first class in the list and calls the 'handleClickInput(x,y)' function on this class. This function will then return a boolean whether the click was applicable to the class or not. If not, the handleClick function is called on the next layer's class and so on.

5.3.3 Friends and Messaging Feature

5.3.4 Leaderboard Feature

- **HTML and CSS:** The leaderboard.html contains the HTML structure, including a table where the leaderboard will be dynamically populated. The CSS file leaderboard.css styles the leaderboard page, ensuring a visually appealing and consistent design.

- **JavaScript:** A JavaScript function loadLeaderboard() (event handling; this function is triggered when document content is fully loaded) is responsible for fetching the leaderboard data from the backend API. Upon successful fetch, it populates the leaderboard table in the HTML dynamically. Error handling is incorporated to manage and alert users in case of failures in data retrieval.

5.3.5 Books

- **HTML and CSS:** The bookChat.html and bookMarket.html contain the HTML structure of the books, covers and pages (front and back), along with the content for the chat and the market respectively.

- **JavaScript:** There is a JavaScript function: openBook() to open the cover of the book and a closeBook() to close the cover of the book. In addition, you have the goNextPage() function to go to the next page and the goPrevPage() function to go to the previous page. In addition to these functions, there are querySelectors for the covers, pages and buttons

6 Challenges and Solutions

Throughout the development, the team encountered and overcame numerous challenges, such as optimizing database queries and ensuring smooth synchronous player interactions. Specific issues included managing the real-time update of resource levels and implementing secure authentication mechanisms.

7 Results and Testing

The game was rigorously tested to ensure functionality across different systems and scenarios. Our testing approach was comprehensive, incorporating automated unit tests, integration tests, and user acceptance testing to cover both the backend and frontend components of our application.

7.1 Backend Testing

For backend testing, we utilized Pytest, a powerful and flexible testing tool. Our tests included:

- **Database Connection Tests:** Using `test_dbconnection.py`, we verified the stability and reliability of our database connections, ensuring that all interactions with the database handled data correctly and maintained integrity under various conditions.
- **API Functionality Tests:** We conducted extensive testing on our RESTful API endpoints to ensure accurate response statuses and proper data formatting. This included testing:
  - **User Endpoint:** Verified that only administrators could retrieve user data, and that the data was correctly formatted as JSON.
  - **Maps Endpoint:** Ensured that map data could only be accessed by administrators, testing for correct map attributes and response format.
  - **Resources Endpoint:** Checked both general and specific resource retrieval endpoints for proper authorization checks and JSON formatting.
  - **Terrain Map Endpoint:** Tested user-specific access to terrain maps, ensuring the integrity and format of the terrain data returned.
  - **Friendships Endpoint:** Assured that users could reliably retrieve their list of friends, with responses properly formatted as JSON arrays.
  - **Market Data Access:** With `test_market_data_access.py`, we insert and fetch new crop elements , to check if the database requests work.
  - **Building Data Access:** With `test_building_data_access.py`, we check if the databse is updated correctly when inserting new elements and fetching them.
  - **Chat Endpoint:** Confirmed functionality for retrieving chat messages between users, focusing on correct data handling and security measures.

  These tests not only verified proper access control and data integrity but also ensured that our server effectively handled errors and returned appropriate status codes under various scenarios.
- **Data Access Layer Tests:**
  - **User Data Access:** Using `test_user_data_access.py`, we tested CRUD operations for user data to ensure accurate storage, retrieval, updating, and deletion of user information.

  - **Map Data Access:** In `test_map_data_access.py`, we verified the functionality of our map management system, ensuring that map data manipulations were handled correctly.
  - **Tile Data Access:** The `test_tile_data_access.py` allowed us to ensure that tile-based operations, crucial for the game's map functionality, were accurate and efficient.
  - **Resource Data Access:** Through `test_resource_data_access.py`, we tested the handling of game resources, confirming the correct implementation of resource accumulation and usage.
  - **Friendship Data Access:** With `test_friendship_data_access.py`, we assessed the systems managing player interactions and relationships within the game.
  - **Chat Message Data Access:** Using `test_chatmessage_data_access.py`, we evaluated the functionality of in-game chat systems, ensuring reliable and secure message delivery and storage.

- **Integration Tests:** We conducted extensive tests to ensure that these individual components functioned together seamlessly, simulating real-world usage to detect any integration issues.

These targeted tests helped us to systematically validate each aspect of our backend, ensuring robustness and reliability throughout the game's infrastructure.

7.2 Frontend Testing

Frontend testing was conducted using Jasmine, focusing on the interactive aspects of our application:

- **UI Component Tests:** Using `canvas-tests.js`, we tested the rendering and behavior of graphical components, such as the game map and resource widgets, to ensure they behaved consistently across different browsers and resolutions.

- **User Interaction Tests:** We simulated user interactions such as clicking, dragging, and keyboard inputs to ensure the UI responded correctly and efficiently without errors or unexpected behavior.

- **Performance Tests:** To assess the application's performance, particularly during peak load times, we measured response times and resource usage to identify and mitigate any potential bottlenecks.

7.3 User Acceptance Testing

User acceptance testing was carried out with a group of target users who provided valuable feedback on the usability and overall experience of the game. This feedback was crucial in refining the gameplay mechanics and interface, leading to several iterations that enhanced user engagement and satisfaction.

7.4 Continuous Integration

We integrated continuous integration tools into our development process, allowing us to automatically run tests upon every commit to our version control system. This helped us quickly identify and rectify issues early in the development cycle, improving product quality and reducing time to deployment. Overall, our structured and thorough approach to testing ensured that 'FarmClash' was robust, user-friendly, and scalable, ready to handle the demands of real-world usage by players across the globe.

8 User Manual

8.1 Installation

Details the steps required to install and run the game locally, as outlined in the README document.

8.2 Gameplay

Explains basic gameplay mechanics, how to interact within the game, and strategies for new players. It covers logging in, starting a farm, upgrading buildings, and interacting with other players.

9 Conclusion

The project successfully demonstrates the application of database and web development skills in creating a functional and engaging multiplayer game. Future enhancements could include more complex defense strategies and a broader range of market dynamics.

10 References

List all references and resources used during the development of the project.