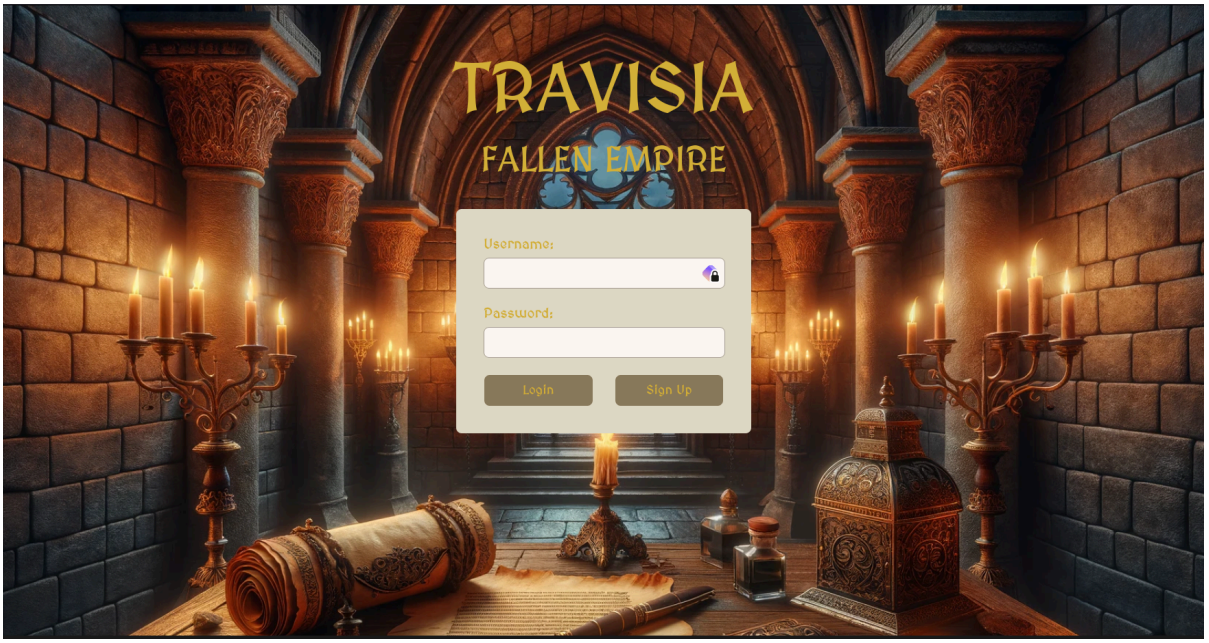
Travisia, Fallen Empire

Technical Report

By Abobaker Rahman, Raadin Bahrami, Jonas Degruytere, Salaheddine Edfiri, Kars van Velzen

Web-based idle game in a medieval context. A project for the course Programming Project Databases (INFORMAT 1002WETDAT)



The logo and welcome screen of our website

Table of Contents

A thriving settlement captured in-game

General Considerations

Features

Singleplayer:

- ☐ City Management
  - ☐ Create and upgrade buildings
  - ☐ Unlock system, e.g. a Barrack unlocks soldiers
  - ☐ Train Soldiers
- ☐ Resources Management
  - ☐ Soldiers consume resources
  - ☐ Creating entities have costs
- ☐ Enhancements
  - ☐ Spin a daily wheel for bonuses
  - ☐ Level & XP System
  - ☐ Achievements
  - ☐ Cookie Clicker Button
  - ☐ In-game Guidance & Explanation
  - ☐ Button Sounds
  - ☐ Background music
  - ☐ Admin Panel to manage seasons
  - ☐ Settings menu

Multiplayer:

- ☐ See other settlements & transfers
- ☐ Interact with settlements & transfers:
  - ☐ Attack
  - ☐ Espionage
  - ☐ Trade
  - ☐ Create new outposts (settlements)
- ☐ Live Map Updates

Social:

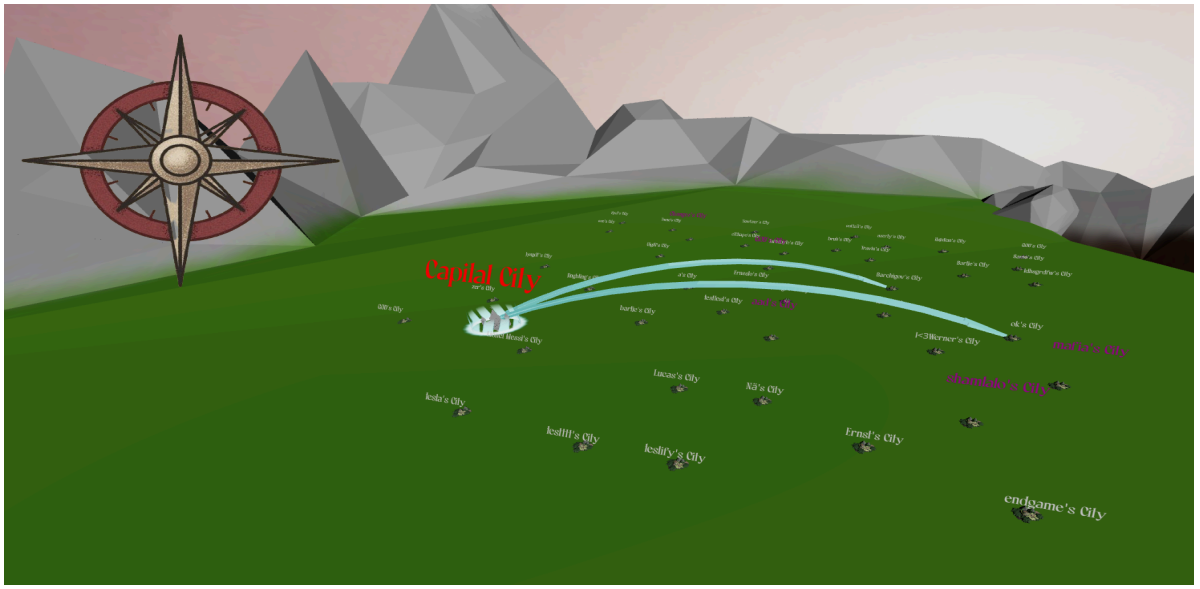
- ☐ Chat with friends
- ☐ Group Chats (Clans)
- ☐ Report system
- ☐ Join & Create Clans (allied)
- ☐ Add friends (allied)
- ☐ Search other players
- ☐ Leaderboard

Guidance:

- ☐ Narrator Story Intro upon sign up
- ☐ In-game explanation of the game

Technical Functionalities

- ☐ Working project
- ☐ Scope & Game Description
- ☐ Documentation & Technical Report
- ☐ Auto-Build for Development
- ☐ Loading Bar
- ☐ Timer System/Synchronisation
- ☐ Multiplayer Mechanics
- ☐ Low server load
- ☐ Expandability
- ☐ Modularity
- ☐ Inappropriate username checker
- ☐ Username validation checker
- ☐ Sounds
- ☐ 3D Visualisation
- ☐ SSL Certificate



World Map with active transfers, captured in-game

Design Vision

We made a **W**eb-based **I**dle **G**ame *in a medieval context.*

Web-based, as usual on the web we had a client-server architecture. We choose scalability and optimisation in mind, meaning our resources should get used reasonably. We chose a system in which the server would only deal with requests in case a client is actively interacting and playing the game.

Idle: The game is idle thus we had to implement a synchronization system. The game should yet have a low server load, thus a socket connection was not an option. We chose to implement our own “timer” system.

The game should be compact, modular and extendable.

Game: A game should be fun and playable. We wanted to enhance our idle game nature to a next level by implementing multiplayer mechanics and making it more immersive. This is why we chose to use 3D Entities, as it looks so much more real!

in a medieval context: We took our inspirations from childhood games like GoodGameEmpire & Clash of Clans, like the project description stated.

The game’s graphics, resources and types are chosen with care and of course placed in the medieval ages.

An abstract of the lore:

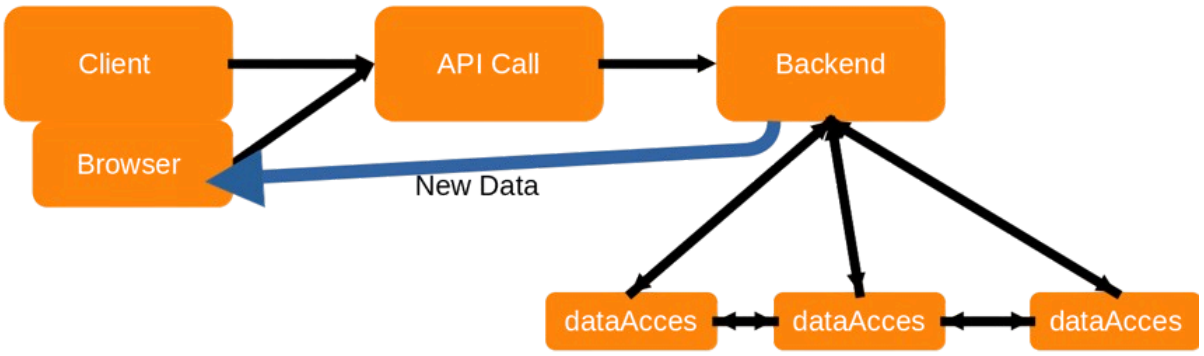
“A century has passed since the empire's fall, yet its shadow still looms large over the fractured lands it once ruled. The power vacuum caused countless factions to emerge, each vying for control over the empire's former territories. Warlords, noble houses, religious orders, and ambitious individuals of all stripes seek to carve out their own domains, driven by visions of power, glory, or survival in this new, uncertain world.
This is where you come in. Will you, the ruler of a small village, be able to fend off rival warlords, forge new alliances and expand your village to be the shining capital of a new empire capable of bringing peace and prosperity to this desolate region once again?”

For the complete story, please refer to our scopeDescription and the in-game narrator.

Project Overview

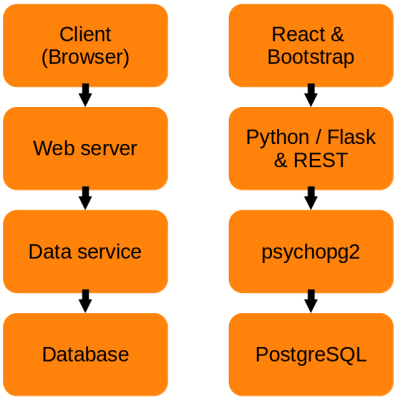
The application works upon a client request (e.g. clicking a button) who will make an API call using the frontend. Then the backend delegates the request towards different access entities who will handle the request by using the database. The different sub components might work interactively with each other.

New data will be sent back to the client, which the frontend will visualize.



Visualization of the workflow from the projects documentation

We chose to implement the frontend using React. The server framework is based on Flask (Python) in combination with a psycopg2 database connector. The database itself is PostgresQL

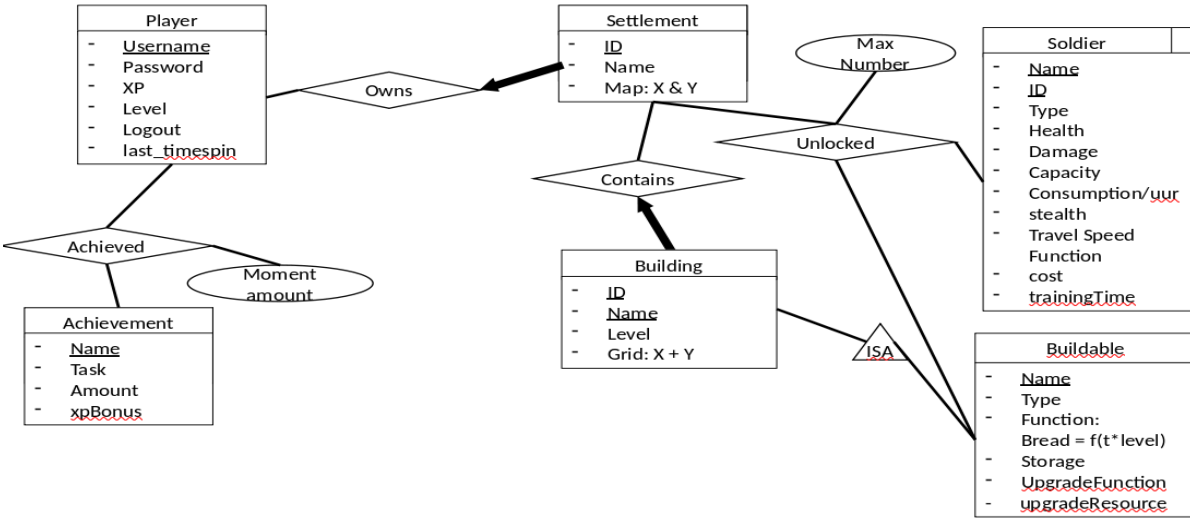


Design Choices

Database Design

The general idea is that the user owns and manages a settlement. They create resources and upgrade/build their own buildings to increase their level. A player can train different types of soldiers and attack other players. The main gameplay centers around sending "packages/transfers" around the world. A transfer can be an attack, espionage, trade (also named transfer). A transfer goes to a settlement or another transfer that currently travels around the map. You can capture and recapture transfers as long as they move around the globe. Stealing each other's transfer is an essential game mechanic! Multiple enhancements are added to improve the user experience; social menus, quests, levels & guilds.

Our database is used to store the different types of information in a compact way, minimal information is stored but yet enough to reflect the current state of the game.



Slide of the ER-Diagram considering the SinglePlayer part of the game

Player

Players are uniquely identified by their username. The login is only password protected. The frontend verifies the integrity of the username.

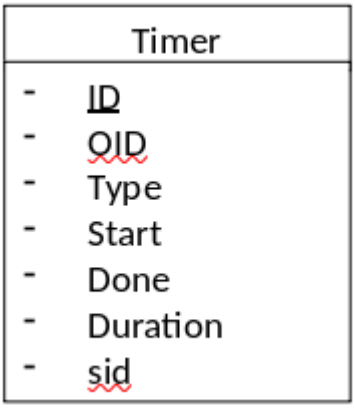
Each player has an amount of gems, a level and associated XP. The last logout timestamp is kept to compute the time in between a user logs in and out. This is used to generate new resources overtime.

It is possible to achieve Achievements/Quests, which are uniquely identified by a name. A task is given. Each Achievement contains a xp bonus which is rewarded after completion. The amount contains a number which is decremented, if it has reached zero, the task is completed.

Optionally, a user can interact with a Wheel of Fortune to get a bonus package, this is why the last time of the spin is kept since you can only spin once in 24 hours.

Players own one or more Settlements.
There is one special user; the admin. (Game Administrator)
Players can befriend each other.

As well as the achievements, the soldier & buildable class are made to ease expandability. By inserting new entries in those tables and some minor changes in the code, new achievements, soldiers & buildings can be introduced. Also, since most entities are identified by a numerical id, they can be easily used interwined. This choice is reflected in the timer class, in which the “oid” (Object ID) can refer to multiple objects at the same time.



Settlement

A settlement is owned by a single player. It is identified by an ID (serial) and can be given a name to ease communication for the user.
The unique location (enforced by constraints) on the map is stored in the settlement.

Settlements contain buildings and can unlock them. A settlement can also unlock soldier-types. A certain level is required for each entity and defines the maximum amount of types a settlement may have. The unlocked relation is explicitly expressed in 1 table, to keep things compact.

A settlement has a finite amount of resources. It can produce more overtime by using the correct buildings.

Soldier

Different types of soldiers with strengths towards other types exist. For example: An archer will have a damage advantage over infantry types but infantry could bring more resources with them.

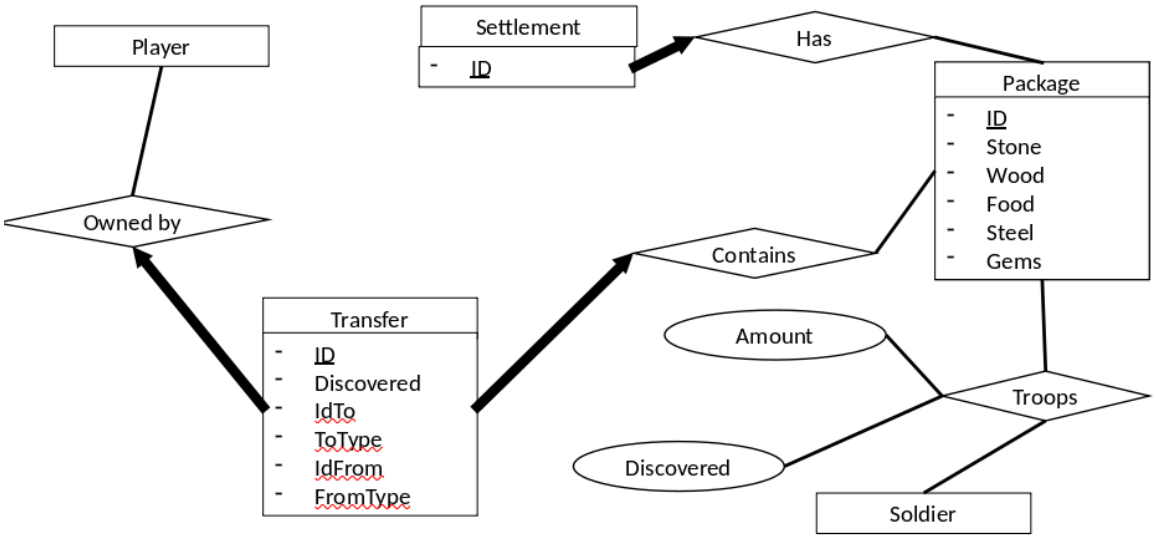
Soldiers have an amount of health and damage they do. Also, they consume food at a time function. When food runs out in the settlement, the soldiers will die or desert. Some soldiers might move faster than others. Plan your transfer strategically! A soldier has a cost & time to train, which is specified per type.

Building

You can place multiple buildings of the same type. The types are referred to as "Buildables" and have a unique name. Buildables keep general info like the timeFunction. Inheritance is used to link this info to a “Building”, which is defined per settlement. E.g. It has info about the location of the building or the level. Therefore users can have multiple buildings of the same type. This is convenient as well as compact. Once again does this decision contribute to database expandability since new types can be added easily.

Buildings perform an action specified in a category (type). Farms may produce food, woodcutters produce wood for upgrading buildings. Some buildings store amounts of resources and others allow for bonus effects such as increasing the health of a soldier type. Refer to the other documentation for specific amounts & functions.

A building costs resources to build, which are specified in the upgradeFunction. The time needed to build the building is specified in a timeFunction. We use a more technical representation of these functions to evaluate them with ease and, logically, keep things compact.



Slide of the ER-Diagram concerning the Multiplayer part of the game

---

Package

A package is an entity to store resources. It is purely for technical convenience and hasn't anything to do for the user.
Uniquely identified with an id. All other parts are optional and are amounts. e.g. Stone is the amount of stone in the package.

Soldiers are linked to the package via the Troops relation in the database. An amount is specified, as well as a boolean: Discovered.
The discovered boolean is used to determine if the soldier is spotted/spied on by a rivaling player.

Each settlement has an associated unique package, and thus a finite amount of resources and troops.

Using this entity makes resource management a lot easier!

Transfer

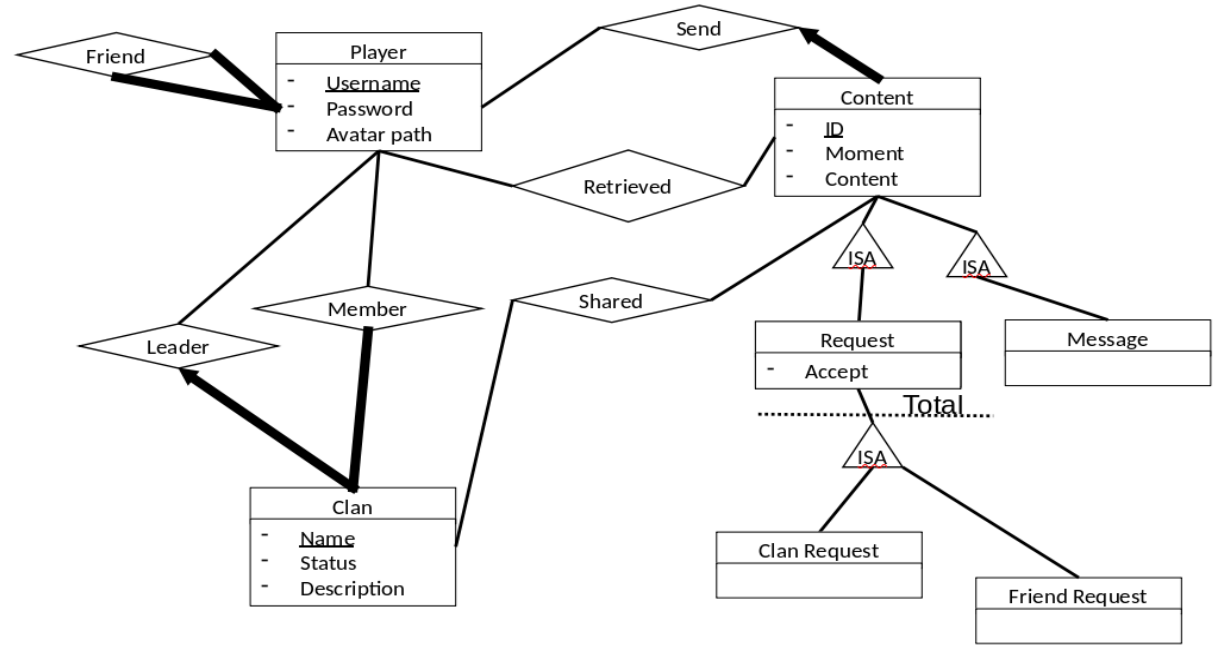
Transfer, uniquely identified by their ID (Serial) are the entity used to, quit literally, transfer resources from location to location.
The speed of the transfer is determined by the speed of the soldiers, amount of resources and the distance. [Read more in the backend documentation](../backend/transfer.md)

The discovered bool is used to indicate if rivaling players can see this transfer.

Transfers move from a settlement/transfer towards another settlement/transfer. Transfers can be captured/intercepted by other transfers.

As should be clear, a transfer also contains a package. A transfer is owned by a player. (A transfer can be intercept or go towards another transfer so it would get complex fast to compute this with ease. To prevent this complex behavior, the extra relationship was introduced.)

A transfer can go to a transfer or a settlement, which both use an ID! This way we didn't have to make any specialization, only a compact bool defines this. It also prevented us from having multiple tables explaining these more complex relationships.



Slide of the ER-Diagram concerning the social aspect of the game

Clan
A clan is uniquely identified by their name. They have a status, like an oneliner/headline and a more formal description, specified by the clan Leader: a unique player. Players can optionally join a clan. (Member)

Allied players can transfer goods to support each other and bring peace to their alliance: they can't attack each-other. (Similar for befriended, thus allied, players)

Clans are also favored with a group chat.

Content
A content can be a message or a request. The send time & content (actual message, e.g. "Hi friends!") is saved. Content is identified by a serial; id.

Contents/Messages are sent from one player to another, to perform direct messaging. The shared relation is used to make a group chat for a clan.

A request is a specialization of content and has a status; True = Accepted, False = Rejected. It is used to store Clan Invitations and Friend Requests (Inherited specializations). Using inheritance, once again, gives us a more compact database.

Timer
A building, soldier or transfer might take time until it arrives. In the database we keep track of the current actions that are running. Upon user interaction, the server checks if there are timers that need to be resolved.

Timers have a unique id and refer to an unique object (OID). This object is an entity in the database and corresponds with the timer type; e.g. building, transfer, ...

The start time, end time (which can be computed at start) and duration are saved.

Timers also have a sid which refers to a settlement or a transfer, which is the owner of the specific action. You could consider a timer as an ongoing action.

By saving this small part of information in the database, our whole synchronization system can work in a simplistic manner.

Backend Design

In the backend, we chose to work with functionality that comes from classes we have created. The classes are similar to the entities of the database. We made it similar so we can easily store and retrieve data. (E.g. the package object can be initialized with the array data from the query) Also, when changes are made in the database, you can change it very quickly in the functionality. So the interaction between the functionality and the database is very efficient. Also, this is why the backend contains flexibility and expandability . Furthermore, our backend is stateless. It will only execute functionality upon API requests. This means that the server load is reduced to a minimum and depends solely on the amount of users actively playing. The backend does make sure everything stays consistent.

We have created a clean codebase because the backend is modular. Everything is divided into libraries to avoid any code duplication. It is also very clear for someone who wants to change something in the functionality where to look and change it. To avoid duplicated code, helper functions are made to keep things compact.

We chose a central root file, which was provided by the course. This choice visualizes the structure of our application tree. We also used object composition in Python to make working with data easier and to keep the code logical and well-structured. (E.g. the PackageWithSoldiers object which contains the package data from the database as well as the Troops relation to do similar computations at ease. Therefore we overloaded the arithmetic operators too)

Our backend performs various calculations and simulations, including resource management. This ensures that the application efficiently manages available resources, leading to optimal performance and resource allocation.

We chose to create our backend using REST API and PostgreSQL. The REST API facilitates effective communication between the backend and frontend. PostgreSQL was selected as the database for our game due to its robustness, efficiency and reliability. Using the REST API allowed us to test each route thoroughly with Postman, ensuring the functionality works as expected.

We designed the backend close to the database to be efficient, structured and coherent with each other. Combining these aspects created a solid, yet flexible backend for our game.

Frontend Design

Below, you'll find a brief explanation of our design choices and tools we've selected for our user interface.

React Components

The frontend is structured using React components, each responsible for specific parts of the user interface and game logic. This modular approach allows for easy updates and maintenance. In addition, we utilized React Three Fiber for creating immersive building structures and settlement modals. This technology allowed us to seamlessly integrate three-dimensional elements into both the main page and the map interface.

React

React was chosen as the primary frontend framework due to its simplicity and built-in features which enhances frontend web development.

- **Component-Based Architecture**: We structured our application into reusable components to promote code reusability and maintainability. The React project initially creates a main.jsx (root JSX file) which is bonded with the head HTML file. In this file, the App.jsx is used which contains the routes to all the game pages.
- **Hooks**: We used built-in React hooks such as `useState`, `useEffect`, `useRef, useMemo` for better code rendering since React recommends it a lot (actually forced to use it, otherwise you'll end up with an error).
- **Conditional Rendering**: Used conditional rendering techniques to display different components based on application state and user interactions.

React Three Fiber

React Three Fiber (R3F) was chosen for rendering 3D buildings and settlement modals. We could've used Three.js itself but obviously React-Three-Fiber is made for React and makes Three.js itself simpler to use:

- **Canvas Component**: The `<Canvas>` component of R3F made everything simpler since it already contains the initial code of the 3D

scene of the Three.js library such as a basic camera view, lighting and so on.
- Other built-in components such as `directionalLight, ambientLight and OrbitControls` made our job a whole lot easier.

Frontend API Integration

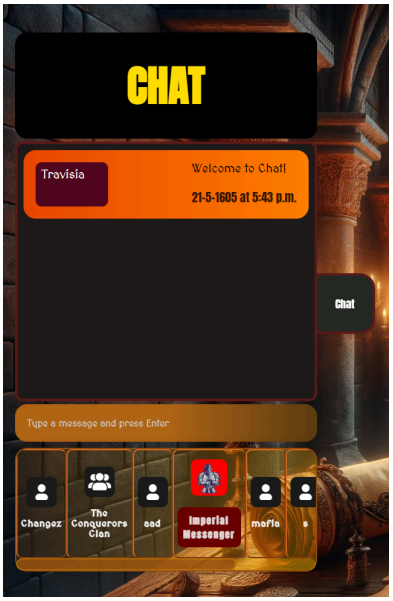
Our frontend interacts with backend APIs to fetch and update data, enabling dynamic and real-time features within our application.

- **Axios for HTTP Requests**: Used Axios, a promise-based HTTP client, to make asynchronous HTTP requests to our backend APIs from within React components.
- **RESTful API Design**: Followed RESTful principles to design our backend APIs, ensuring consistency, scalability, and ease of integration with the frontend.

Styling

- **CSS Modules**: We used CSS Modules to keep our component styles organized and avoid any messy conflicts between styles in different components of our game.
- **UI Libraries**: We used lots of great and attractive button- and input styles from uiverse.io

---



In-game chat with clan and private messaging .

Copyright Statements
We don't own any right to the used artworks in our game (pictures, audio, 3D Models). We call upon the educational use of any potentially copyrighted materials. Throughout our files and code you may find credits and reference to the original authors, where applicable. (e.g. see documentation/frontend/buildingList.md)

Closing Comments

References

Our server setup is initially based on the setup from Joey De Pauw, course assistant.

User Feedback

The general opinion about people playing our game was that they were very enthusiastic about it and found our game fun to play. Some people had questions about the game mechanics and what they could do. Negative feedback was minimal. It was really helpful for us to catch bugs and figure out what was giving us trouble. We thoroughly listened to their feedback, e.g. We created a video to guide everyone through understanding the game mechanics and how to play it. (Since some users seemed unsure about what to do).

In-game guidance video demonstrating how the game operates.

Conclusion

Regarding the produced functionality, we decided that our project matches the criteria from our scope description and the project assignment. We implemented a **W**eb-based **I**dle **G**ame *in a medieval context.* Components are coherent The codebase is documented, compact and focused on extendability. Moreover, we extended it with multiple enhancements to make the game more interactive & smooth, improving user experience. Think about the guidance video, intro video, button optimisations, cookie clicker, sounds, appropriate username checker et cetera. (See the list at the begin of this document) User feedback was helpful to make our game more fun and playable.

Frame of the game introduction video upon sign-up.

By Abobaker Rahman, Raadin Bahrami, Jonas Degruytere, Salaheddine Edfiri, Kars van Velzen