Ordinary Wizards - Technical Report

Ordinary Wizards - Technical Report

Authors: Joren De Smet, Aaditiya Nepal, Flynn Mol, Daria Matviichuk, Lucas Vanden Abeele & Thomas Urkens

Programming Project Database, Group 2

Created at the University Of Antwerp, 2024 for the course Programming Project Databases (2nd Bachelor's degree in Computer Science)

This technical report has been autogenerated and composed from the project's documentation found **here**. This file may have wrong formatting and dysfunctional/misplaced images/GIFs. Although the essence is still here, we recommend for the full experience refering to the original documentation at the GitHub repository. You can also open the original markdown files with a proper markdown viewer.

Table of Contents

Readme

Introduction

Web-based 3D idle game using THREE.js, Flask, PostgreSQL, SQLAlchemy, SocketIO & Alembic. Made by 6 students for the course Programming Project Databases at the University of Antwerp.

Ordinary Wizards is a web-based 3D idle game where you are a magical wizard that can build his own island, mine crystals & gems, progress through multiple levels and fight other players in a real-time multiplayer battle. With the power of WebGL, the game is completely rendered in 3D and can be played on any device that supports a modern web browser. Craft magical gems to boost your buildings, upgrade your island, beat other players and become the most powerful wizard in the game!

Project setup

The project is divided into two main parts: the backend and the frontend. The backend is a RESTful API built using Flask, SQLAlchemy, flask-restful and flask-migrate, that communicates with an external PostgreSQL database. The frontend is a multi-page (landing, login, register & index pages) web application built using jQuery, THREE.js, SocketIO and out-of-the-box JavaScript, CSS & HTML.

Installation & running debug server

Execute all the following commands from the root directory of the project.

1. Setup a Python (3.10+) virtual environment and install the required packages using the following commands:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

2. Setup a PostgreSQL database and fill in the connection details in the (user created) `.env` file:

```
APP_POSTGRES_HOST=127.0.0.1
APP_POSTGRES_PORT=5432
APP_POSTGRES_DATABASE=ordinary_wizards
APP_POSTGRES_USER=ordinary_wizards
APP_POSTGRES_PASSWORD=<password>
```

3. Setup app environment variables as well in the `.env` file: For a full, detailed list of environment variables, see the Environment Variables chapter.

```
APP_BIND=127.0.0.1
APP_HOST=127.0.0.1:5000
APP_HOST_SCHEME=http
APP_SECRET_KEY=<secret_key> # optional
```

```
APP_NAME=Ordinary Wizards
APP_JWT_SECRET_KEY=jwtRS256.key
```

4. Generate secret `jwtRS256.key` using the `keygen.py` script:

```
python3 keygen.py
```

5. Make sure to run the following command to create the database tables:

```
flask --app src.app db upgrade
```

6. Run the Flask debug server using the following command:

```
python3 -m src.app
```

Important note when running Gunicorn WSGI server: Gunicorn does not support WebSocket connections (as the loadbalancing algorithm does not work with WebSockets), which this app requires. In order to fix this, you must run gunicorn with only **1 worker** and use threading for workload spread.

Intellectual Property

All assets used in this project are free to use. You can find the original sources of the used assets as well as their authors in the credits.txt file.

Documentation

Due to the complexity of the project, we have decided to split the documentation into multiple parts. This 'wiki' file will explain what documentation type does what. Some documentation types require generation by the user (PyDoc & JSDoc), please see below how to do so.

In general, the documentation is divided into the following parts:

- General documentation (auth flows, API conventions, etc.): see this document
- REST API: see Api chapter and the Swagger documentation at `http://127.0.0.1:5000/api/docs` (when running the debug server locally with `APP_SWAGGER_ENABLED=true`)
- Entity-Relationship (ER) diagram: see diagram at draw.io **here** and in the last chapter **Entity-Relationship Diagram**
- Backend code documentation: see PyDoc documentation at `/pydocs`
- Frontend code documentation: see JSDoc documentation at `/jsDocs`
- High-level user manual, with links to technical details: see **user manual**
- Test cases: see **tests**
- Used assets: see **credits.txt**

Backend code documentation

The backend code documentation is generated using PyDoc.

To generate the documentation, run the app with `APP_GENERATE_DOCS=true` in your environment variables. Then, HTML documentation will be generated in the `pydocs` folder, starting from the `index.html` file.

Frontend code documentation

The frontend code documentation is generated using JSDoc and therefore requires the usage of the `npm` package manager. First, setup the project. You only have to do this step once:

```
sudo npm install -g jsdoc
sudo npm install @ckeditor/jsdoc-plugins
```

Then, to generate the documentation, run the following command from the root directory of the project:

```
jsdoc -c jsdoc.json
```

Then, HTML documentation will be generated in the `jsDocs` folder, starting from the `index.html` file.

API information

For all api endpoints and their specific documentation, please refer to the Swagger documentation

Swagger Documentation

Available at `<url>/api/docs` with `APP_SWAGGER_ENABLED=true` set as environment variable when using a local development server.

General API structure (backend)

For the complete, detailed relationships between entities: please refer to the ER diagram.

The entire API backend is stateless and RESTful. This means that multiple instances of the backend can be run at the same time, and they will all function correctly. However, the websocket server (everything inside `src.socketio`) is not stateless and should be run as a single instance. Running multiple instances will result in undefined behaviour.

Each entity has a Model, a Schema and a Resource class. The Model lies in the `model` package, in a file with the same name as the entity. The model is the Mapper for SQLAlchemy and has all column definitions and relationships between other SQLAlchemy models.

A Schema lies in the `resource` package, in a file with the same name as the entity. The class name (of the schema) is the entity name appended with `Schema`. The Schema defines and parses the JSON representation of the entity. It should function as a JSON to Entity Mapper and reverse. The Schema is used to validate input JSON bodies, format output objects and documentation.

A Resource lies in the `resource` package, preferably in the same file as the Schema. The resource functions as a RESTful API endpoint that accepts HTTP requests. As per RESTful spec, it has at most 4 methods for each CRUD operation on an entity. Each function has `@swagger` decorators with information about what schema is returned / used in what use case / error message.

- `post()` - HTTP `POST` for create
- `get()` - HTTP `GET` for read
- `put()` - HTTP `PUT` for update
- `delete()` - HTTP `DELETE` for removal

Please note that not all endpoints support all CRUD operations, please refer to the Swagger documentation for more information.

A Resource parses the query methods and/or JSON body and manipulates / retrieves the SQLAlchemy Models accordingly. It functions as both the Controller and the View for an entity.

Endpoints of objects that allow listing of all objects are usually suffixed with `/list` (e.g. `/api/spell/list`).

Conventions on API responses

All API responses are JSON objects. Except the `DELETE` method, all HTTP 200 responses will return the object itself rather than a status and/or message field. All non-200 responses will have a `status` key with the value `error` and a `message` key with a human-readable error message. Please note that the `auth` endpoints don't follow these conventions, as they are not part of the RESTful API.

For more details, please consult the Swagger documentation.

- `GET` - Unless specified otherwise, requires always an `id` parameter in the **query string**. If the id is not found, it will return a 404. It will always return the object schema on HTTP 200. No status key is then returned.
- `POST` - Requires a JSON body with the object schema. **No id** (Primary Key of the entity) has to be present in the request body (it is ignored anyway). **ALL** other fields are **mandatory** (unless specified otherwise). It will automatically be assigned and returned in the response body. It will return the object schema on HTTP 200. It will return a 400 if the object is not valid.
- `PUT` - Requires a JSON body with the object schema. An **id (PK) has to be present** in the **request body**. All other fields are *optional*. Please note that not all fields are updatable. Please refer to the Swagger & PyDoc documentation for more info. A JSON object (schema) is returned on HTTP 200. It will return a 400 if the object is not valid. It will return a 404 if the object is not found.
- `DELETE` - Requires an `id` parameter in the **query string**. It will return a HTTP 200 with `status` key `success` if the object is deleted. It will return a 404 if the object is not found.

Unless clearly specified otherwise, the `type` field is always IGNORED in the request body. The `type` field is used for polymorphic identities. For POST request, this type is determined by the used endpoint and is therefore also never required. When applicable, the `type` field is present in the response body and can be safely used to determine the polymophic identity (so which fields are present in the response body), depending on the properties the polymorphic object has. Eg. a mine has a `mined_amount` field that is only present in the `Mine` object. The `type` field can be used to determine if the object is a `Mine` or not. For a complete overview of polymorphic object and their fields, please refer to the ER-diagram.

A special note on registering new endpoints

Each RESTful endpoint (=Resource) should be registered with an `attach_resource()` function defined in the resource file. This method should be mostly copied over from other examples, and be modified accordingly. Finally, add this function to the `resource.__init__#attach_resources()` by importing it locally and invoking it.

The reason this is done this way is because the current flask-restful-swagger-3 package does not support multiple Flask route blueprints. So they all have to be bundled to one. The package is the only that adds Swagger 3 support, but is unfortunately unmaintained and broken in many ways.

A special note on creating new buildings, entities and tasks that belong to an island

When creating a new entity or building that belongs to an island (and all implementing classes), the accompanying `Schema` should also be registered in the `resource/island.py` file. For new task subclasses, these schema's should be registered in the `resource/placeable.py` class. Entity schema's should be added to the `_resolve_placeable_schema_for_type()` method and buildings in the `_resolve_building_schema_for_type()` method. Tasks in the `_resolve_task_schema_for_type()` method in `placeable.py`. This is necessary for the `island` class to parse the entities and buildings to JSON values to pass on to the frontend.

A special note on grouped objects

Grouped objects such as `placeble` and `entity` have all a common `DELETE` endpoint (`/api/placeable` for placeables, `/api/entity` for entities). This endpoint will delete all objects that are related to the object that is being deleted. No specification of the subclass / type of object is needed.

Access constraints

All non-GET endpoints (POST, PUT, DELETE) must be invoked by either an admin or the user that owns the object (= the user/player that has said object associated with himself or the island it owns). Failure will result in a 403 error. GET endpoints are not subject to this constraint because they are read-only and do not modify the database.

Disabling this constraint can be done by setting the `CHECK_DATA_OWNERSHIP` environment variable to `false`. This will however still log a warning to the server console.

Data constraints

The following variables have certain constraints on their values. It is safe to assume these names are unique across the application (therefore these constraints are applicable accorss all variables with said name).

- `level`: `value` >= 0 (entities & buildings)
- `xpos`: `value` >= -7 and `value` <= 7 (grid size)
- `zpos`: `value` >= -7 and `value` <= 7 (grid size)
- `rotation`: `value` >= 0 and `value` <= 3 (north, east, south, west)
- `cost`: `value` >= 0 (blueprints)
- `buildtime`: `value` >= 0 (blueprints)
- `used_crystals`: `value` >= 0 (tasks)
- `to_level`: `value` >= 0 (tasks)
- `multiplier`: `value` >= 0 (gem attributes)
- `mined_amount`: `value` >= 0 (mines)
- `crystals`: `value` >= 0 (player)
- `xp`: `value` >= 0 (player)
- `mana`: `value` >= 0 and `value` <= 1000 (player)
- `audio_volume`: `value` >= 0 and `value` <= 100 (user settings)
- `performance`: `value` >= 0 and `value` <= 3 (user settings)
- `selected_currency`: `value` >= 0 and `value` <= 2 (user settings)

- `horz_sensivity`: `value` >= 0 and `value` <= 100 (user settings)
- `vert_sensivity`: `value` >= 0 and `value` <= 100 (user settings)

Authentication

For all api endpoints and their specific documentation, please refer to the Swagger documentation

Sitemap (excluding API)

- `/` - Index page: this is the main game application page. It is only accessible when the user is logged in. When the user is not logged in (no JWT token), the user is redirected to the landing page.
- `/register` - Register page: this is the registration page. It is only accessible when the user is not logged in. When the user is logged in, the user is redirected to the index page.
- `/login` - Login page: this is the login page. It is only accessible when the user is not logged in. When the user is logged in, the user is redirected to the index page.
- `/logout` - Logout page: this is the logout page. It simply unsets the `access_token_cookie` cookie.
- `/landing` - Landing page: this is the landing page. It is always accessible and is the first page the user sees when accessing the application. It contains a brief description of the application and a link to the register page.
- `/password-reset` - Password reset page: this is the password reset page. It is only accessible when the user is not logged in. When the user is logged in, the user is redirected to the index page.

JSON-Web-Tokens (JWT)

The application uses the stateless JSON-Web-Tokens for user authentication. The inner functions are primarly managed by the `flask_jwt_extended` package. The `JWTManager` instance can be accessed after initialisation through the `app.jwt` or `current_app.jwt` property. Protected endpoints are marked with the `@jwt_required` decorator. The invoking user identity can be accessed through the `get_jwt_identity()` function. Please refer to the flask_jwt_extended documentation for more information.

The JWT tokens are signed using the binary contents of the file in the `APP_JWT_SECRET_KEY` environment variable. A secure key can be generated using the keygen.py script. The JWT tokens are used to authenticate users and to authorize them to access certain resources.

There are currently 2 roles. An `admin` role and a `user` role. The `admin` role has the ability to open developer tools on the frontend, see/modify other user (protected) data and promote another user to admin. The `user` role is the default and has access to all api endpoints, but is limited to read-only unless the user owns the entity.

The tokens are stored as a `http-only` `access_token_cookie` cookie. The cookie is only valid for the domain of the site (as per default cookie behaviour). A token is valid for 1 hour by default, but can be changed through the `APP_JWT_TOKEN_EXPIRE` env variable. The token is automatically refreshed if the user makes a new request 30 minutes before the token is about to expire.

User Profile Permissions

Across all API endpoint a data ownership check is performed. This means that a user can only create (POST), modify (PUT) or delete (DELETE) data that is owned by the user. A user can only see (GET) data that is not owned by him. This is done by checking the `user_id` field of the owning entity against the JWT identity that the invoking user uses to authenticate in the first place. Violations of this rule will result in a `403 Forbidden` response.

Please refer to the individual API endpoint documentation for more information on the permissions of each endpoint.

Password-based authentication

Registration

The app supports registration of new users. The registration endpoint is `/api/auth/register` and accepts a `POST` request. Registration can be enabled/disabled through the `APP_REGISTER_ENABLED` environment variable. A frontend is available at `/register`.

On registration, the user is required to provide a unique username, firstname, lastname and password. The password is hashed with a salt using the `bcrypt` package. Both the password hash and the salt are stored in the database. The password is never stored in plaintext. There is currently no email verification or captcha required for registration. There is also no possibility to reset a forgotten password.

Once a new user is created, the user is automatically logged in and a JWT token is both returned in the response and as a `http-only` cookie. When using the frontend, the user is automatically redirected to the index page, which is now accessable. On creation, a new Player and Island object are created among the UserProfile object. This is because these 3 entities all have a weak, one-to-one relationship with each other (see ER-diagram).

Login

Login is onde through the `/api/auth/login` endpoint. The endpoint accepts a `POST` request with the username and password in the request body. Login can be enabled/disabled through the `APP_LOGIN_ENABLED` environment variable. A frontend is available at `/login`.

On login, the user is required to provide a username and password. The password is hashed and compared to the stored hash in the database. This comparisation uses the `bcrypt.checkpw()` function, which is safe from timing attacks. If the hashes match, the user is logged in and a JWT token is both returned in the response and as a `http-only` cookie. When using the frontend, the user is automatically redirected to the index page, which is now accessable.

Note: There is NO CSRF protection in the entire application. This would otherwise complicate the use of the application even more and is beyond the scope of this project. When implementing CSRF protection for the JWT cookie, please refer to this page.

OAuth2 based authentication

The app currently supports the OAuth2 protocol for external authentication. The current design works for Google login, but should be usable for any OAuth2 complient provider (slight modifications may or may

not be required). For more information about the workings of the OAuth2 protocol & implementation, see the oauthlib documentation.

To enable OAuth2, set the following environment variables in the `.env` file (see env.md for a complete list):

```
APP_OAUTH_ENABLED=true
APP_OAUTH_CLIENT_ID=<client_id>
APP_OAUTH_CLIENT_SECRET=<client_secret>
APP_OAUTH_DISCOVERY_URL=<discovery_url> # for Google, this is https://accounts.google.com/.well-known/op
```

To start a new OAuth2 procedure, call the `/api/auth/oauth/login` `GET` endpoint. This endpoint will redirect the user to the OAuth2 provider's login page for this application. When the user has successfully authenticated, the OAuth2 provider should redirect the user to the `/api/auth/oauth/callback` `GET` endpoint with the `state` and `code` query parameters set in order to complete the authentication process.

OAuth login with Google is possible through the frontend by using the 'continue with Google' button on the login page. When a user has no account associated with its Google account (identified by the SSO id), a new account is created like the password-based registration. In either case is the user is automatically logged in on succesful authentication.

Note: The `APP_DEBUG` environment variable can be set to `true` to enable insecure oauthlib transport (so you can host your site over http instead of https). This is useful for development purposes, but should never be enabled in production (and is therefore disabled by default in oauthlib).

List of cheats

The following is a list of cheats the admin can use to speed up the game process.

Command	Description
`\level()`	change the current level
`\mana()`	change the current mana
`\xp()`	change the current xp
`\position()`	change the current position
`\crystal()`	change the current crystal
`\shieldCooldown()`	change the cooldown of shield spell
`\fireCooldown()`	change the cooldown of fire spell
`\thunderCloudCooldown()`	change the cooldown of thunder cloud spell
`\buildCooldown()`	change the cooldown of build spell
`\health()`	change the current health
`\forceSpells`	set required mana of all spells to 0.
`\forceBuilds`	build Buildings with no timer.

List of environment variables

The following is a list of environment variables that can be set in the `.env` file. The `.env` file is used to store sensitive information and configuration settings for the application. The `.env` file is not included in the repository and should be created manually. The `.env` file should be placed in the root directory of the project.

Note: this table does not display properly in the PDF file. Please refer to the Markdown file here for a better rendered version.

Key	Value type	Description	Default Value	Required
`APP_POSTGRES_HOST`	string	The host (domain or IP) of the PostgreSQL database	127.0.0.1	True
`APP_POSTGRES_USER`	string	The user to login into the PostgreSQL database. This user has to have create, read, update and delete permissions on the database.		True
`APP_POSTGRES_PASSWORD`	string	The plaintext password for the PostgreSQL user.		True
`APP_POSTGRES_DATABASE`	string	The name of the PostgreSQL database to connect to.		True
`APP_POSTGRES_PORT`	integer	The port number of the PostgreSQL database.	5432	True
`APP_BIND`	string	The IP address to bind the Flask application to.	127.0.0.1	True
`APP_HOST`	string	The host domain of the Flask application. This is how you would connect to the app through a webbrowser.	localhost	True
`APP_HOST_SCHEME`	string	The scheme to use for the Flask application.	http	True
`APP_SECRET_KEY`	string	The secret key used to sign the session cookie.	Random string	False
`APP_NAME`	string	The name of the application.		True
`APP_JWT_SECRET_KEY`	string	The secret key used to sign the JWT tokens. Generate this key using the keygen.py script.	jwtRS256.key	True
`APP_JWT_TOKEN_EXPIRES`	integer	The number of seconds a (new) JWT token is valid for.	3600	False
`APP_DEBUG`	boolean	Enable/disable debugging. Setting this to `true` will enable Flask debugging, debug level logging and insecure oauthlib transport.	False	False
`APP_SWAGGER_ENABLED`	boolean	Whether to enable the Swagger documentation at /api/docs.	False	False
`APP_LOGIN_ENABLED`	boolean	Whether to enable the login endpoint. Disabling this will return an HTTP 409 when invoking the /api/auth/login endpoint.	True	False
`APP_REGISTER_ENABLED`	boolean	Whether to enable the register endpoint. Disabling this will return an HTTP 409 when invoking the /api/auth/register endpoint.	True	False
`APP_DISABLE_SCHEMA_VALIDATION`	boolean	Whether to validate the Database schema on launch or not. If the schema does not match, the application will not start (unless automigration upgrades the schema). It is recommended to keep this enabled.	False	False
`APP_AUTOMIGRATE`	boolean	Whether to upgrade the database schema on application start if it's out of date with the latest revision.	True	False

Key	Value type	Description	Default Value	Required
APP_OAUTH_ENABLED	boolean	Whether to enable OAuth2 login/registration or not. Disabling this will return an HTTP 409 when invoking the /api/auth/oauth/login endpoint. When enabled, OAuth2 client id and secret are required.	False	False
APP_OAUTH_CLIENT_ID	string	The client id for the OAuth2 login/registration.		False (unless APP_OAUTH_ENABLED=true)
APP_OAUTH_CLIENT_SECRET	string	The client secret for the OAuth2 login/registration.		False (unless APP_OAUTH_ENABLED=true)
APP_OAUTH_DISCOVERY_URL	string	The discovery URI for the OAuth2 provider endpoint.		False (unless APP_OAUTH_ENABLED=true)
APP_GENERATE_DOCS	boolean	Generate PyDoc documentation on startup into the pydocs directory	False	False
APP_PASSWORD_RESET_ENABLED	boolean	Enable the password-reset endpoint. Disabling disallows the reset of a forgotten password	True	False
APP_LOG_FILE	string	Set the filename to save the logs to. Leaving this empty won't save logs to a file.		False
APP_MATCHMAKING_LEVEL_RANGE	integer	The difference between 2 player levels to be matched against each other. Setting this to 1 would allow a player of level 5 to be matched against a player of level 6, but not to a level 7.	1	False
CHECK_DATA_OWNERSHIP	boolean	Allow access to another user his data if the calling user is not admin.	True	False

Friends making

This chapter describes the friend request system and general usage in the backend / API. For information about the frontend usage, please refer to the user manual. For specific JSON body information & endpoint usage, please refer to the Swagger documentation.

Friend requests

Friend requests are created by sending a POST request to the `/friend_request` endpoint with the 'target' (other) user id. Friend requests are pending while they exist. The target user can accept or reject the request by sending a PUT request to the `/friend_request` endpoint with the 'status' field set to 'accepted' or 'rejected'. When setting the status to 'accepted', the two users are now friends and can visit the other his island. When setting the status to 'rejected' or deleting the friend request has the same effect. In both cases will the friend request be deleted (as it is no longer pending).

Open friend requests for a certain user can be retrieved by sending a GET request to the `/friend_request/list` endpoint.

Friends

Friend relations are in both ways, thus A -> B and B -> A. This means that if user A is friends with user B, user B is also friends with user A. This design choice (in the db) is made because of its ease of use and decent SQLAlchemy support.

Friends can be retrieved by sending a GET request to the `/player` endpoint. Friends will appear with their user ids in the 'friends' field.

Removing friends can be done by sending a PUT request to the `/player` endpoint with the updated 'friends' field set (thus without the friend to remove).

Task scheduling

Backend

Single Tasks

Task scheduling is how idle aspects that require a single task to be completed are executed. These tasks are scheduled to be completed at a certain deadline. Once the deadline is reached, the task is considered completed. These are so called 'single tasks', for example building a new building or upgrading an existing building. Periodic tasks are handled differently, see below.

Single tasks have each an unique id, but are also always associated with an island (id). This is to ensure that the task is only handled by 'the island that created it'. Single tasks are not required to be bound to a building (for more flexibility), but are most commonly associated with a building.

Real-time tracking of the deadline is done on the frontend of the application, where the user can see (to the second) how long it will take for a task to complete. The backend, however, does not track real-time scheduling. Instead, it 'schedules'/keeps a record of the tasks based on the time they should be completed. When the frontend boots up (the player logs in), then the frontend calculates the time remaining time until completion based on the endtime timestamp. Once the task is completed and the 'on complete' action is performed, the frontend send a `DELETE` to the task endpoint to delete said task.

Creating a new task

A task is created by sending a `POST` request to the `/api/task` endpoint with a predefined deadline. The task will be scheduled to be completed at the deadline. However, the frontend needs to perform a manual check to resync its catalog of pending tasks with the backend. The backend does not notify the frontend of new tasks. This is because the frontend is normally the one creating tasks and should therefore be aware of the new tasks (that are relevant to it) before it's submitted to the backend.

After the task is created, the frontend keeps a real-time timer of the task. Once the task is completed, the frontend will 'notify' the backend that the task has been handled (by deleting the task in the backend).

Resyncing tasks

The frontend should fetch the current pending tasks from the backend every time the user logs in or the page is refreshed. Listing of tasks is done through the `/api/task/list` endpoint. First, all tasks that have been completed since the last logout should be executed. These are retrieved when setting the

`is_over` flag to `true`. Once these tasks have been handled (eg the resources have been added to the island), the tasks should be deleted from the backend.

Finally, real-time timers should be started for all tasks that are not yet completed (`is_over=false`) for the given island id.

Cancelling a task

A task can be cancelled by sending a `DELETE` request to the `/api/task` endpoint with the task id.

Periodic Tasks

Periodic tasks are tasks that are executed on a regular basis. These tasks are not bound to a deadline, and are therefore implemented differently.

These type of tasks are less flexible than single tasks, but are more efficient for tasks that are executed on a regular basis. When a player logs off, the timestamp is logged into the database. When the player logs in, the frontend calculates the time that has passed since the player last logged off. This 'delta time' is then used to reward the player with resources that have been produced in the meantime.

An important limitation of this implementation is that tasks must have a certain reward that is based on the time that has passed since the player last logged off. Periodic tasks that require a more general approach (eg a 'broadcast chat message task' that is executed every 5 minutes) are not supported.

Frontend

Types of tasks

There are five types of tasks that are handled by the frontend:

- Crystal mining task
- Gem generation task (in fusion table)
- Gem generation task (in mine)
- Building task
- Building upgrade task

Let's go through each of these tasks and see how they are handled by the frontend.

High-level overview

- Crystal mining task: The frontend keeps track of the last time the player collected resources from the mine. The frontend calculates the resources that have been produced since the last collection and adds these resources to the player's inventory.
- Gem generation task (in fusion table): The frontend calculates the chance of generating a gem based on the 'fortune' parameter of the fusion table and the number of the put crystals. When a gem is generated, the frontend adds the gem to the player's inventory and sends a .
- Gem generation task (in mine): The frontend calculates the chance of generating a gem based on the 'fortune' parameter of the mine. If a gem is generated, the frontend adds the gem to the player's inventory.

Creating single tasks

When a player performs an action that requires a task to be created, the frontend sends a `POST` request to the `/api/building_upgrade_task` (in the case of building upgrade or placing a new building) or to the `/api/fuse_task` (in the case of a fuse task) endpoint. The frontend should also keep track of the deadline of the task, so it can show the user how long it will take for the task to complete. The frontend contains a timer that is updated in real-time to show the user how long it will take for the task to complete. When the task is completed, frontend completes the task by performing the action that the task was created for (e.g. upgrading a building, fusing crystals).

Real-time tracking

As already mentioned, the frontend keeps track of the time remaining until a task is completed. This is done by calculating the difference between the current time and the deadline timestamp. The frontend then updates the timer, to show for some tasks (e.g. buildings in progress) to the user how long it will take for the task to complete.

Deleting tasks

After each refresh of the page, the frontend should fetch the current pending tasks from the backend. This is done by sending a `GET` request to the `/api/task/list` endpoint. The tasks are filtered based on the island id and their completion status. If the task is completed, the frontend should delete the task from the backend by sending a `DELETE` request to the `/api/task` endpoint.

Handling periodic tasks

The main periodic task where information is exchanged between the frontend and the backend is mining. Each time the player collects resources from a mine, the frontend sends a `POST` request to the `/api/task` endpoint to update last collected timestamp. To calculate the resources that have been produced since the last collection, the frontend calculates the difference between the current time and the last collected timestamp.

Other periodic task (random gem generation in mine) is completely handled by frontend. The chance of generating a gem is calculated on the frontend, ans is effected by "fortune" parameter of the mine. If a gem is generated, the frontend sends a `POST` request to the `/api/gem` endpoint to add the gem to the player's inventory.

Handling single tasks

Single tasks are handled by the frontend in a little bit different way. Each page refresh, the frontend should fetch the current pending tasks from the backend. This is done by sending a `GET` request to the `/api/task/list` endpoint. Then the frontend should start a real-time timer for each task that is not yet completed.

On the hand of the type of the task (building_upgrade, fuse_task), the frontend should perform different actions when the task is completed:

- Building upgrade task: The frontend should show a timer above the building and send a `POST` request to the `/api/placeable` endpoint to upgrade the building after the time comes to zero.

- Building task: Works the same as building upgrade task, but the building is created and upgraded to level 1.
- Fuse task: The frontend should show a timer above the fusion and should send a `POST` request to the `/api/gem` endpoint to fuse the crystals.

Websocket (SocketIO)

Immediate communication is done through Websockets using the SocketIO library. This allows for real-time communication between the client and the server. The server can send messages to the client and the client can send messages to the server. This is useful for real-time updates like chat, but most notably: real-time multiplayer.

SocketIO uses Websockets as a transport layer, but it also has fallbacks to other transports like long-polling, which is useful for older browsers that do not support Websockets.

General

We use SocketIO namespaces for 'grouping' of messages and events. This allows us to have multiple 'channels' of communication. For example, we have a namespace for the game (most notably forwarding for peer-to-peer communication) and a namespace for the chat. This allows us to have different events and messages for each namespace.

On the backend, adding a new namespace is done by adding a new Namespace class in the `socketio` module. This class should inherit from `flask_socketio.Namespace` and should have a `@socketio.on('event')` decorator for each event that it should handle. Register this class in the `socketio` module by importing it in the `__init__.py` file and register it using `socketio.on_namespace()`.

On the frontend, you can connect to a namespace by calling `io('/namespace')`. This will return a socket connection for the given namespace that allow you to send and receive messages from that namespace.

The data field / contents of a message is always a JSON object. A 'session' is a single WebSocket connection between a client and the server. A 'player'/'user' can have multiple sessions (for example, when the player is logged in on multiple devices/tabs).

Chat

The chat is a simple example of a SocketIO namespace. It has a single event: `message` that holds the chat message, a username and the timestamp. Client sends a message to the server using the `message` event with the given parameters and the server will broadcast this message to all connected clients (thus also to the original sender).

Forwarding

Forwarding is a Peer-to-Peer communication system that allows clients to send messages to other clients. This is used for real-time multiplayer and friend visiting where the server should not be involved in the communication between clients. The server will only forward the message to the target client (specified by the sender) and will not store nor intervene in the communication.

A client can forward a message to another client using the `forward` namespace. The sender is required to set a target user id in the `target` field of the message. The server will then forward this message to the target user. The target user will receive the message using the `forwarded` event and the original JSON object. The server will also send the message to other sessions (not the sending session) of the same player to keep the state in sync. A `sender` field is added to the root JSON object with the userid of the original sender of the message.

Example of a message that is forwarded:

```
{
  "target": <target_user_id:int>,
  "message": "Hello, this is a forwarded message!",
  ... other JSON attributes
}
```

Feature overview:

Our purpose for our game was implementing all the necessary requirements according to the assignment while also making the game as enjoyable as possible. This is why besides the idle aspect of the game we also implemented a real-time 1v1 multiplayer mode that is a combination of a 3d person hero shooter and a tower defence game. The tower defence aspect of the game works well with the idle nature of placing buildings, slowly increasing your strength in multiplayer by increasing the amount of buildings that can be placed and how powerful those buildings can be as you level up. The spell casting is a feature that will enhance the enjoyability and immersion of the game by really allowing the player to participate in the game & interacting with the different entities. It also makes the game more competitive since you can try to kill the other player.

Another important part for our game was the progression. We felt it was really important that our players are rewarded while they are playing our game so that they will want to keep coming back. In general we have the following progression scheme:

1. The player spawns on their island for the first time. They can familiarize themselves with the controls and get a first taste of how the resource collection and building placement works. From lvl 1-3 the player will collect crystals from the mine, eat the crystals to gain xp and place buildings to increase their resource generation.
2. Starting from lvl 3 the player will unlock the fusion table, a very important building that is necessary for starting multiplayer. After building the fusion table the player can start creating gems or if they're lucky get one from a mine. These gems can be put up as stakes to start a multiplayer match.
3. The player will get their first taste of multiplayer. Depending on their progression they could have a hard time if their opponent has placed more buildings (especially towers and warrior huts). To mitigate unbalanced multiplayer matches we introduced a cap on how many buildings of a certain type can be present on your island as well as with which players you can match during matchmaking: the max building cap is determined by your lvl and a player can match with another player that has a difference of max 1 lvl. At lvl 3 though the player will not have unlocked towers or warrior huts yet and the game basically comes down to which player can kill the other's altar first with their fireball spell.

4. As the player progresses and increases their lvl, they will unlock more buildings that can both increase their power during multiplayer as well as increase their resource regeneration. Currently this is our endgame content where players mainly increase their xp by playing multiplayer matches (winning gives double xp, killing players & minions also gives xp).

During this progression the player will have periods of "downtime" where they are waiting for a gem or crystals to be produced or they're waiting for a building to be completed.

Frontend

Program Design

We have tried to maximize the extendability & efficiency of our code although this was not always an easy feat when working together with multiple people.

For our front-end we decided to keep things simple and (mostly) stick to vanilla javaScript. The main reason was to increase our initial development speed by not needing to go through the hassle of learning both javaScript itself and a new framework like React, Angular, etc. The exception being the use of Three.js for our rendering, socketIO for managing state during multiplayer (and real time chat) and some other small libraries/modules. We think this was also the better choice from a learning standpoint since most of us have never developed a web application and creating a good foundation and understanding of the basics is never a bad thing.

At the start of our development we were thinking about also using a physics engine for our collision detection and movement but decided against it since we thought it would unnecessarily complicate things and also the player movement that we got with a physics engine was not really "snappy" like we wanted to. Looking back on it, it might have been a better choice to use one since, during development, we had to implement a lot of physics on our own, which did not always yield the greatest results.

While we are talking about physics one of the great optimisations in our game is the use of an axis aligned bounding box volume hierarchy for our collision detection. We used a library for this and this allowed us to greatly optimize the collision detection. The way it works is that when we update our static geometry (the island + the buildings placed). We run all this geometry through a collider generator that will calculate a bounding box for every face in the geometry and also calculate bounding boxes that encapsulate these smaller ones recursively until we get one singular bounding box that encapsulates the entire geometry. We can then traverse this tree of bounding boxes to efficiently determine where an entity collides with the static geometry. Note that this is only done for the static geometry. For collision detection between 2 moving entities we just use a boxCollision function from threejs. One issue we encountered with our implementation is that for high velocity and/or low framerate our entities/static geometry could phase through each other since we did not have swept AABB (continuous) collision detection but just regular AABB (discrete) collision detection. We considered a 2 different solutions:

1. Implementing swept AABB: this would allow us to calculate all collisions by determining a collision at each frame just before it happened (use velocity and deltaTime to check whether the collision will happen on the next frame) however this proved hard to combine with the BVH library which we were not willing to discard since it was a drastic optimisation

2. Separating the control update loop from our physics updates: We decided to adjust the way our game update works. Now every frame we do the basic updates (updating animated views, checking player input, etc) and seperately we do a physics update multiple times where newDeltaTime = deltaTime/NrOfPhysicsSteps. This is an easy way to mitigate the phasing issue (at the cost of performance) while not having to drastically change our codebase. This did improve our situation but we still had to implement some tricks to totally remove phasing and clipping through the static geometry.

Another great optimisation was the use of and instanced mesh with our timers (for the buildings). When we first implemented our fancy rotating 3d timers, we quickly realized we had to optimise this since the framerate would drop in half for every timer that was in our game. We did some research and decided on an instancedMesh implementation. An instanced mesh leverages the power of GPU instancing by re-using the same mesh and rendering it in 1 draw call (with different transformations for every instance). We now have an instancedMesh (using threejs InstancedMesh class) for every number that is generated at game load and is rendered when necessary. A downside of this implementation is that we need to have a static amount of instances for every mesh and if we want to change the amount of instances we would need to create a new InstancedMesh. However this is a non-issue since our game progression limits the amount of buildings that can be built at the same time to two (at lvl 5). The amount of instances reserved on game load (as with most of our "magic values") is configurable in the files contained within the configs folder. This optimisation drastically improved our framerate and we can now have multiple timers without seeing any notable framerate drop.

The general lay-out of our front-end tries to take advantage of a MVC-model with subject-observer (models & views) and factory (Factory & SpellFactory) patterns while also making use of interfaces (Placeable & Entity) and composition (ConcreteSpell) to make our code easily extendible. We also tried to make our code event-driven by utilising javaScript's asynchronous & events capabilities. During development however, our app object (the main entrance object to our code which is responsible for updating/controlling the game) became quite the god object. We could improve on this by better encapsulating code in our different classes but had to leave this optimisation behind due to prioritisation of other features. We have tried to take into account the single responsibility principle as much as possible, which is why the front-end has around 80 different classes.

For the multiplayer itself we use the socketIO library which integrates nicely both on the front-end and the back-end with the flask-socketIO extension. We use websockets for the real time chat, checking if a player is online in the friend menu and syncing state between peers during a multiplayer match. One of the design choices we made was keeping as much of the multiplayer game as possible on the front-end to reduce the load on our server since first and foremost this would reduce costs for upkeep of multiplayer services and second because most if not all of the game itself was run on the front-end already. During a multiplayer match a player is responsible for sending state updates of the player itself and also for their minions every 16ms (corresponds to 60 fps, we have not implemented a dynamic update interval). A player is also responsible for updating the health of the opponent's entities as to reduce delay between the player seeing their spells hit the entity and seeing the health being updated. Both the animation state (i.e. which animation is playing for that 3d model) and the health updates of the player and the minions is sent separately because generally they do not change every frame. The back-end receives these update events (and other control events which the server handles itself) and forwards these to the other player (peer to peer). The back-end is the final authority on the results of the match (who won/lost, which stakes are assigned to who, when the match is stopped) and this is why we need to keep state for the multiplayer matches (this is also the only case were the server is stateful).

The communication between the players and the server for multiplayer follows a certain protocol:

1. The player tell the server they want to start matchmaking: On the front-end players can start matchmaking from the altar menu. The will player will only be able to activate matchmaking if he has put up enough stakes, there is a path from the connection point to the altar that minions can follow & the connection point of the island (the point where the bridge will attach to) is not occupied by a building.
2. When the player presses the matchmake button, he will be added to the matchmaking_queue table in our database. When another player of similar lvl (difference of max 1 lvl) enters the queue, the back-end will emit a "match_found" signal to the players.
3. On receival of the "match_found" signal, both players will start retrieving the info of their opponent and start setting up the multiplayer match.
4. When a player has finished loading everything a "player_ready" signal is emitted to the back-end where the server will keep state of the match (players & match timer) & if both players have sent the signal, the server will fire the "match_start" signal to tell the players they can start playing.
5. During the match state updates are exchanged as described earlier and the match can finish in 4 different ways:
    1. The server ends the match because the timer reached 0, The result will be a draw.
    2. The websocket on the server detects a player disconnected and gives an automatic win to the other player.
    3. One of the player presses the leave match button in the settings menu & fires the "player_leaving" signal. The server also gives an automatic win to the other player in this case.
    4. One of the players destroyed the opponent's altar and fired the "altar_destroyed" signal.
6. All different cases of ending a match are handled by the server which will redistribute staked gems based on who won & fire the "match_end" signal which the players use as a signal to reset their game state to singleplayer.

The other form of communication between the front-end and back-end is via our REST API. This is mainly used for updating our database but also for authorisation purposes and getting the server time (which is needed for calculating resource production). For our idle implementation we again try to do as much as possible on the front-end which is why the back-end keeps time stamps for when certain tasks started or need to be completed and the front-end handles the calculation part and when to remove/create those tasks.

Combat System:

Here, we will try to explain some of the decisions and techniques used in regard to the combat system. First of all, at the start of multiplayer, for each altar and tower present on the islands of both players, an extra entity gets created at the same position as the building it is made for, called a proxyEntity. This proxy entity has a view and a model, as expected. The view consists of a bounding box for easy collision detection, and a visual health bar that changes color and percentage filled based on the health of its model. The model has functions for taking damage and for death. By using these proxys, we make

the single player less messy. For all towers, a spellspawner gets created too. These spellspawners fire a spell (given in the parameters of the spawner) at a certain interval with a certain amount of damage, dedicated by the stats of its corresponding tower. These spawned spells then function the same as spells cast by a player. Dealing damage is done fairly simple. Every entity that can take damage has a `takeDamage(damage)` function. These are spells, proxy, and characters. This function updates the health according to the given damage, and also checks if death of the entity needs to be handled. The takeDamage function is always called by the damage dealer. For spells this is done by collision detection, and when collision occurs with a target, it will call the takeDamage function on the target, if dealing damage is one of the effects of the spell. Minions on the other hand will have a target they are locked on, which can be a building, an altar or a shield spell, determined by the `getClosestEnemy(character, targets[])` function. When they are close enough to their target, they will go in the attack state and periodically call the takeDamage() function on their target. As mentioned before, the takeDamage() function checks if the receiver has reached 0 or less hp. If this is the case, the `dies()` function will be called. For towers, this makes the spellspawner stop shooting, minions will be deleted, characters will get an overlay with a respawn timer after which they can respawn, and an altar dying will result in the end of the match and a victory for the destroyer. Finally, we will quickly go over the spell effects. First, we have the fireball. This shoots a projectile that checks for collision, and will be deleted on the first enemy entity hit. If that enemy is targetable, it wil deal damage using the previously described method. The thundercloud spell is a bit different, as it doesn't get removed after the first collision. Instead, it stays on the field for a certain amount of time, and deals damage to ALL enemies within its bounding box every x amount of time. The shield spell will form a bounding box around the player and thus make sure all attacks that would hit the player, hit the shield. While the shield is active, the player will have a boolean `shielded = true` and therefore won't take damage when the takeDamge() function is called on the player. Instead, the takeDamage() will be called on the shield. The takeDamage() function of shield will decrease the amount of shields there are (visually through a listener and in the model with a shield variable). Once this amount reaches zero, the shield is deleted and the shielded variable on the character is removed. Through using this implementation, we basically make sure the next three damaging effects on the player deal no damage. The last spell is the icewall, which just creates an entity with a bounding box, so it can block enemy spells and paths.

HUD

The HUD (Heads Up Display) is a graphical user interface that displays information to the player. The HUD is displayed on the screen and is used to show the player's health, manna, and other important information. The HUD is an essential part of the game and helps the player keep track of their progress and status. The html part of the hud is located in the `index.html` file, and the css is located in the `index.css` file. In practice, the HUD is basically and overlay of different elements that are displayed on the screen over the 3D world. When the players pointer is not locked or hovering over a button, the mouse will have the appearance of a magic wand. For a higher level overview of the HUD and its utility, please refer to the user manual. In this document, we will focus more on the technical implementation of the HUD. The HUD uses a lot of images, which are all credited in the credits file. We will not go into detail about the different elements of the HUD.

Hotbar

The hotbar is made up of different flexbox items that are displayed horizontally, and has different layers. The lowest layer are the empty boxes, which are always displayed. These boxes are all images, and these images are changed using javascript to display a golden variant when that box is selected, or back to the default when switched to a different slot. The second layer are the spells equipped by the player. These are displayed as images, and are also changed using javascript to display the equipped spell in the correct slot. The equipped spells and position in the hotbar are persistent and are stored in the database, by using the 'player spells' table. The third layer are the spell cooldowns. These are normally not displayed, but when a spell is cast, the cooldown of the cast spell is displayed in the correct slot by a decreasing number, and by making the lower two layers have a higher opacity. These cooldowns are updated every delta time, and are hidden when the cooldown is over.

Health and Mana bars

The health and mana of the player are displayed by the red and blue potions respectively, displayed to the left and right of the hotbar. On these potions, the current and maximum health/mana are displayed. The potions themselves are images, and have two layers. The bottom layer is the potion in a gray colour, and the top layer is the potion in colour. The top layer is resized based on the current health/mana, and the bottom layer is always displayed. This way, the potion is filled with the current percentage of health/mana. The health and mana numbers and the percentage of the potion that is full are both updated using event listeners that listen when the health/mana of the player is updated.

Friends menu

Please refer to the **friends documentation** for more information about the friends menu.

Level, experience and player name

The level, experience and player name are displayed at the top of the screen. The bar shows the percentage of xp you have towards the next level. The level and experience are updated when the player gains experience. When this happens, a little animation is played where the bar fills up with the new experience. This is done using a combination of css and javascript. Below that, the precise amount of current xp and the xp needed for the next level are displayed. The player name is displayed above the level bar. This name is the same as the username of the player, and is retrieved from the database when the player logs in. The players experience is stored in the database, and retrieved when the player logs in. The level is calculated based on the experience by the front end, and therefore is not stored in the database.

Settings menu

The settings menu is displayed when the player clicks on the settings icon in the top left corner of the screen. The setting icon changes when the player hovers over it, and when the player clicks on it. This is done by using different images for the `:hover` and `:active` states of the icon. In the settings menu the player can change the horizontal and vertical sensitivity of the camera, which changes a variable in the config file so the camera moves faster or slower when moving you mouse. They can also the change the performance of the game in case they have a slower computer. There are three options: low, medium

and high. On low performance, the game doesn't render shadows or grass. On medium performance, the game renders grass but not shadows. On high performance, the game renders both shadows and grass. The cursor that helps the player aim can also be switched to different types of cursors. Based on the selected cursor, a different image will be displayed at the middle of the screen. The biggest part of the settings is changing the keybindings. The player can change the keybindings of the spell slots, the movement keys, eating, etc. When a player changes a keybinding and clicks apply, that keybind is changed in the keybinds config file, and therefore impacts the game. The keys work with codes and values. The code is the relative position of the key on the keyboard, and thus is independent of azerty or qwerty. The value what is on the key that is presses. If I press the first letter key on my keyboard, the code will always be KeyQ, but the value will be Q on a qwerty keyboard and A on an azerty keyboard. We mostly use the key codes, that way the default keybinds are independent of the keyboard layout and are therefore always intuitive. All these settings are gathered in a JSON object and sent to the backend at the endpoint `/api/settings` using a `PUT` request when the player clicks apply. Both the code and the value of the keybinds are stored in the database. The settings are retrieved from the database using a `GET` request when the player logs in, then visually put in the settings menu (this is where the values of the keys get used) and the menu then does an apply of the settings. This uses the same function that is called when the player presses apply, except with an extra parameter that blocks the `PUT` request and prevents unnecessary requests to the backend. This way when a player logs in, the settings the user are instantly applied. At the bottom of the settings menu, there are some buttons with different options. The first button is the 'reload world' button. When the player clicks this button, the world is reloaded. This is useful when the player is stuck in a wall for example. The second button is the 'leave match' button. Depending on the situation of the player, this button will either take the player out of a match, out of friend visit mode, or just out of the menu. The third button is the 'logout' button. When the player clicks this button, the player is logged out and redirected to the landing page. This is done by redirecting the player to the `/logout` endpoint. The fourth button is the 'fullscreen' button. When the player clicks this button, the game goes fullscreen, as it would when you press F11. This is simply done by calling the `requestFullscreen()` function. There is also the help button, which opens a help menu. This menu displays aims to help the player with the game, and contains information about the game, the controls, the spells, etc. This menu also features images to make it more visually appealing and clear, as wel as an image with the default layout of the keybinds marked on a keyboard. This menu was partially made by converting the `USER_MANUAL.md` file to html, and then adding some extra styling and changing the text a bit. It also includes an overview of everything you unlock and at which level, which gets dynamically made based on the config file, so it doesn't need to be changed when the levels get balanced differently. The last button is the 'delete account' button. When the player clicks this button, the player is asked to confirm the deletion of the account. When the player confirms, the account is permanently deleted from the database. This is done by sending a `DELETE` request to the `/api/user_profile` endpoint. The player is then redirected to the landing page.

User manual

This chapter is intended to provide a high-level overview of the project, with some technical details. **Note: GIFs in this chapter do not work in the PDF version. Images are also misplaced. Please see the original User manual here**

Historical setting

The player will be an “Ordinary” wizard. Someone without the inherent ability to cast magic. However, through a certain event the player is transported to a magical island where, on arrival, they discover crystals that grant magical abilities when eaten. Excited by their newfound power the player will want to grow their power by mining for those crystals.

Goal of the game

The goal of the game is to mine for crystals and grow your power. The player will be able to use different spells and gems to improve the speed of mining and the amount of crystals mined. Besides mining, the player will also be able to explore the islands of other players and fight them in a combat system.

Landing page

The landing page is the first page the player will see when they visit the website. The page is optimized for mobile and desktop users. It contains the name of the game and buttons to register or login. Also, desktop users get a magic wand as a cursor.

Register page

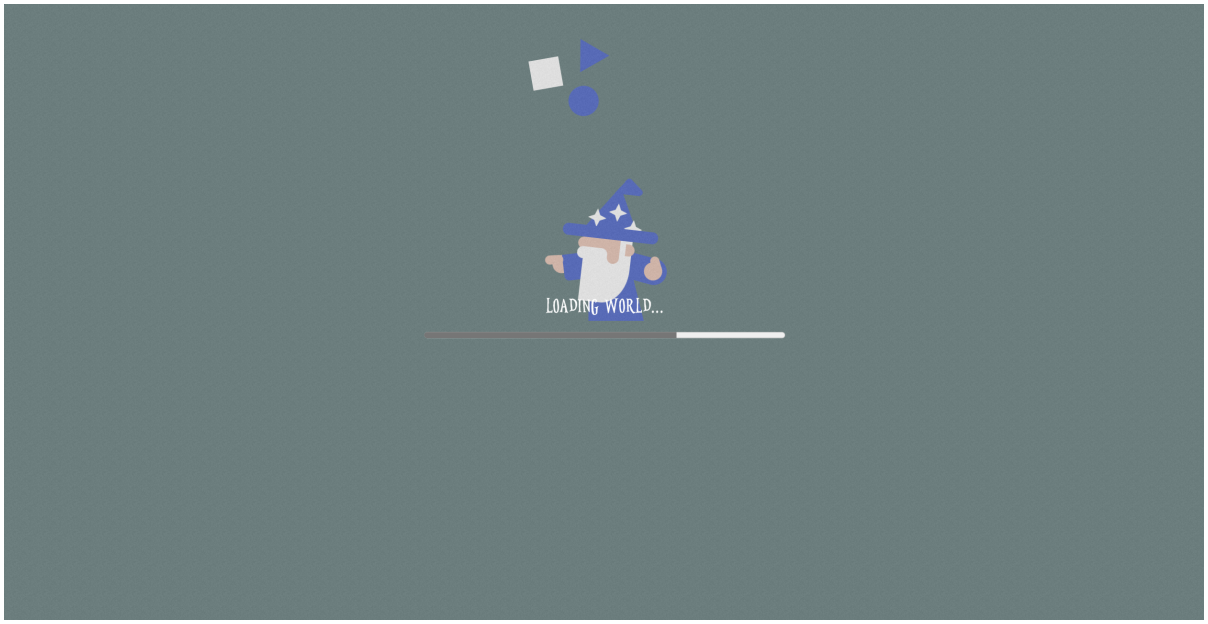
Upon arrival of the landing page, the player has the option to create a new island, this will redirect the player to the register page. Here, the player can create a new account and get a new island by filling in the following fields:

- Username
- Password
- First name
- Last name

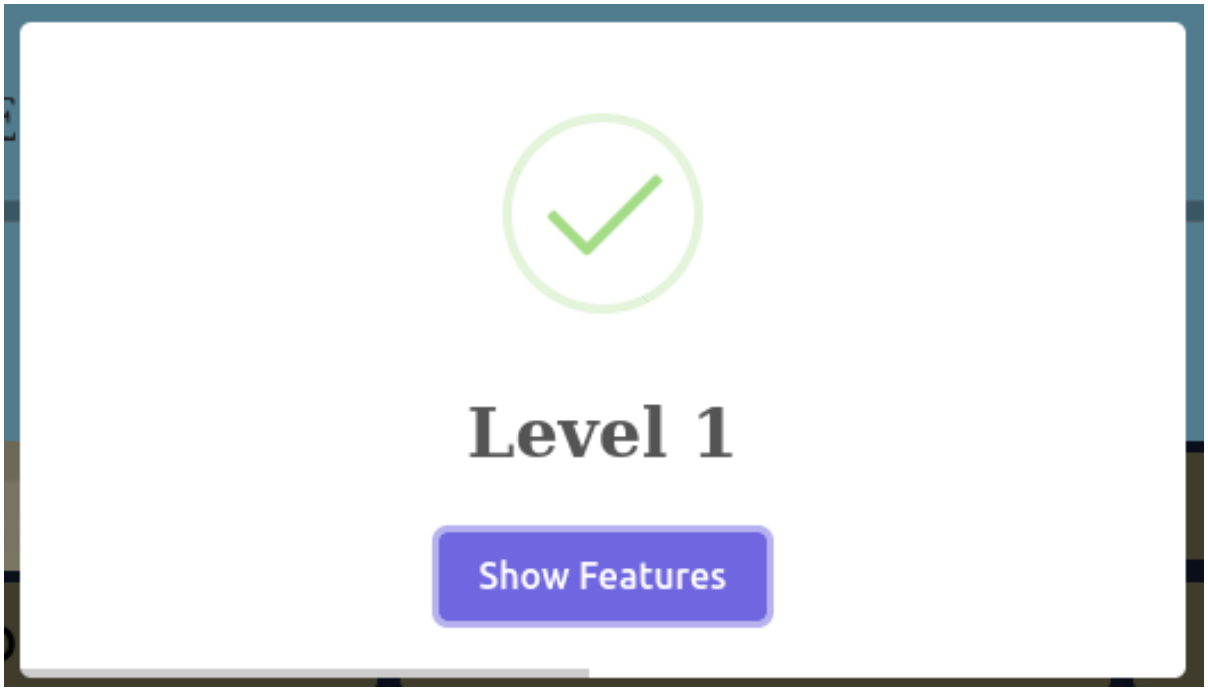
Login page

If the player already has an account, they can log in by clicking “I have already eaten some crystals” which refers to the crystals that grant magical abilities when eaten which the player already has if they have an island. For the login the player needs to fill in their username and password. There is also an option to log in with Google and reset the password.

Index page



Level popup: After the player has gained some experience playing the game (e.g. for eating crystal or placing buildings), the player can level up. When the level increases the player gets a popup message saying “Level {currentLevel}”. If the player clicks on “Show Features”, the player can view the features of current Level.



After logging in, the player will be redirected to the index page. First at all, the loading screen will appear, after which the player will be redirected to the game. The loading screen consists of a progress bar showing the amount of the game that is loaded, some text displaying what part of the game is being loaded, and an animation of a wizard casting a spell to keep it visually interesting.

The index page is the main page of the game. It contains the following elements:

- Settings button: opens settings menu
- Chat button: opens chat menu
- Crystal counter: shows the amount of crystals the player has
- Health bar (red potion): shows the health of the player
- Mana bar (blue potion): shows the mana of the player
- Inventory: shows the equipped spells
- Username: shows the username of the player
- Current level: shows the current level and the progress bar of the player
- Game screen: shows the game

Settings menu

By clicking the top left icon, the player can open the settings menu. Here, the player can change some basic settings about the game, such as audio, graphics and controls. He can also log out from the game and delete his account from here.

Chat

By clicking on the bottom left icon, the player can open the chat. The chat is a global chat where every player can chat with each other. The chat is implemented with Socket.IO library.

Cheats

If the player has the right to use cheats, the player can open the chat menu and type the following commands:

- `\level(level)`: set the level of the player
- `\xp(xp)`: increase the xp of the player
- `\crystal(crystals)`: increase the crystals of the player

Inventory

At the bottom of the screen are 5 boxes, that all provide a space for an equipped spell. The first spell can't be changed and will always be the build spell, with which the player can build buildings on his island. The other 4 equipped spells can be changed in the altar menu, and will be displayed in the inventory hot-bar. The player always has one spell selected, which is shown by gold highlighting of the selected spell slot. Each individual spell has a unique icon, which is then shown on its equipped spot on the hot-bar. After using a spell, that spell becomes transparant for the cooldown period, and a timer is shown on the spell icon, displaying the exact remaining cooldown time. The spell can't be used for that period of time.

Game

Basic player movements

The player can move around the game using the following keys (can be changes in the settings menu):

- W: move forward
- A: move left
- S: move backward (rolling back)
- D: move right
- F: Toggle fps counter
- Space: jump
- Shift: sprint
- C: toggle chat
- 1-5: select a spell slot
- E: interact with objects
- Mouse: look around
- Left mouse button: cast spell
- Q: eat

Collision detection

The player can't walk through buildings or other objects. Collision detection in the game is always done via basic axis aligned bounding box intersection tests. However for the static geometry (= island + buildings) there is an extra optimisation:

We use a library to add a bounding volume hierarchy (=bvh) which allows us to optimally test collision between the entities in our game and the individual polygons in our static geometry. This way we can let our entities move smoothly over every terrain in our game.

Spell to entity, spell to spell and entity to entity collision does not make use of this optimisation and may appear more "clunky" (for example player movement when moving along an icewall spell).

Multiplayer

When a multiplayer match is started, the islands of both players will appear next to each other, connected by a bridge. The goal is to destroy the other player's altar. By doing so, the player will win the match and gain the stakes of the other player and get his own stakes back. The player can also lose the match, in which case the player will lose his stakes. If needed, the player can surrender the match by pressing the leave match button in the settings menu. On top of the screen, the player can see a timer. If this timer ends, the match will automatically end and result in a draw. The player can fight by using his spells, and is helped by his towers, which will shoot fireballs at the enemy. The player can get helped by his minions too, which will move towards the enemy's altar and attacks enemies along the way. These minions spawn from warrior huts on the player's island. By eating crystals the player can regain health and mana during the battle Health bars above allys and enemies show the current health of the entity.

In-game menus

The buildings can be accessed through menus. Here you can interact with the buildings and use their functionalities and upgrade them. Most buildings can be upgraded by using gems and enhancing their abilities. Opening a menu is done by default by pressing the `E` key when the player is aiming at the building. In each menu, you have a possibility to delete the building by pressing on the bin.

Altar menu

In the altar menu the player can configure their inventory and equip spells. They can also select stakes to start a multiplayer battle.

Mine menu

The mine menu passively generates crystals for the player and also has a small chance of generating gems.

You can collect the crystals by clicking on the collect button. The number of crystals will be added to the player's inventory.

Fusion table menu

In the fusion table the player can use crystals and fuse them into gems. This will take some time. The player can use gems to upgrade buildings and also to participate in multiplayer battles.

Tower menu

The tower is used for combat and displays its health and damage.

Eating

By eating, the player will consume crystals in exchange for mana, health and experience.

Spells

Fireball: creates a fireball projectile that will damage enemies on hit.

Thundercloud: creates a thundercloud above the position the player is looking at, doing area of effect damage for some time.

Shield: creates 3 rotating shields around the player that will block the next 3 incoming damages.

Icewall: conjures an ice wall out of the ground, blocking entities movement.

Each spell has own cooldown period and mana cost. The player can cast a spell by pressing the left mouse button.

The first spell is the build spell, which is used to build buildings on the island.

Building system

BuildSpell is used to build and move buildings. Will always be present in slot 1 in the hotbar.

You can fully customize your island by building different buildings. Each building has its own functionality. To place a building, the player have to select the build spell in the inventory and click on the desired location. Then the build menu will appear, where the player can select the building he/she wants to place.

The build menu has three tabs: - Combat: buildings that are used for combat (e.g. tower) - Resources: buildings that are used to generate resources (e.g. mine) - Decorations: buildings that are used for decoration (e.g. tree) In each item cell you can see the required number of crystals and time in seconds

to build the building. Also, there is an icon and the description of the building. After you select the building you want to place, the build menu will disappear, the number of crystals will decrease and the building preview will be placed on the selected location.

Above the building preview you can see the timer that shows the remaining time to build the building. When the timer reaches 0, the building will be placed on the selected location.

After that, you can interact with the building by pressing the `E` key.

If you want to move the building, you have to select the build spell in the inventory and click on the building. Then the building will be selected and you can move it to the desired location. To place the selected object on the new place, click on the desired cell. If you want to rotate the building, you have to click with the right mouse button.

Building upgrade

This upgrade submenu shows the player the current level of the building and the cost and time to upgrade it to the next level.

The game supports two types of building upgrades: - Building upgrade: the player can upgrade the

building to the next level - Gem upgrade: the player can upgrade the building with gems by putting them in the gem slots

The first one is done by clicking on the upgrade button. The second one is done by clicking on the gem slot and selecting the gem you want to put in the slot. Building upgrades provide the player more gem slots and in some cases improves stats of the building.

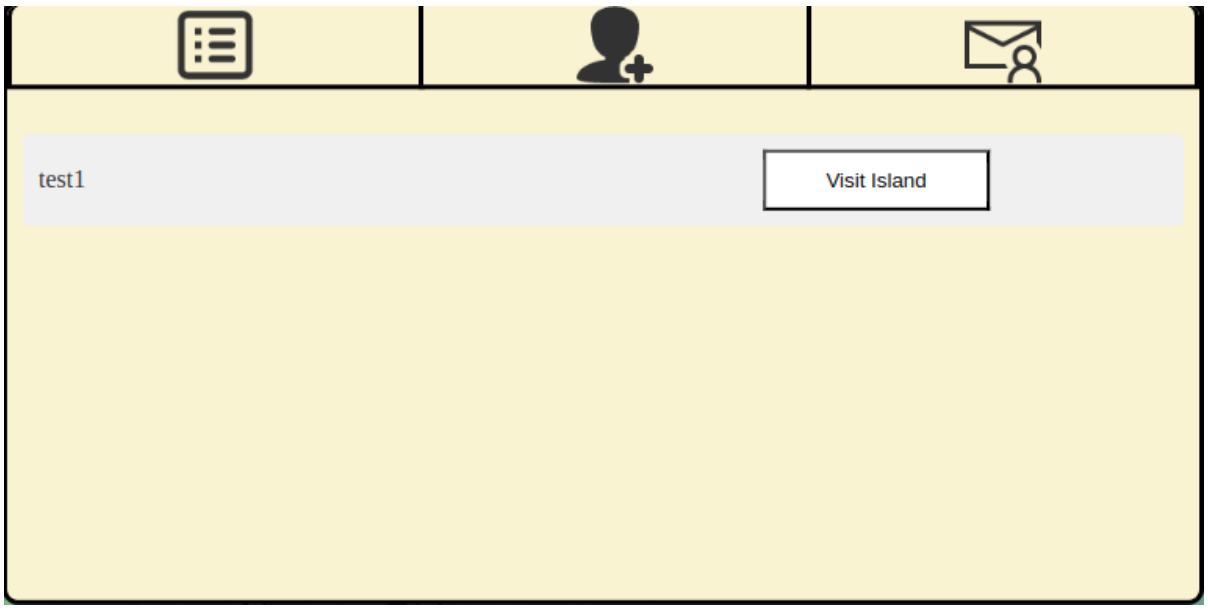
Currency

The general currency in the game are the crystals. These can be mined by the player and are used for most actions in the game. There are also different gems with attributes that can be used to upgrade buildings.

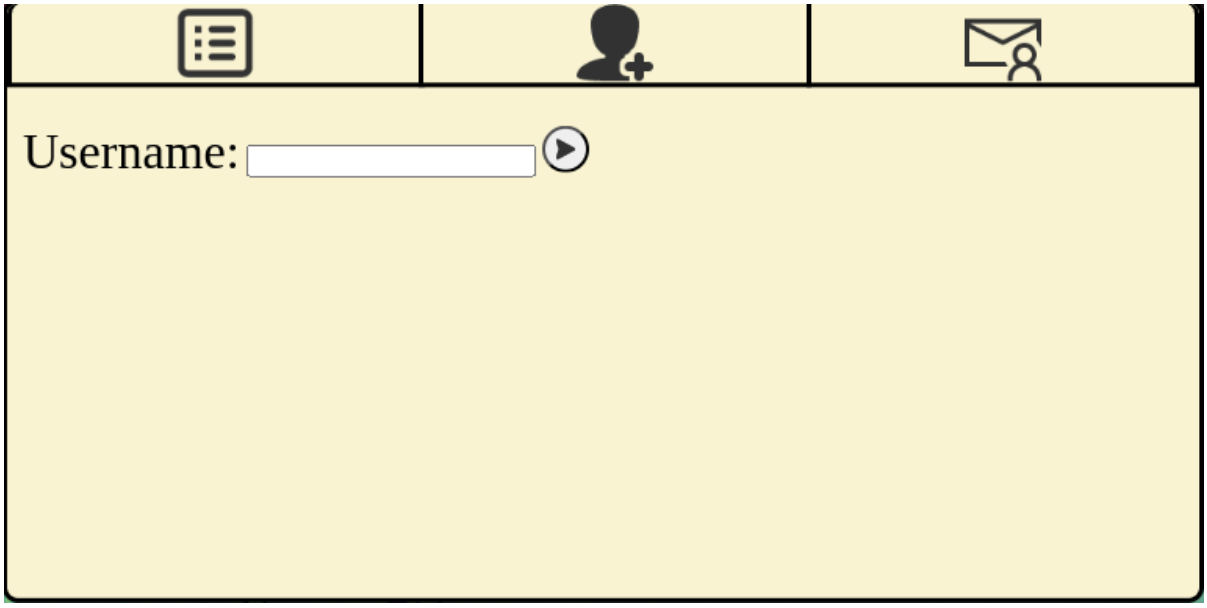
Friends

In the game, you can access the Friends Menu through the button located at the bottom right of the screen.

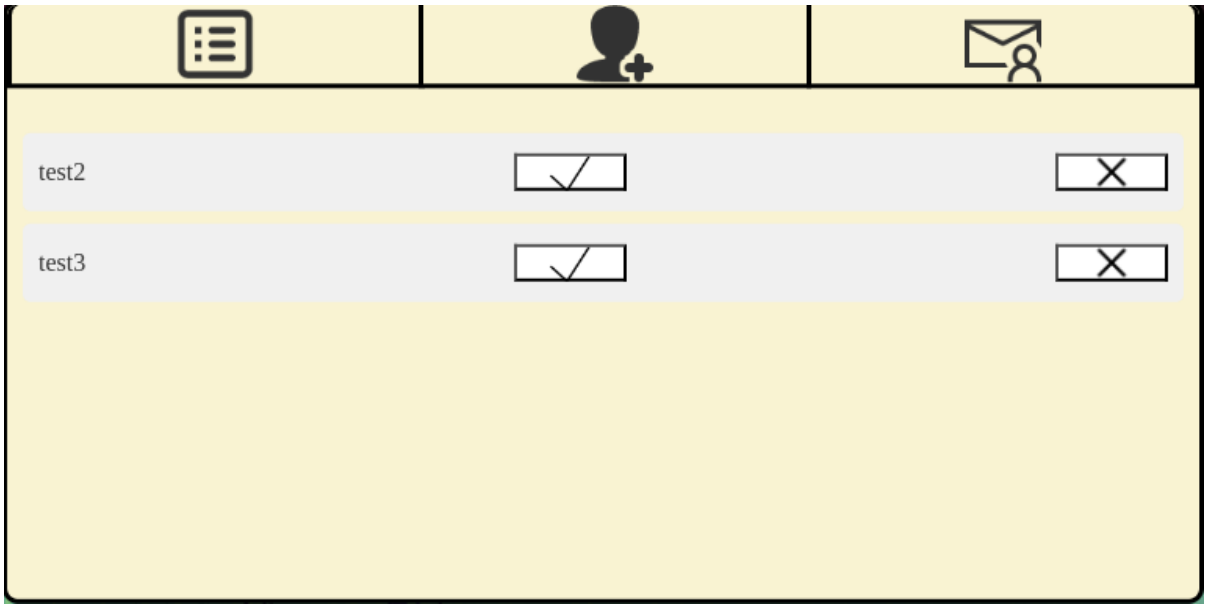
Click on the first icon to view your Friends List. Here, you can see the list of your friends. There is also a “Visit Island” button next to each friend’s name. Pressing this button will send a visit request to your friend. If your friend accepts the request, you will be able to visit their island.



Click on the second icon to open the Add Friend forum. Here, you can enter the username of the player you wish to add as a friend.

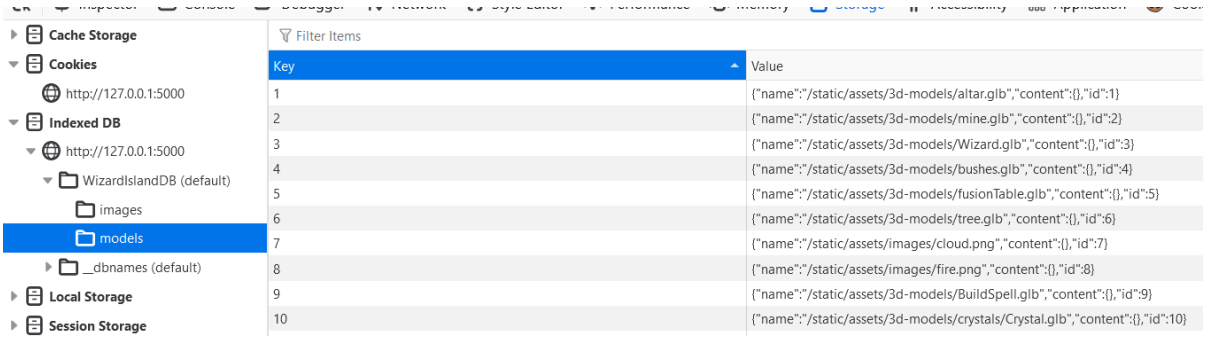


Click on the third icon to view the list of friend requests and “Visit Island” requests you have received. From here, you can accept or decline these requests.



Optimizations

Asset caching



When player logs in for the first time certain 3D-models and images are stored in cache. This makes the process of loading assets faster because the player does not have to download all assets again. To store the models we use IndexedDB.

Tests

In this document we will give an overview of the tests that we have written for our project. We will explain how they are structured and will give a checklist to check them all.

Test structure

Each manual test has the following attributes:

- **Test Case Title**: A short description of the test case.
- **Test Case ID**: A unique identifier for the test case.
- **Test Case Description**: A brief description of the test case.
- **Preconditions**: The conditions that must be met before the test can be executed.
- **Test Steps**: The steps that must be followed to execute the test.
- **Expected Results**: The expected outcome of the test.
- **Postconditions**: The conditions that must be met after the test has been executed.
- **Test Data**: The data that is used in the test.
- **Notes**: Additional notes about the test.

Test checklist

Please refer to the code files themselves at `/test-cases/tests` for the test contents.

- REG-001: Registration Form Submission
- CHAT-002: Chat Menu Test
- CTRL-003: Game Controls Test
- ISL-BUILD-004: Island Building Functionality Test

- ISL-BUILD-005: Building Placement and Interaction Test
- MINE-006: Mining Resources Test
- CACHE-007: Asset Caching on First Loading Test
- LEVEL-008: Level Up Popup Display Test
- LOAD-009: Loading Screen and Redirection Test
- CHEAT-010: Cheat Commands Functionality Test
- RESET-011: Password Reset Functionality Test
- SPELL-012: Spell Casting Functionality Test

Database Migrations

Alembic

Database schema migrations are handled by flask-migrate which uses Alembic as its backend. Alembic is a lightweight database migration tool for SQLAlchemy.

All migrations (up & downgrade) are found in the `/migrations/versions` directory. Each migration file is named with a timestamp and a description of the migration.

You can create one yourself by modifying the SQLAlchemy model structure (thus editing the source code) and running the following command with `APP_AUTOMIGRATE=false` and `APP_DISABLE_SCHEMA_VALIDATION=true`:

```
flask db migrate -m "migration description"
```

This will create a new migration file in the `/migrations/versions` directory. You can then apply this migration to the database by running:

```
flask db upgrade [revision_id]
```

You can check the wether the current database is up-to-date with the source code by running:

```
flask db check
```

You can see the current migration history by running:

```
flask db history
```

If you want to downgrade a migration, you can run:

```
flask db downgrade [revision_id]
```

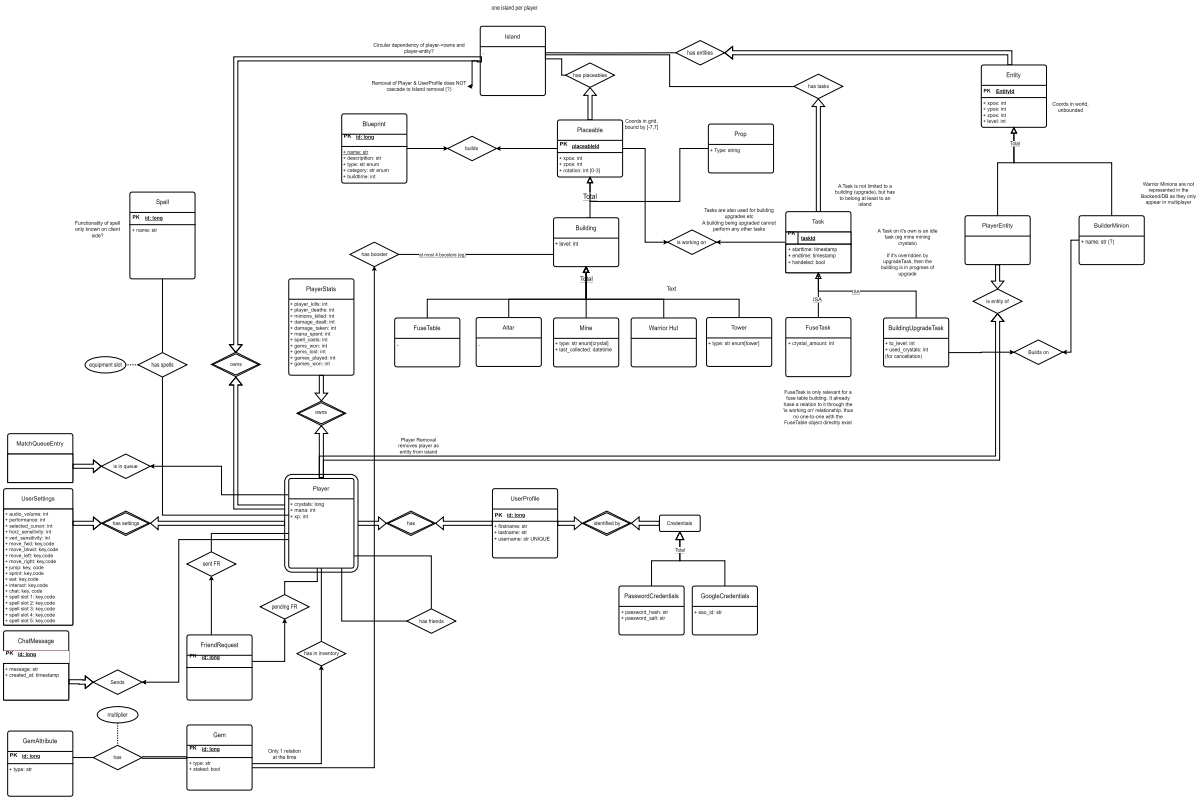
For a full list of commands, see the Flask-Migrate documentation.

Setting up a new database

Follow the steps in the README.md to setup a new database. After the database is created and SQLAlchemy is able to properly connect, you can run the following command to create the final database schema iteravely:

```
flask db upgrade
```

Entity Relationship Diagram



See bigger version **here**.

Class diagrams

Frontend

See bigger version **here**.

Backend

See bigger version **here**.