San Francisco State University
SW Engineering CSC 648/848
Milestone 2

October 16, 2024

Section 04 — Team 02

Members

Katy Lam	**Team Lead**
Arjun Singh Gill	**Back-end**
Matthew Aaron Weesner	**Back-end**
Niko Galedo	**Front-end**
Kevin Lam	**Front-end**
Kullathon “Mos” Sitthisarnwattanachai	**Git Master**
Arizza Cristobal	**Scrum Master**

Revision History

Version	Date	Summary
1.0	2024-10-16	First version submitted for instructor approval
0.2	2024-10-16	Update formatting and document style guide
0.1	2024-10-09	Initial draft proposed to Team Lead

Data Definitions

The following section outlines key data definitions for our game, focusing on high-level descriptions. As the project evolves, more detailed definitions will be added in future milestones.

User Entity

id - Integer. A unique identifier (primary key) for each user account.

FirstName - String. User's first name.

LastName - String. User's last name.

username - String. The user's chosen username for logging in and in-game identification. Must be unique across all users.

hashedPassword - String. The user's password after it has been securely hashed using a. Used for authentication purposes. The original plaintext password is not stored for security reasons and will need to go through for protection. Low level to start.

email - String. The user's email address. Used for account verification, password recovery, and communication. Must be unique and follow standard email format.

resources - List *of UserResource*. A collection representing all the resources the user currently owns. Establishes a one-to-many relationship with the **UserResource** entity.

items - List *of UserItem*. A collection representing all the items (Miners, Constructors etc.) the user owns. Establishes a one-to-many relationship with the **UserItem** entity.

Usage

Stores information about registered user accounts.

ResourceType Entity

id - Integer. A unique identifier for each resource type. (primary key)

name - String. The name of the resource (Iron Ore, Copper Ingot, Nails etc.). Used for display and identification purposes.

description - String. A textual description providing details about the resource, such as its uses or how to obtain it.

icon - String. A URL or file path pointing to an icon image representing the resource in the game's user interface.

userResources - List *of UserResource*. A collection of **UserResource** entries associated with this resource type.

recipeInputs - List *of RecipeInput*. Recipes where this resource type is an input.

recipeOutputs - List *of RecipeOutput*. Recipes where this resource type is an output.

itemBuildCosts - List *of ItemBuildCost*. Items that require this resource type for construction.

Usage

Represents different types of resources available in the game.

UserResource Entity

id - Integer. A unique identifier for each user-resource pair. (primary key)

userId - Integer. A foreign key referencing the **User** entity. Indicates which user owns this resource quantity.

resourceTypeId - Integer. A foreign key referencing the **ResourceType** entity. Specifies the type of resource.

quantity - Integer. The current amount of this resource the user has. Must be zero or a positive integer.

user - User. The user who owns this resource.

resourceType - ResourceType. The type of resource this entry represents.

Usage

Represents the quantity of each resource a user owns.

Recipe Entity

id - Integer. A unique identifier (primary key) for each recipe.

name - String. The name of the recipe (e.g., "Smelting Iron Ore"). Used for display purposes.

inputs - List *of RecipeInput*. A collection of resources required as inputs for this recipe.

outputs - List *of RecipeOutput*. A collection of resources produced as outputs from this recipe.

timeRequired - Integer. The time in seconds needed to complete the recipe. Production time is a game mechanic.

requires - List *of ItemType*. Items or machines required to execute the recipe (for example, a Constructor). Specifies any prereqs for using the recipe.

Usage

Defines how resources can be transformed into other resources.

RecipeInput Entity

id - Integer. A unique identifier for each recipe input. (primary key)

recipeId - Integer. A foreign key referencing the **Recipe** entity. Indicates which recipe this input is associated with.

resourceTypeId - Integer. A foreign key referencing the **ResourceType** entity. Specifies the type of resource required.

quantity - Integer. The amount of the resource needed as input for the recipe. Must be a positive integer.

recipe - Recipe. The recipe this input belongs to.

resourceType - ResourceType. The type of resource required.

Usage

Represents the input resources required for a recipe.

RecipeOutput Entity

id - Integer. A unique identifier for each recipe output. (primary key)

recipeId - Integer. A foreign key referencing the **Recipe** entity. Indicates which recipe this output is associated with.

resourceTypeId - Integer. A foreign key to ref the **ResourceType** entity. Specifies the type of resource produced.

quantity - Integer. The amount of the resource produced by the recipe. Must be a positive integer.

recipe - Recipe. The recipe this output belongs to.

resourceType - ResourceType. The type of resource produced.

Usage

Represents the output resources produced by a recipe.

ItemType Entity

id - Integer. A unique identifier for each recipe output. (primary key)

recipeId - Integer. A foreign key referencing the **Recipe** entity. Indicates which recipe this output is associated with.

resourceTypeId - Integer. A foreign key to ref the **ResourceType** entity. Specifies the type of resource produced.

quantity - Integer. The amount of the resource produced by the recipe. Must be a positive integer.

recipe - Recipe. The recipe this output belongs to.

resourceType - ResourceType. The type of resource produced.

Usage

Represents buildable items like Miners, Constructors, and Assemblers.

ItemBuildCost Entity

id - Integer. A unique identifier (primary key) for each item build cost entry.

itemTypeId - Integer. A foreign key referencing the **ItemType** entity. Indicates which item this build cost is for.

resourceTypeId - Integer. A foreign key referencing the **ResourceType** entity. The type of resource required to build the item.

quantity - Integer. The amount of the resource needed to build the item. Must be a positive integer.

itemType - ItemType. The item that requires this build cost.

resourceType - ResourceType. The type of resource required.

Usage

Defines the resources required to build an item.

UserItem Entity

id - Integer. A unique identifier (pk) for each user item pair.

userId - Integer. A foreign key referencing the **User** entity. shows which user owns this item.

itemTypeId - Integer. A foreign key referencing the **ItemType** entity. Will describe the type of item.

quantity - Integer. The current amount of this item the user has. Must be zero or a positive integer.

user - User. The user who owns this item.

itemType - ItemType. The type of item this entry represents.

Usage

Represents the items owned by a user.

Miner Entity

id - Integer. A unique identifier (pk) for each miner.

type - String. The type of miner (for example, Portable Miner, Miner Mk1). Could be a foreign key to an **ItemType** if miners are part of items.

productionRate - Integer. The rate at which the miner produces resources, measured in units per minute.

associatedResourceTypeId - Integer. A foreign key referencing the **ResourceType** entity. showing which resource the miner extracts.

ownerUserId - Integer. A foreign key referencing the **User** entity. Shows which user owns this miner.

status - String. The operational status of the miner (active, inactive, paused etc.).

location - String or *Coordinates*. (May or may not be implemented. May just be fixed)

Usage

Items that automate the mining of ores.

Constructor Entity

id - Integer. A unique identifier (pk) for each constructor.

ownerUserId - *Integer*. A foreign key referencing the **User** entity. Specifies which user owns this constructor.

Status -*String*. The operational status of the constructor (active, idle, offline etc.).

currentRecipeId - *Integer*. A foreign key referencing the **Recipe** entity. Indicates which recipe the constructor is currently set to process.

Efficiency - *Float*. Represents how efficiently the constructor is operating, possibly affecting production speed.

location - String or *Coordinates*. The placement of the constructor in the game world. (May or may not be implemented. May just be fixed)

Usage

Machines that process one input resource into one output.

Assembler Entity

id - Integer. A unique identifier for each assembler. (pk)

ownerUserId - *Integer*. A foreign key referencing the **User** entity. Specifies which user owns this assembler.

status - String. The operational status of the assembler (active, idle, offline etc.).

currentRecipeId - Integer. A foreign key referencing the **Recipe** entity. Indicates which recipe the assembler is currently set to process.

efficiency - Float. Represents how efficiently the assembler is operating.

location - String or *Coordinates*. The placement of the assembler in the game world. (May or may not be implemented. May just be fixed)

Usage

Machines that process two input resources into one output.

recipeRequires Entity

recipeId - Integer. A foreign key referencing the Recipe entity. Indicates which recipe this requirement is associated with. Part of the composite primary key.

itemTypeId - Integer. A foreign key referencing the ItemType entity. Specifies the item or machine required to execute the recipe. Part of the composite primary key.

Recipe - Recipe. The recipe that requires the item or machine.

itemType - ItemType. The item or machine required to execute the recipe.

Usage

Specifies the items or machines required to execute a recipe.

Functional Requirements

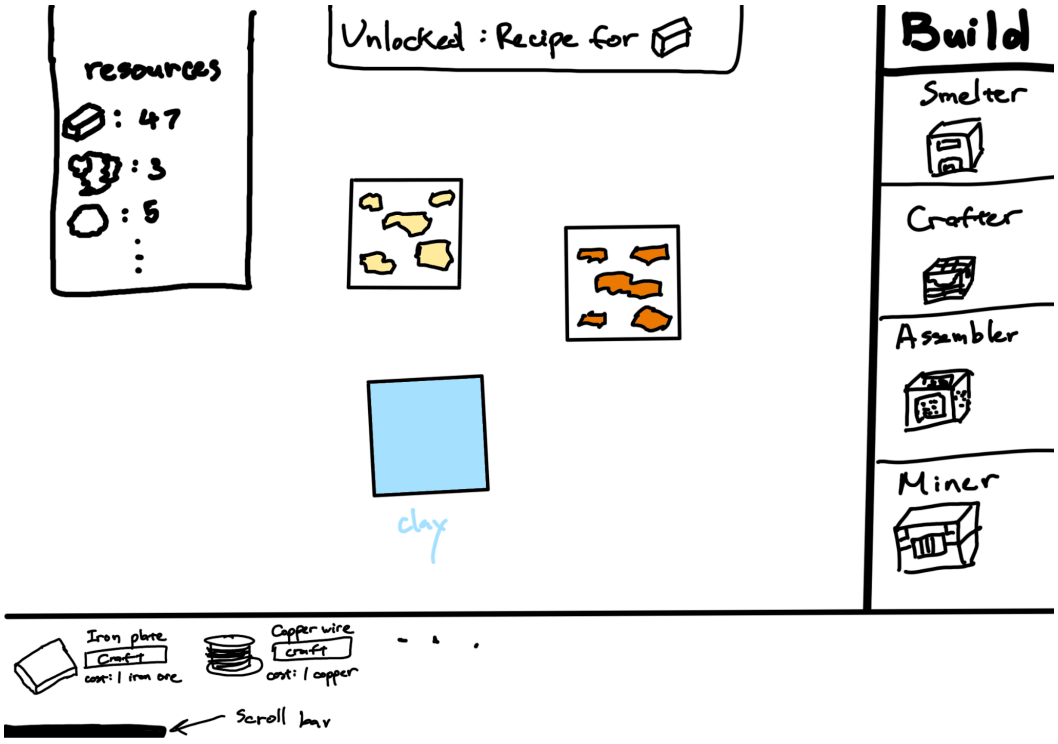
Priority	Functional Requirement	Description
0	Login, Logout	Basic user authentication functionalities.
0	Inventory Management	Ability to manage items and resources in the user's inventory.
1	Manually Mine Resource	Users can manually mine resources by clicking or selecting options.
1	3 Boxes	Interface elements for displaying information or resource counts.
1	Automated Mine Resource	Automated collection of resources using in-game miners.
1	Miners	Automated tools or characters that collect resources over time.
1	Manually Craft	Users can craft items manually by selecting recipes and resources.
1.1	Automated Craft: Crafter	Automatic crafting with a single input to output conversion per tick.
1.2	Automated Craft: Assembler	Automated crafting with multiple inputs to multiple outputs per tick.
1	UI/UX Graphics	Basic graphics and user interface elements.
2	Progression	In-game progression mechanics, such as leveling or unlocking features.
2	Leaderboard	Display leaderboard rankings based on resource values and points.
2	Retrievable Automated Guidance (RAG) Chatbot	Game assistant that provides tips, formulas, recipes, and inputs.
2	Achievement System	Track achievements and milestones within the game.
3	Skins	Customizable character or interface appearances.

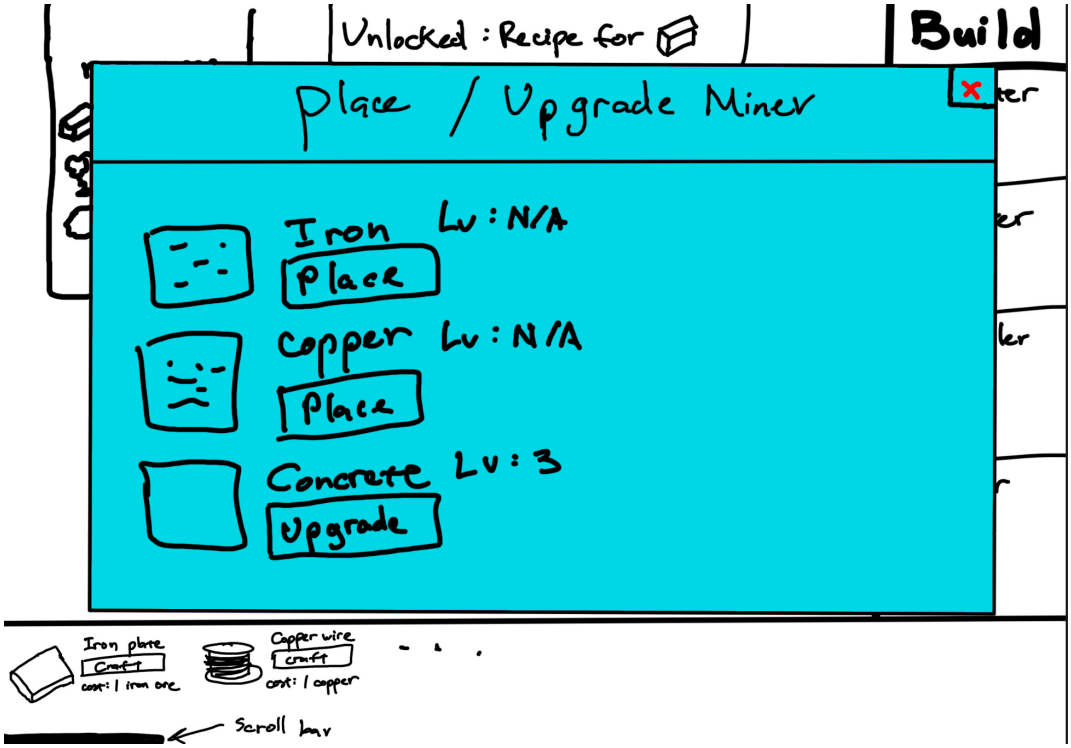
3	Minigames	Optional smaller games or activities within the main game.
3.1	Gambling	In-game gambling mechanics or mini-games involving chance.
4	Expanding Map	A larger, expandable map for exploring more areas.
4	Grid System	Map grid layout for resource placement or navigation.
4	Ads	Display advertisements within the game.

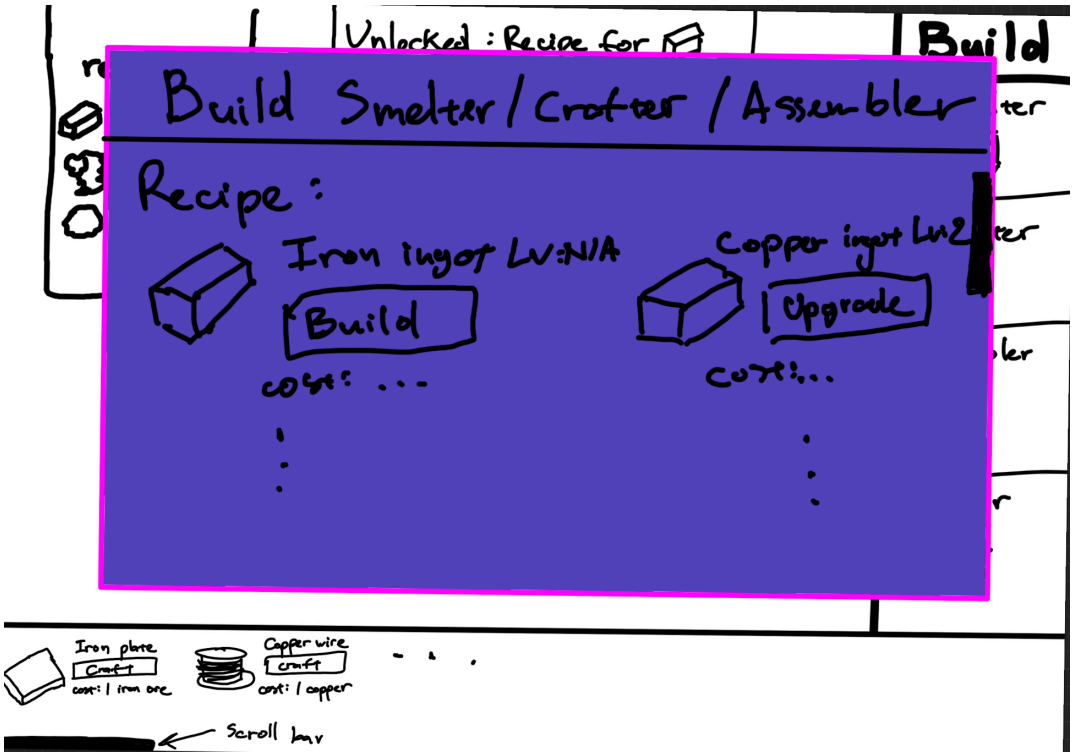
UI Mockups and UX Flows

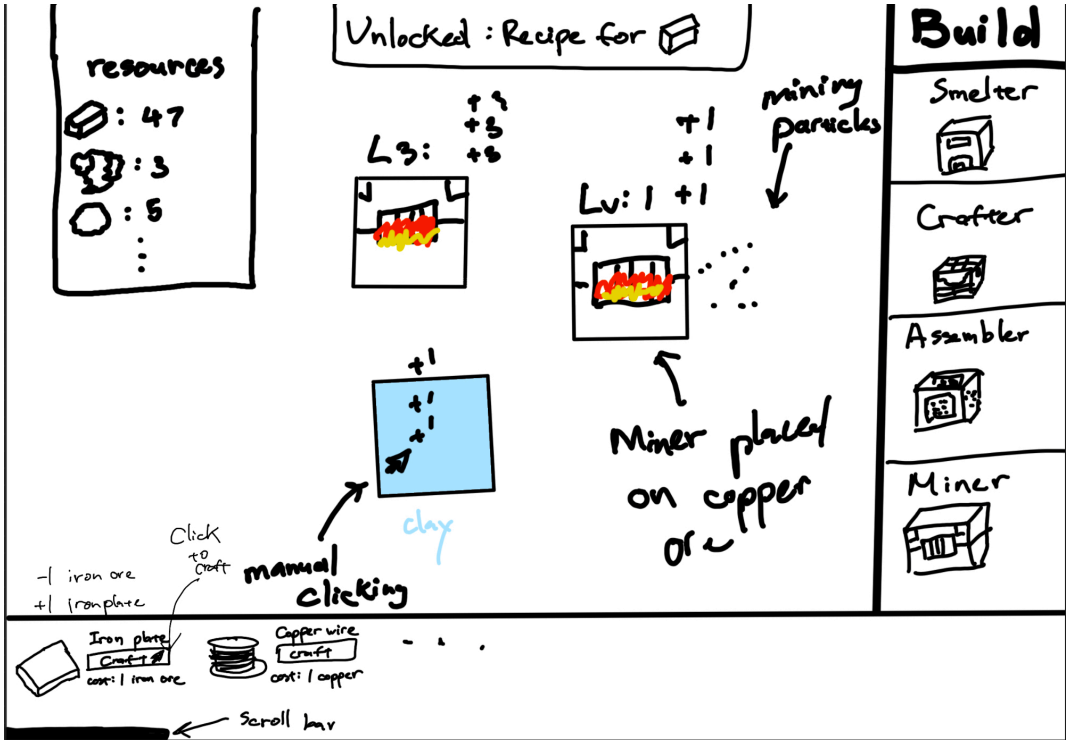
Summary

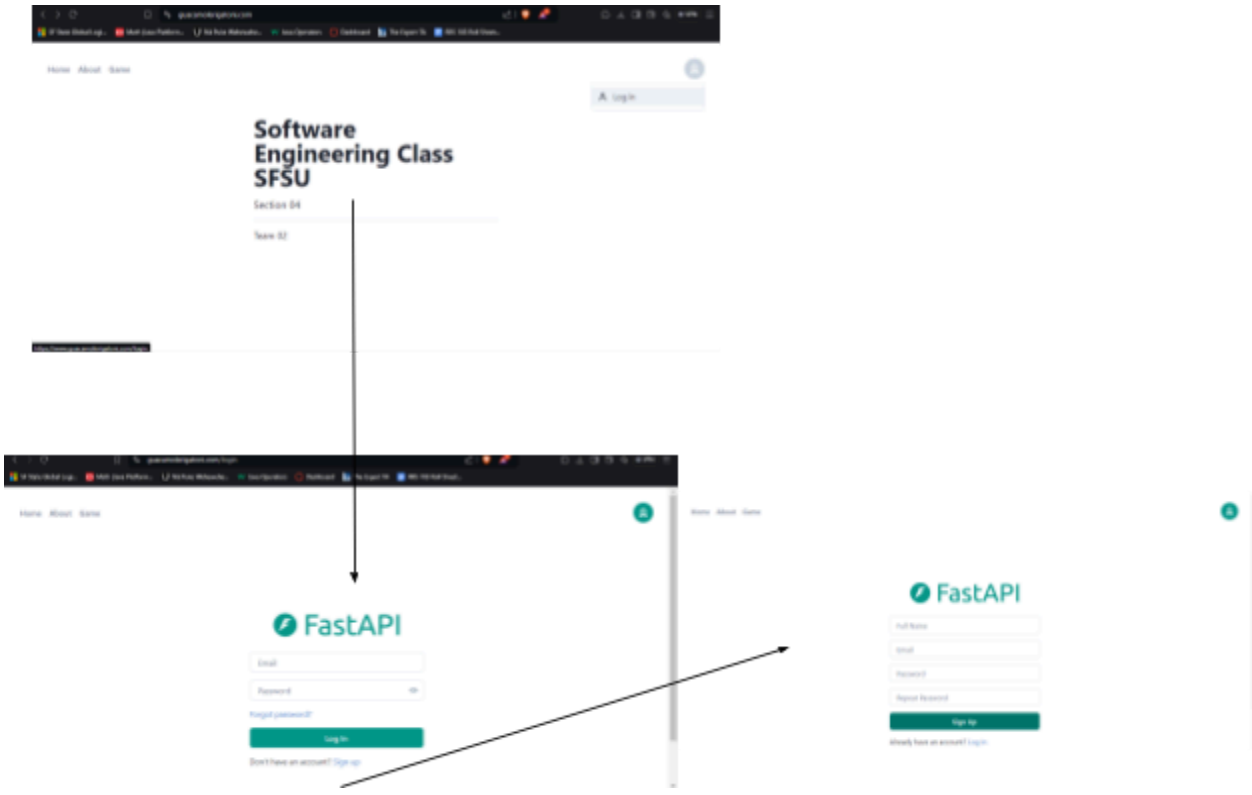
We created black-and-white wireframe prototypes for five key functions in our resource management game: resource management and crafting, building selection and placement, miner placement and upgrades, recipe unlock notifications, and building upgrades. Each mockup focuses on core user interactions, ensuring the design is useful, usable, desirable, accessible, findable, and credible. These prototypes will guide the development of major UX flows, with further refinements planned as we iterate on the design from the inspired game of minecraft for the purpose of testing.

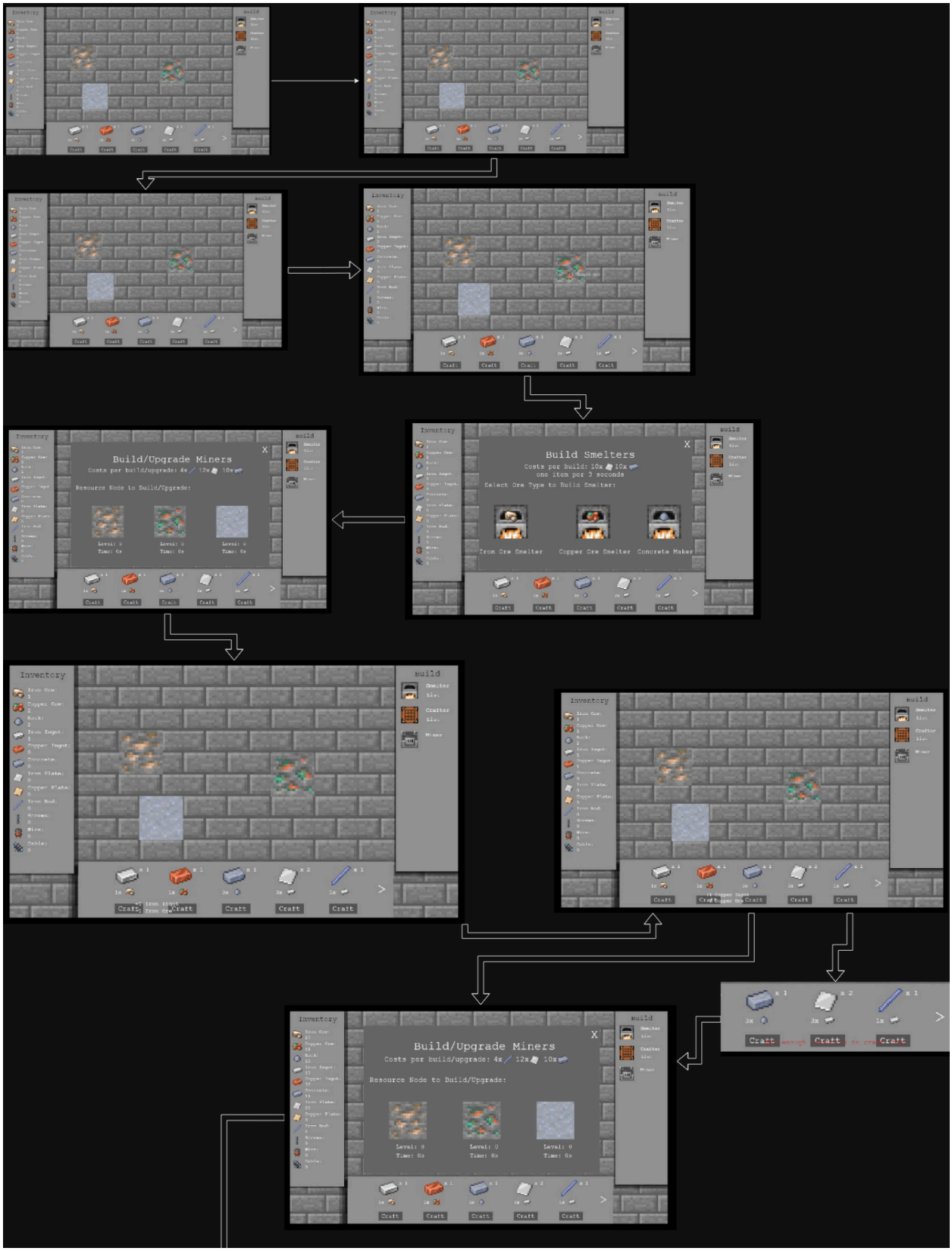


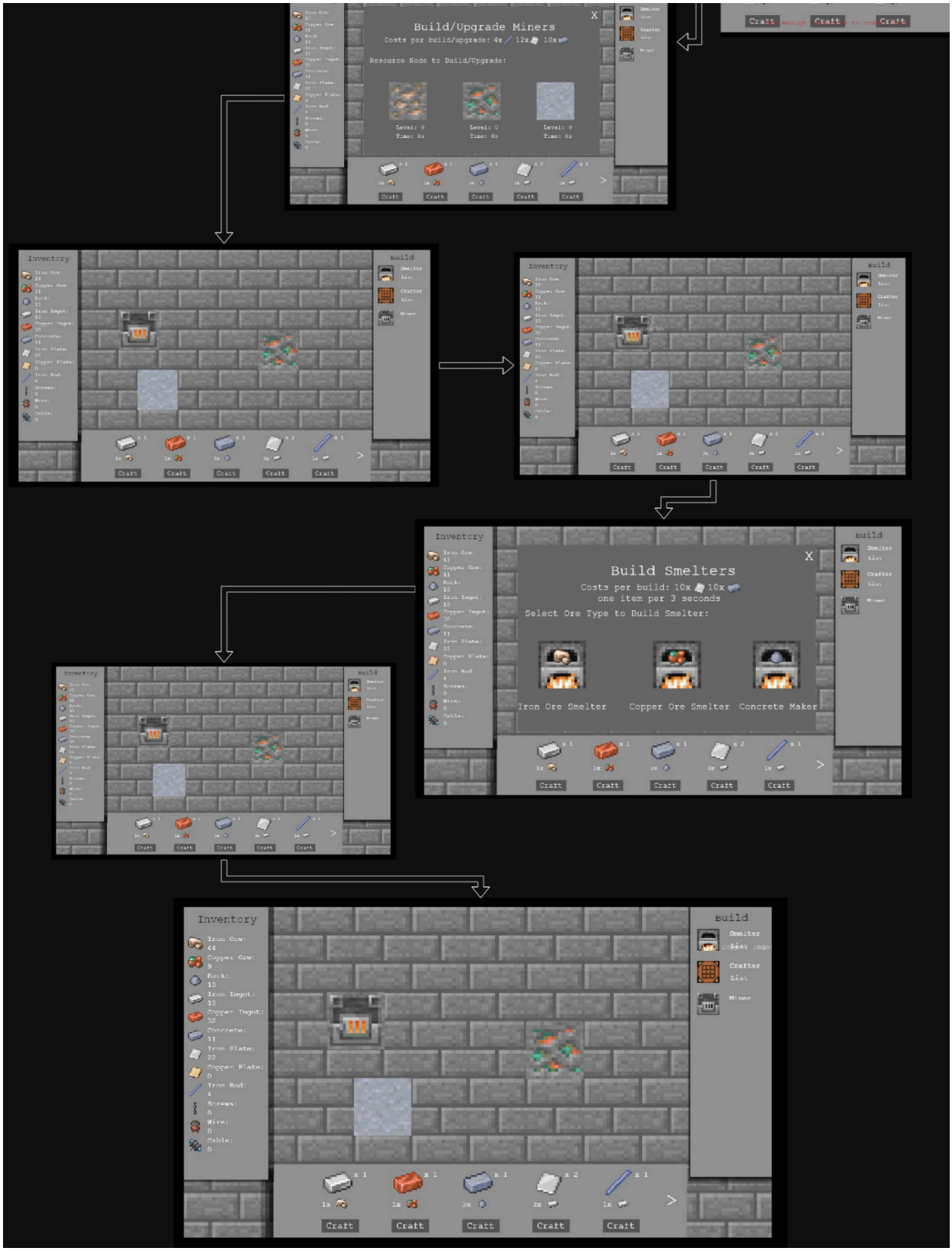

Unlocked : Recipe for
Build
Place / Upgrade Miner
Iron
Lv : N/A
Place
Copper
Lv : N/A
Place
Concrete
Lv : 3
Upgrade
Iron plate
Craft
cost: 1 iron ore
Copper wire
Craft
cost: 1 copper
Scroll bar


Unlocked : Recipe for
Build
Build Smelter / Crafter / Assembler
Recipe :
Iron ingot LV:N/A
Build
cost: ...
Copper ingot Lv2
Upgrade
cost:...
Iron plate
Craft
cost: 1 iron ore
Copper wire
Craft
cost: 1 copper
Scroll bar


resources
: 47
: 3
: 5
Unlocked : Recipe for
Lv: 1
+1
+1
+1
mining parucks
Lv: 1
+1
+1
+1
Miner placed on copper ore
clay
manual clicking
Click to craft
-1 iron ore
+1 ironplate
Iron plate
Craft 1
cost: 1 iron ore
Copper wire
1 craft
cost: 1 copper
Scroll bar
Build
Smelter
Crafter
Assembler
Miner


Software Engineering Class SFSU
FastAPI
FastAPI


Inventory
Build
Craft
Build/Upgrade Miners
Costs per build/upgrade: 4x 12x 10x
Resource Node to Build/Upgrade:
Level: 0
Time: 0s
Build Smelters
Costs per build: 10x 10x
one item per 3 seconds
Select Ore Type to Build Smelter:
Iron Ore Smelter
Copper Ore Smelter
Concrete Maker
Smelter
Crafter
Miner


Build/Upgrade Miners
Costs per build/upgrade: 4x 12x 10x
Resource Node to Build/Upgrade:
Level: 0
Time: 0s
Craft
Inventory
Build
Build Smelters
Costs per build: 10x 10x
one item per 3 seconds
Select Ore Type to Build Smelter:
Iron Ore Smelter
Copper Ore Smelter
Concrete Maker

Review for UX Principles

Useful

In the flow of how the game will grasp the user's interaction smoothly to play the game as constructed to be is the biggest goal to solve the problem of messy and rough UI design.

Usable

Overall the game will be easy to grasp as this game has similar features to other essential resource, building, and growth concepts.

Desirable

Undoubtedly this game will immerse the user to be on this platform as long as possible as they continue to grow their world the possibilities become more desirable to acquire.

Accessible

As accessible as this can be to those who have access to a webpage really can play this game out and as technology evolves for people who have certain disabilities with advanced technological tools that help them operate computers can access the webpage game.

Findable

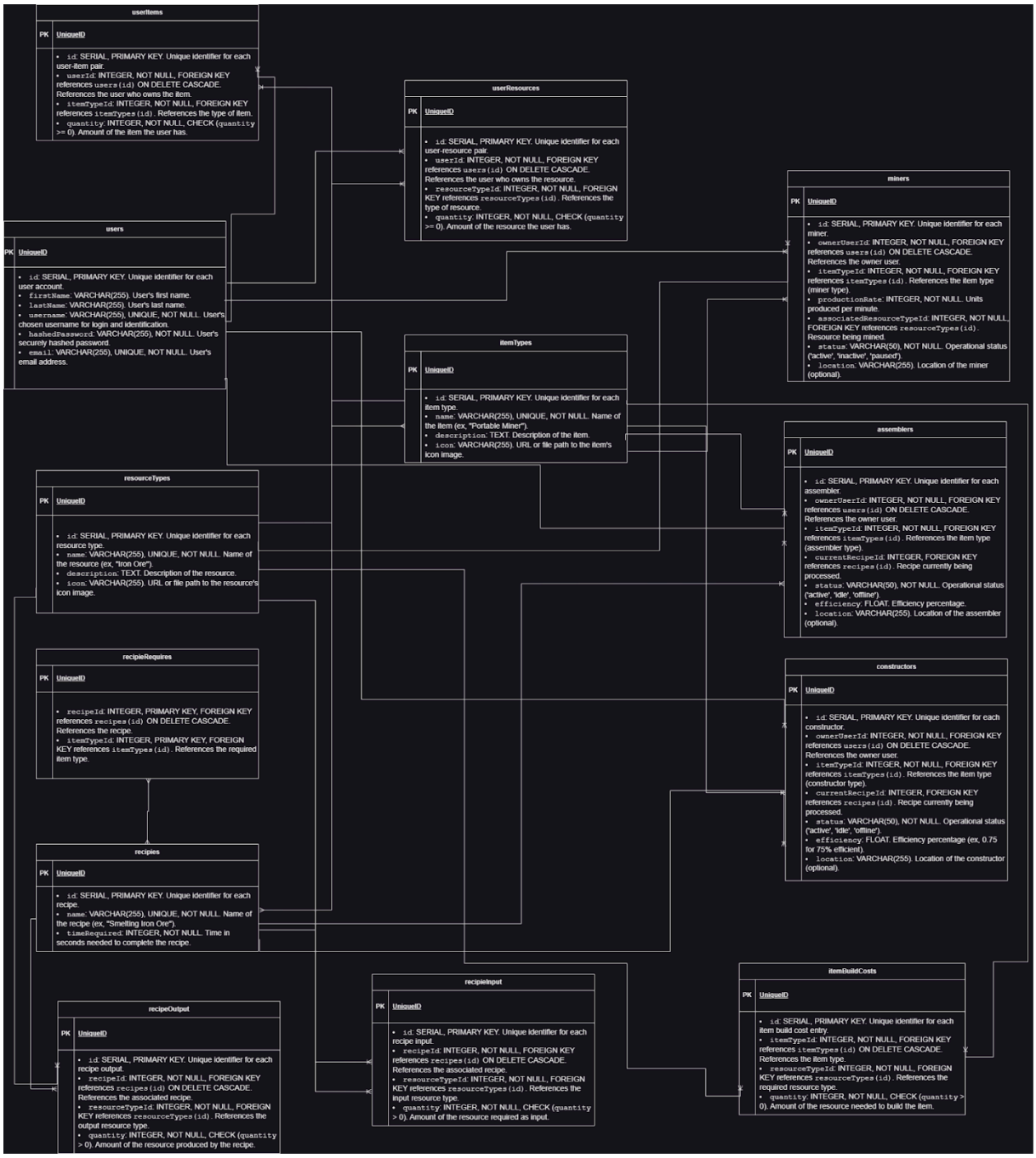
For every action to use for the game is very findable due to easy and intriguing UI design to show the resources, menu options, and

Credible

Our design will become more refined to create a community of trust within their progress and will have a different look from similar games that share the concepts that we are inspired to create.

High-Level Architecture, Database Organization

Database SCHEMA: [Table link for bigger viewing option]



Add/Delete/Search Architecture:

1. users Table

Stores information about registered user accounts.

Add Operations
User Registration: A new user is added to the users table when they register an account. The following fields are populated:
firstName
lastName
username (must be unique)
hashedPassword (securely hashed)
email (must be unique)

Delete Operations
Account Deletion: A user account can be deleted upon user request or administrative action. Deleting a user cascades to related tables due to ON DELETE CASCADE, removing associated records in userResources, userItems, miners, constructors, and assemblers.

Authentication: Search for a user by username or email during login to verify credentials.
Profile Management: Retrieve user details by id for profile updates or display.
Email Verification: Search by email to check if an email is already registered.

Display Operations

User Profile: Display firstName, lastName, username, and email on the user's profile page.

2. resourceTypes Table

Represents different types of resources available in the game.

Add Operations
Game Updates: New resource types are added when expanding the game content (admin operation).

Delete Operations
Resource Removal: Resource types can be deleted by admins if necessary. Caution is advised as it may affect game integrity.

Search Operations
Resource Listing: Retrieve all resources to display in the game interface.
Resource Details: Search by id or name to get details about a specific resource.

Display Operations

Resource Icons and Names: Display icon and name in resource selection menus.
Resource Descriptions: Show description when users view detailed information about a resource.

3. userResources Table

shows the quantity of each resource a user owns.

Add Operations
Resource Acquisition: When a user obtains a new type of resource for the first time, a new entry is added with:
userId
resourceTypeId
quantity

Delete Operations

Resource Depletion: Optionally, delete entries when quantity drops to zero.
Account Deletion: Entries are deleted when the associated user is deleted due to ON DELETE CASCADE.

Search Operations

Inventory Retrieval: Fetch all resources owned by a user using userId.
Specific Resource Check: Check if a user has a specific resource by userId and resourceTypeId.

Display Operations
User Inventory: Display quantity of each resource in the user's inventory.

4. recipes Table

Defines how resources can be transformed into other resources.

Add Operations
New Recipes: Add new recipes during game updates (admin operation), including:
name
timeRequired
Delete Operations
Recipe Removal: Admins can delete recipes, but this should be done VERY CAREFULLY.

Search Operations
Available Recipes: Retrieve all recipes to display crafting options.
Recipe Details: Search by id or name to get inputs, outputs, and requirements.

Display Operations
Crafting Menu: Display name and required timeRequired for each recipe.
Recipe Details: Show inputs and outputs associated with a recipe.

5. recipeInputs Table

Represents the input resources required for a recipe.

Add Operations
Define Recipe Inputs: Add entries when creating a new recipe, specifying:
recipeId
resourceTypeId
quantity

Delete Operations

Recipe Modification: Delete inputs when a recipe is removed or modified (admin operation).

Search Operations

Retrieve Inputs: Fetch all inputs for a given recipe using recipeId.

Display Operations

Crafting Requirements: Display required resources and quantities for a recipe.

6. recipeOutputs Table

Represents the output resources produced by a recipe.

Add Operations
Define Recipe Outputs: Add entries when creating a new recipe, specifying:
recipeId
resourceTypeId
quantity

Delete Operations
Recipe Modification: Delete outputs when a recipe is removed or modified (admin operation).

Search Operations

Retrieve Outputs: Fetch all outputs for a given recipe using recipeId.

Display Operations

Crafting Results: Display the resources produced by a recipe upon completion.

7. itemTypes Table

Represents buildable items like Miners, Constructors, and Assemblers.

Add Operations
New Items: Add new item types during game updates (admin operation), including:
name
description
icon

Delete Operations

Item Removal: Admins can delete item types; must ensure no user items depend on them.

Search Operations
Item Catalog: Retrieve all item types for display in build menus.
Item Details: Search by id or name to get details about an item.

Display Operations
Building Menu: Display icon, name, and description in the UI.
Item Details: Show build costs and functions of the item.

8. itemBuildCosts Table

Defines the resources required to build an item.

Add Operations
Specify Build Costs: Add entries when creating a new item type, specifying:
itemTypeId
resourceTypeId
quantity

Delete Operations
Item Modification: Delete build costs when an item is removed or its costs change (admin operation).

Search Operations

Retrieve Build Costs: Fetch all build costs for a given item using itemTypeId.

Display Operations

Building Requirements: Display required resources and quantities to build an item.

9. userItems Table

Represents the items owned by a user.

Add Operations
Item Acquisition: When a user builds or acquires a new item type, add an entry with:
userId
itemTypeId
quantity

Delete Operations

Item Consumption: Optionally, delete entries when quantity drops to zero.
Account Deletion: Entries are deleted when the associated user is deleted due to ON DELETE CASCADE.

Search Operations
Inventory Retrieval: Fetch all items owned by a user using userId.
Specific Item Check: Check if a user has a specific item by userId and itemTypeId.

Display Operations

User Inventory: Display quantity of each item in the user's inventory.

10. miners Table

Items that automate the mining of ores.

Add Operations
Build Miner: When a user constructs a miner, add an entry with:
ownerUserId
itemTypeId
productionRate
associatedResourceTypeId
status
location (may or may not add in*MAYBE*)

Delete Operations
Deconstruct Miner: Remove the miner entry when a user deconstructs it.
Account Deletion: Entries are deleted when the associated user is deleted.

Search Operations
Miner Management: Retrieve all miners owned by a user using ownerUserId.
Operational Status: Search miners by status to manage production.

Display Operations
Miner List: Display details of each miner owned by the user.

Production Rates: Show productionRate and status in the UI.

11. constructors Table

Machines that process one input resource into one output.

Add Operations
Build Constructor: When a user constructs a constructor, add an entry with:
ownerUserId
itemTypeId
currentRecipeId (can be null to start)
status
efficiency
location (may or may not add in*MAYBE*)

Delete Operations
Deconstruct Constructor: Remove the constructor entry when deconstructed.
Account Deletion: Entries are deleted when the associated user is deleted.

Search Operations
Constructor Management: Retrieve all constructors owned by a user.
Operational Status: Search constructors by status or currentRecipeId.

Display Operations
Constructor List: Display details and status of each constructor.
Current Recipe: Show the recipe being processed and efficiency.

12. assemblers Table

Machines that process two input resources into one output.

Add Operations
Build Assembler: When a user constructs an assembler, add an entry with:
ownerUserId
itemTypeId
currentRecipeId (can be null to starrt)
status
efficiency
location (may or may not add in*MAYBE*)

Delete Operations
Deconstruct Assembler: Remove the assembler entry when deconstructed.
Account Deletion: Entries are deleted when the associated user is deleted.

Search Operations
Assembler Management: Retrieve all assemblers owned by a user.

Operational Status: Search assemblers by status or currentRecipeId.

Display Operations
Assembler List: Display details and status of each assembler.
Current Recipe: Show the recipe being processed and efficiency.

13. recipeRequires Table

Specifies the items or machines required to execute a recipe.

Add Operations
Define Requirements: Add entries when creating a new recipe, specifying:
recipeId
itemTypeId

Delete Operations
Recipe Modification: Delete requirements when a recipe is removed, or its requirements change (admin operation).

Search Operations

Retrieve Requirements: Fetch all required items for a recipe using recipeId.

Display Operations

Crafting Requirements: Display required items (ex. machines) needed to execute a recipe.

APIs

1. Backend APIs:

Method	Inputs	Outputs	Description
Post /login	Username Password (hashed)	Auth token	Used to login user
Post /logout	Auth token	Status Code	Used to logout user
Post /createUser	Email Password (hashed) UserName	Status Code	Used to create a new user
Post /uploadData	Auth token Resources: { Name: value,	Status Code	Used to upload the data from the frontend into the backend, so

	Name2: value2,
	Name3: value3,
	}
	Facilities: {
	Name: value,
	Name2: value2,
	Name3: value3,
	}
			that it can be imported in the future

Get /loadData	Auth token	Resources: {
		Name: value,
		Name2: value2,
		Name3: value3,
		}
		Facilities: {
		Name: value,
		Name2: value2,
		Name3: value3,
		}
			Used to load the data from the backend into the front, so that values can be filled out on the frontend

Post /upgrade	Auth token Facility	Status Code (success or failure)	Upgrade a facility (crafter / smelter /assembler)
Post /craft	Auth Resource	Status Code (success or failure)	Craft a resource

2. Third Party APIs:

- Have Traefik Proxy API as a part of the fast api template. Allows us to set up a load balancer
- Have JWT (JSON Web Token) as a part of the fast api template. Allows us to set up user authentication

2. Open-Source Components:

- Using Chakra UI to help with front end components in place of react.
- Using Playwright to help with end to end testing of our framework
- Using Pytest to help with testing the backend
- Using GitHub Actions for our CI/CD Pipeline
- Using Docker to allow everyone to work on the same environment
- Using Phaser for the game engine

4. Changes to Software Tools/Frameworks:

Replaced MongoDB with PostgreSQL.

- Helps us use the FastAPI Template

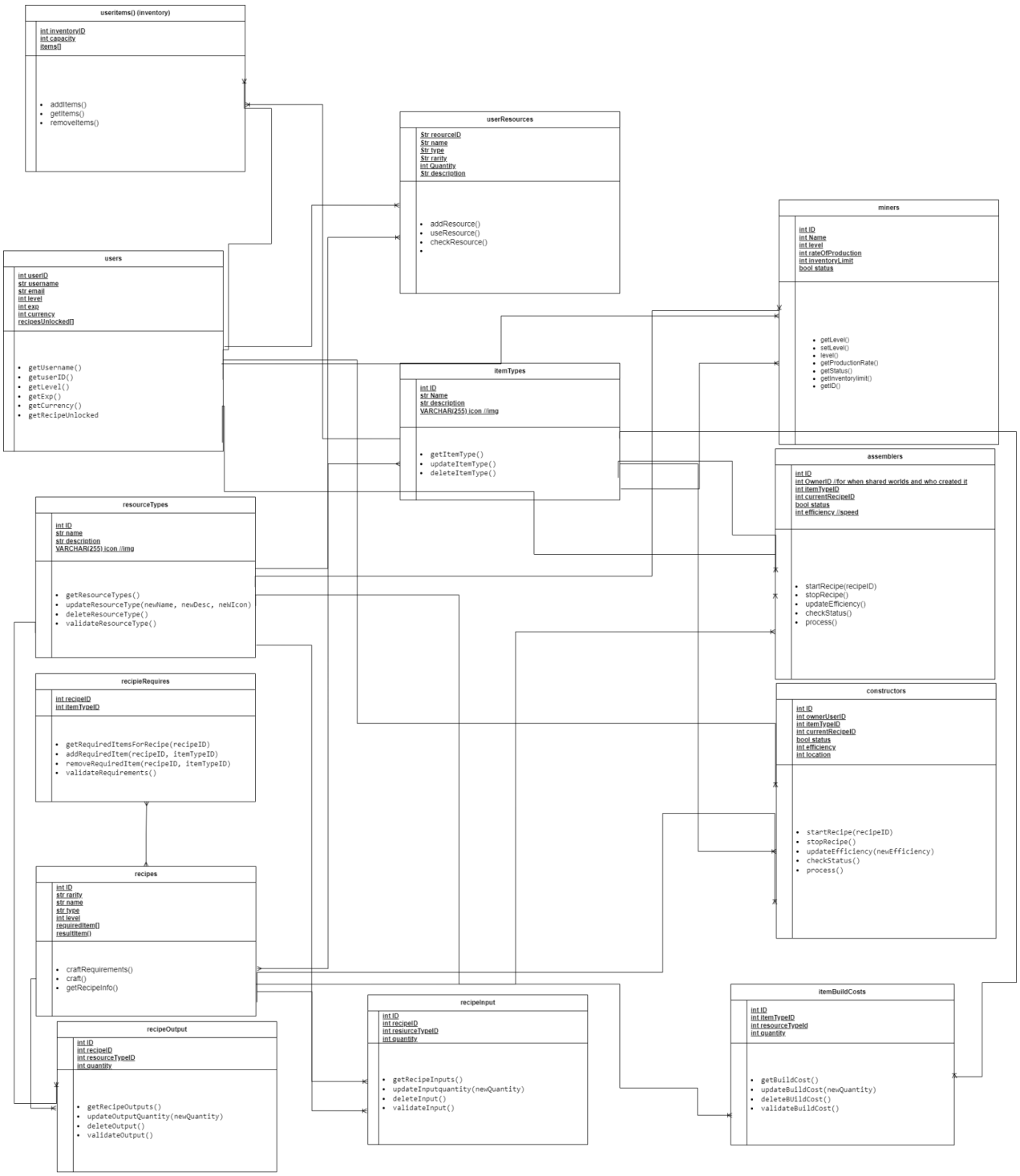
Now uses FastAPI.

- A template to help us with setting up docker and a variety of other things

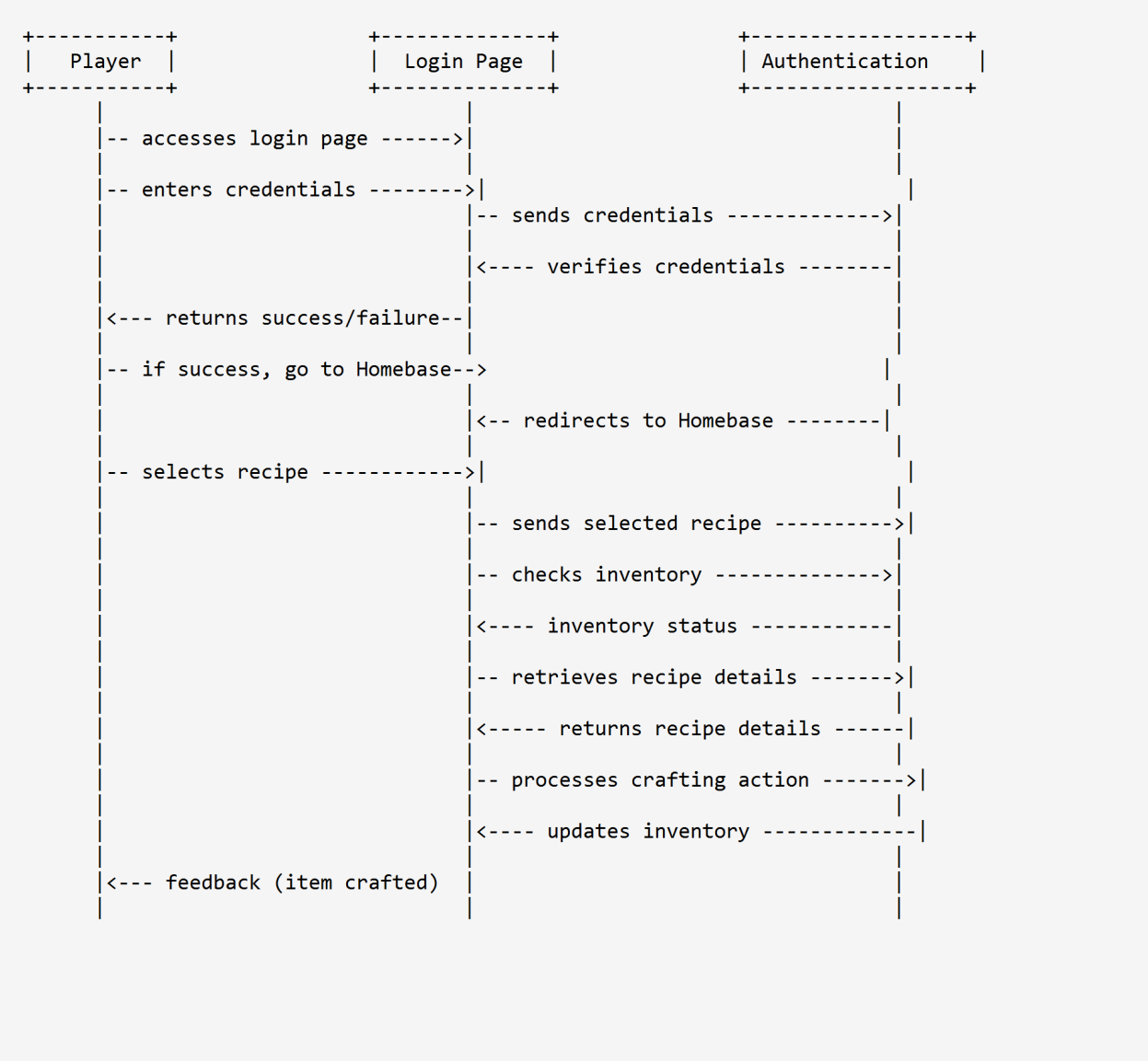
Added basic testing with Pytest.

- Will be using this to set up Unit tests to ensure that any changes made keep the code functional in the main branch

High-level UML Diagrams

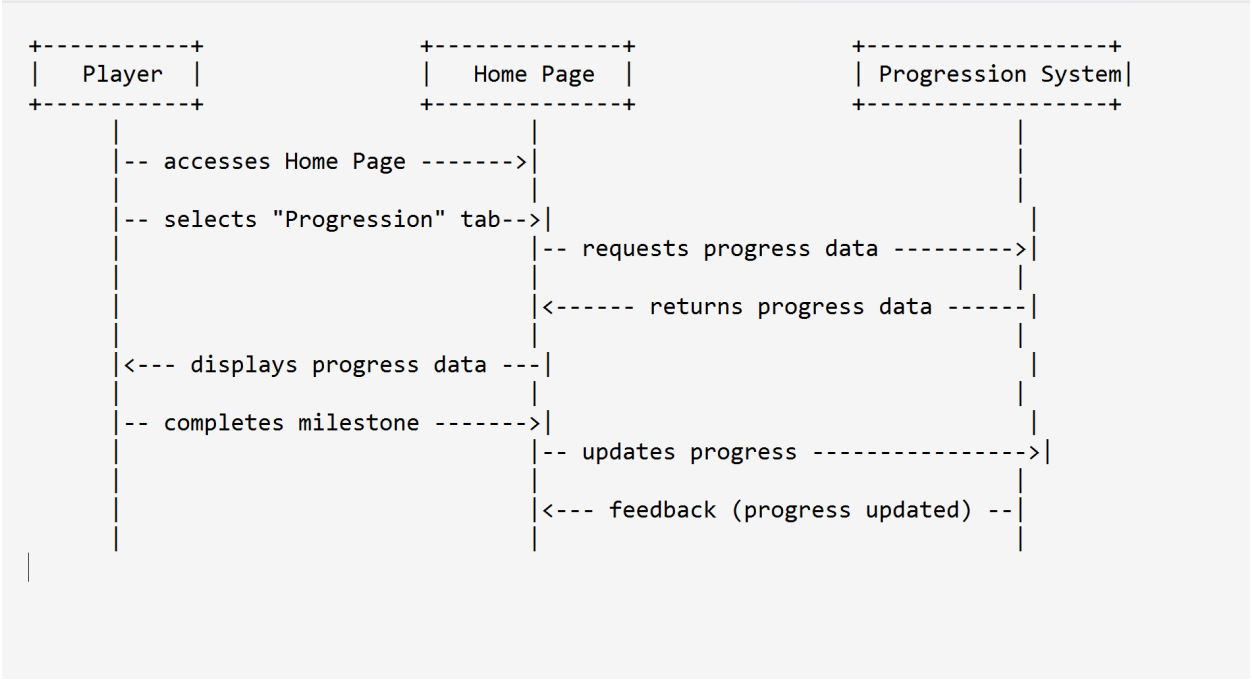


General sequence Diagram

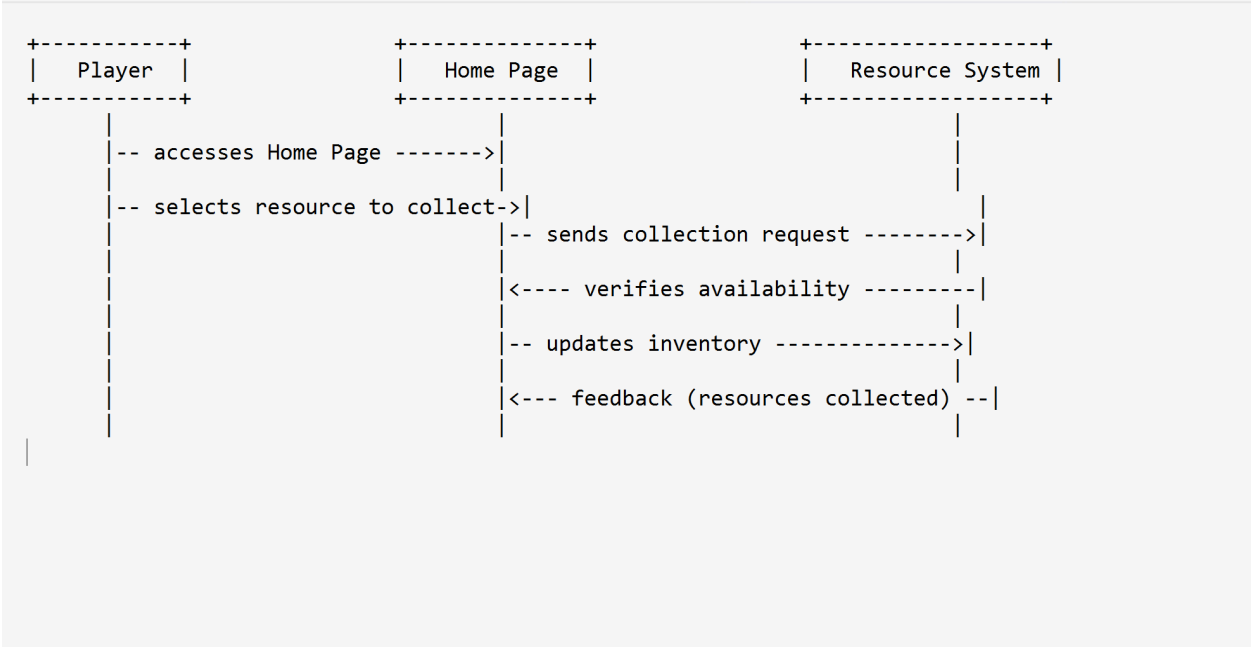


Ln 36, Col 1 | 2,651 characters

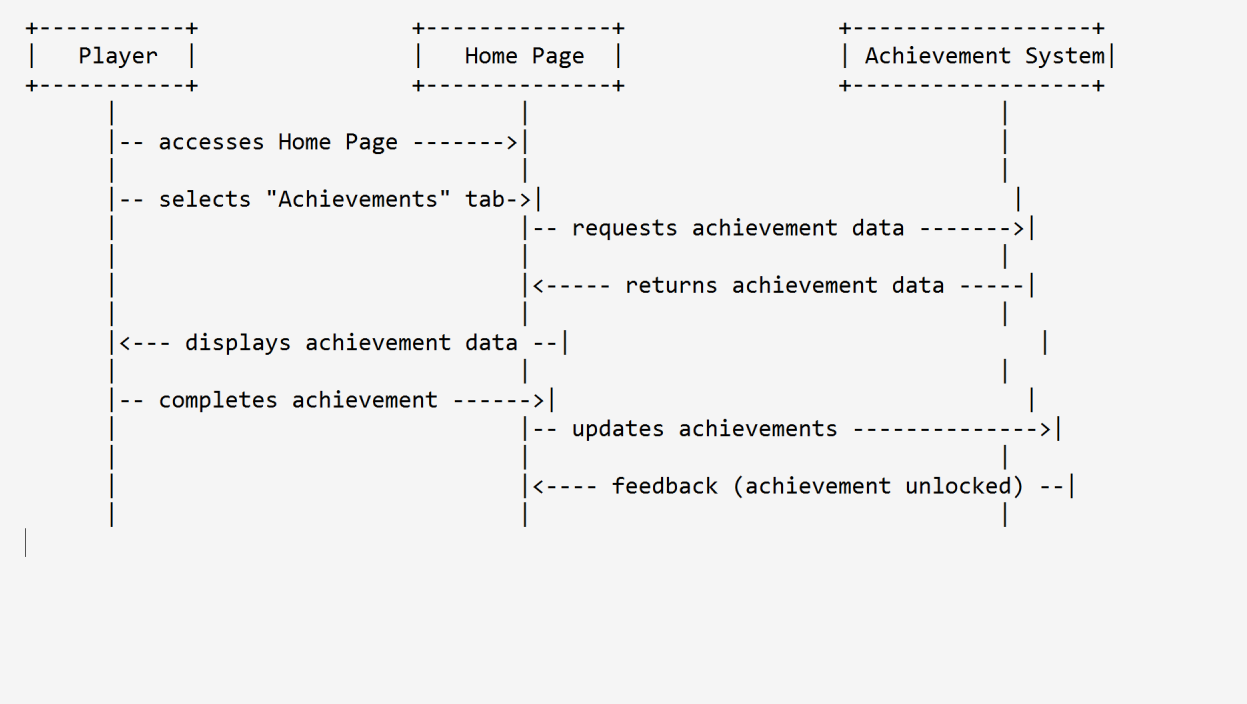
Function 1 Player Progression



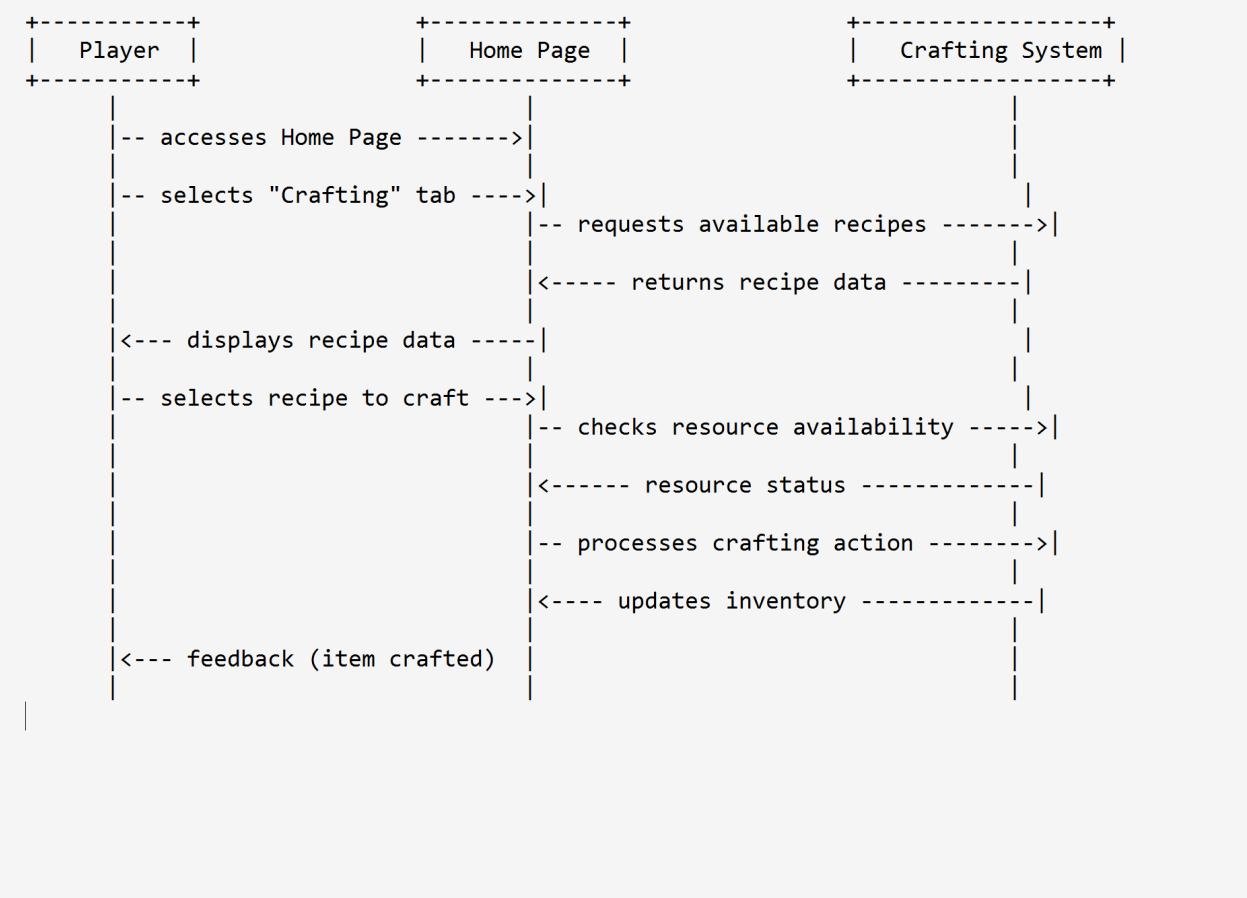
Function 2 Engaging Resource Mechanics



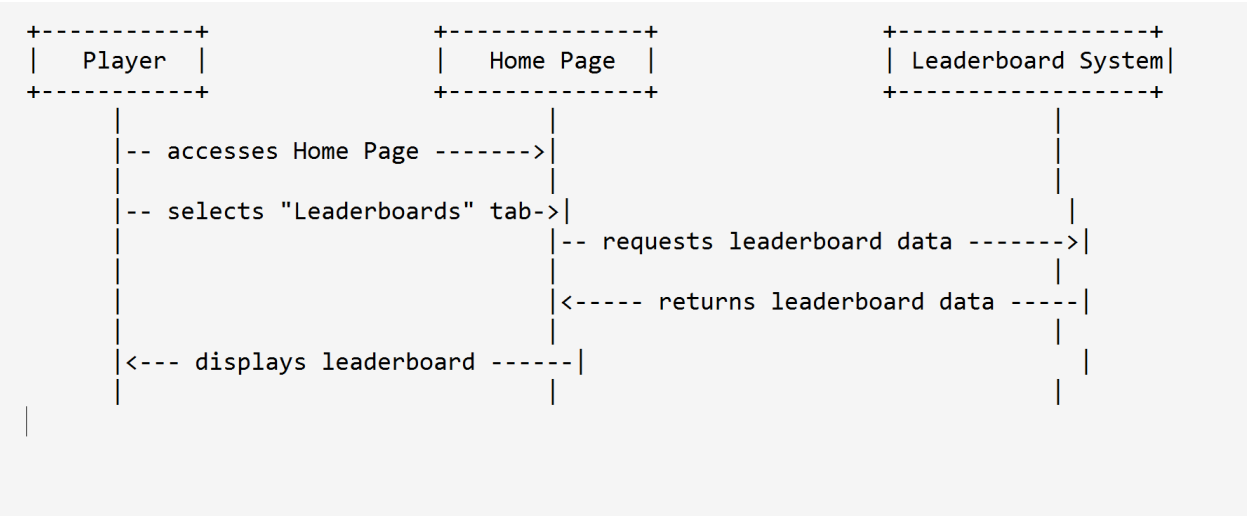
Function 3 Achievement System



Function 4 Crafting System



Function 5 Leaderboards/Scoreboards



Identify actual key risks for your project at this time

Skills Risks and Mitigation Plan

- **Risk:** Team members are new to the current stack.
    - **Details:** The team recently updated the stack, which now uses SQL for database management, FastAPI for the backend, and Phaser for frontend development. There is a learning curve associated with understanding these technologies.
- **Mitigation:**
    - Each member of frontend is assigned to learn Phaser by creating their own version of a cookie clicker game using online resources.
    - Each member of backend is assigned to understand SQL for database management and queries, with a focus on integration with the back-end.
    - Members share their progress during weekly scrum meetings, presenting insights and explaining how different components work.
    - Our Github Master has documented resources such as commands and instructions for deployment, development, frontend, and backend.

Schedule Risks

- **Risk:** The team may not stay on schedule due to unforeseen changes or delays.
    - **Details:** Weekly meetings are planned for the entire team, along with separate meetings for the frontend and backend groups. Each member's progress is monitored by team lead(s), and Jira is used for task tracking. Unplanned events or changes in requirements may cause delays in the project timeline.
- **Mitigation:**
    - Any changes are updated transparently in Jira.
    - The team reports any issues on the Discord server to address problems promptly.
    - The scrum master reviews and manages everyone's progress and updates team and the schedule weekly.
    - Task workload is balanced and split weekly to avoid overburdening team members.

    - The team schedules work one week ahead of time to allow for adjustments if tasks aren't completed. If any task remains unfinished, team members will have another week to complete it before it impacts the project timeline.

Teamwork Risks

- **Risk:** Inconsistent participation or uneven progress among team members.
    - **Details:** Participation and steady progress are critical for meeting project goals. Issues may arise if members do not contribute equally or fall behind on tasks.
- **Mitigation:**
    - Require each team member to provide regular status updates on their tasks, even outside of meetings, using a shared document or project management tool like Jira.
    - Pair up team members to review each other's progress and offer support. This helps detect potential issues early and encourages mutual assistance.
    - If a member misses a meeting, they must catch up through meeting summary notes created by Scrum Master.
    - Team members are encouraged to share any personal challenges or difficulties that may affect participation. The team will work together to accommodate and support each other.
    - Scrum master and team lead(s) check progress regularly, and any issues with workload are addressed by redistributing tasks or adjusting deadlines.

Legal/Content Risks

- **Risk:** Possible issues with content licensing or copyright for UI/UX design.
    - **Details:** The frontend UI/UX content is currently using Minecraft item assets for prototyping.
- **Mitigation:**
    - Ensure all content is created in-house or sourced from open-license resources.
    - Document any third-party resources used, verifying proper licenses and attribution.
    - We are going to be in the process of creating and using our own pixel art for the game.

Project Management

During M2, our team used scrum meetings to coordinate tasks, share progress, and address challenges. Each week, we hold scrum meetings where the agenda, set by the scrum master, was followed. The agenda included instructions on what to prepare before the meeting, a review of previous meeting minutes, and discussions on what each member accomplished, their plans for the upcoming week, and any obstacles encountered. We used sprints to manage task distribution, providing a clear framework for tracking progress. This approach ensured that updates were shared transparently, allowing all members to be aware of the team's status and any roadblocks. This not only fostered accountability but also encouraged collaboration, making it easier for team members to step in and help when someone faced a challenge.

To manage tasks, we utilized Jira as our project management tool. The platform allowed for real-time updates on task status, offering visibility into individual workloads and overall project progress. Tasks were prioritized and organized by status, helping the team focus on high-priority items and meet deadlines effectively. By tracking tasks through sprints, we maintained an organized workflow and could adjust priorities when necessary.

Communication extended beyond the weekly scrum meetings through our Discord server, where ongoing updates and quick discussions took place. Team members regularly informed one another about task completion, progress, and any issues they were facing, keeping the team aligned. Weekly check-ins also complemented the scrum meetings, allowing for a review of progress against milestones and any needed adjustments to the task plan. This combination of structured meetings and ongoing communication ensured that the team stayed on track throughout the M2 phase.

Team

Members

The list of team member names and their roles are repeated here:

Member	Role
Katy Lam	Team Lead
Arjun Singh Gill	Back-end
Matthew Aaron Weesner	Back-end
Niko Galedo	Front-end
Kevin Lam	Front-end
Kullathon “Mos” Sitthisarnwattanachai	Git Master
Arizza Cristobal	Scrum Master

M2 Checklist

The following checklist have the requirements for Milestone 2.

Item	Status
1. **Data Definitions**: ○ Define major data items and sub-data items. ○ Specify formats, sizes, and metadata for images/videos. ○ Ensure consistency in naming across documents and code. ○ Fully define user privileges and main info.	DONE
2. **Functional Requirements v2**: ○ Expand existing requirements with more details. ○ Maintain reference numbers from Milestone 1. ○ Prioritize each requirement (1 - must have; 2 - desired; 3 - opportunistic).	DONE
3. **UI Mockups and UX Flows**: ○ Create UX prototypes for 5-6 high-priority functional requirements. ○ Use black and white wire diagrams to focus on UX flows. ○ Develop mockup screenshots for user stories. ○ Review prototypes against UX principles (useful, usable, desirable, accessible, findable, credible).	DONE
4. **High-Level Architecture and Database Organization**: ○ Describe the main database schema (tables/collections and columns/fields). ○ Specify operations (add/delete/search) for each DB table or collection.	DONE
5. **API**:	DONE

- Define major backend APIs for frontend-backend communication.
- Describe any 3rd party APIs and open-source components used.
- Document any changes to software tools or frameworks.

6. **High-Level UML Diagrams**: DONE
   - Create high-level UML class diagrams for core functionality.
   - Develop high-level sequence diagrams for 5-6 functional requirements.

7. **Identify Actual Key Risks**: DONE
   - Identify specific skills, schedule, teamwork, and legal/content risks.
   - Develop mitigation plans for each risk.
   - Share risk assessments with the team.

8. **Project Management**: DONE
   - Conduct scrum meetings for progress sharing.
   - Use a project management tool (e.g., Notion, Jira) to track tasks.
   - Ensure transparent communication of tasks and changes.

9. **Vertical SW Prototype** ON TRACK
   - It functions as expected with correct data retrieval and display.
   - It is well-organized, properly documented, and deployed on the server.
   - It is submitted correctly following the email process.