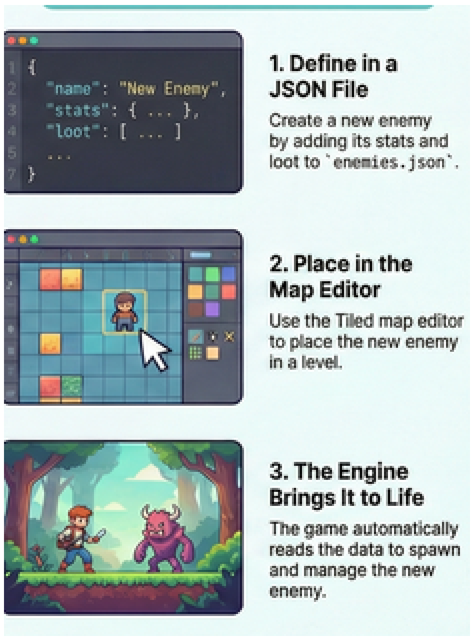
RPG Game

Executive Summary

RPG Game is a 2D top-down action RPG . Built on the Phaser 3 engine with JavaScript (ES6 Modules), the project's core philosophy is a "Data-Driven" and modular architecture. This design decouples the core game engine from its content—such as levels, items, enemies, and quests—which are defined in external JSON files. This approach facilitates rapid iteration, scalability, and modification.

The game features real-time, action-oriented combat, a persistent inventory system, and a robust loot-drop mechanic governed by weighted probability tables. Players progress through a linear sequence of distinct biomes, from forests and dungeons to a cyberpunk city and underwater ruins. Recent development efforts, under the "Polish & Controls" update (v1.2.0), have focused on enhancing the user experience by adding dedicated scenes for settings, controls, and pausing, as well as refining game balance and visual elements. The current roadmap prioritizes the implementation of a full save/load system and the development of complex boss mechanics.

Core Philosophy and Technical Architecture



```
{
  "name": "New Enemy",
  "stats": { ... },
  "loot": [ ... ]
  ...
}
```

1. Define in a JSON File

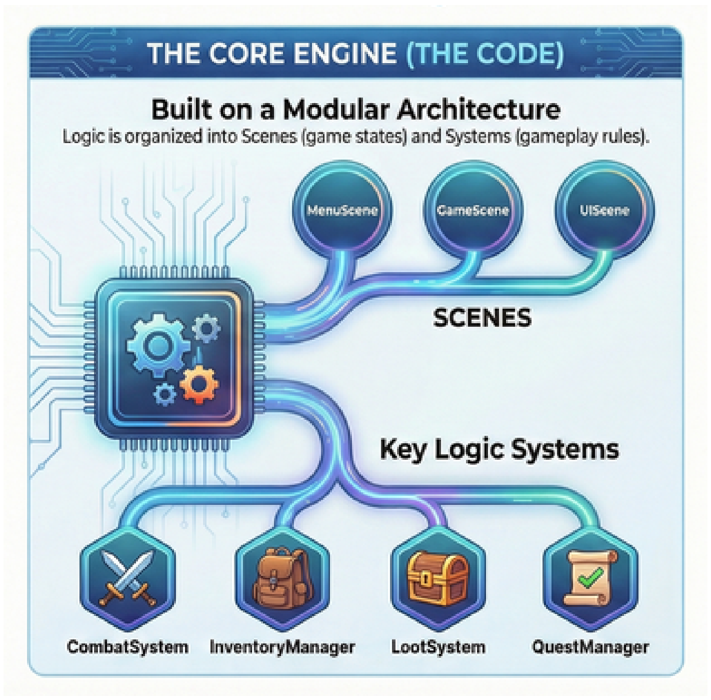
Create a new enemy by adding its stats and loot to `enemies.json`.

2. Place in the Map Editor

Use the Tiled map editor to place the new enemy in a level.

3. The Engine Brings It to Life

The game automatically reads the data to spawn and manage the new enemy.



Built on a Modular Architecture

Logic is organized into Scenes (game states) and Systems (gameplay rules).

The project is fundamentally designed to be data-driven and modular, where the codebase functions as a flexible engine powered by external data configurations. This separation allows developers to extend the game by editing simple JSON files rather than modifying core engine logic.

Technology Stack

Component	Technology/Tool	Purpose
Core Engine	Phaser 3 (JavaScript/ES6)	Provides robust scene management, asset handling, and core game loop functionality suited for 2D web games.
Physics Engine	Arcade Physics	A lightweight engine used for Axis-Aligned Bounding Box (AABB) collision detection.
Map Editing	Tiled	Used for creating and editing game levels, which are then exported as JSON files.
Build Tooling	Node.js / npm	Manages project dependencies.
Asset Formats	PNG, JSON, MP3	Standard web formats for images, data, and audio.

Project Structure Overview

The source code, located in the `src/` directory, follows a strict component-based architecture designed for clarity and separation of concerns.

- **/assets**: Contains all raw game assets, including images, audio files, and tilemaps.
- **/config**: Stores global game constants and input mappings.
- **/core**: Houses low-level utilities, such as the `EventBus` for decoupled communication between systems and the `SaveSystem` (a LocalStorage wrapper).
- **/data**: Functions as the "brain" of the game, containing all JSON files that define game content (e.g., `items.json`, `enemies.json`, `levels.json`, `quests.json`).
- **/entities**: Includes the class definitions for all game objects. The `Entity.js` file serves as the base class, with specific implementations for `Player.js`, `Enemy.js`, and `Item.js`.
- **/scenes**: Manages game states using Phaser's Scene system. This includes scenes for booting, menus, core gameplay (`GameScene`), UI overlays (`UIScene`), game over, settings, and more.

- **/systems**: Contains logic managers that handle specific gameplay mechanics, such as `CombatSystem.js`, `InventoryManager.js`, `LootSystem.js`, and `MapManager.js`.

Gameplay Systems Deep Dive

Entity System

Entities are constructed dynamically at runtime based on their definitions in the `data` directory's JSON files.

- **Enemies**: Defined in `enemies.json`, enemies are loaded with properties like HP, speed, damage, chase distance, and an assigned loot table. Current enemies include:
    - Green Slime
    - Blue Slime
    - Goblin Scout
    - Goblin Warrior
    - Null Shard (Boss)
- **Items**: Physical item drops in the world are handled by `Item.js`, but their properties (type, effect, rarity, etc.) are sourced from `items.json`. Examples include healing potions and weapons with specific damage stats.

Combat Mechanics

The combat system is designed for real-time action and provides clear player feedback.

- **Action**: Players attack using the `SPACE` key or `F` keys.
- **Damage Calculation**: Damage is calculated with the formula `(Attacker.Damage - Defender.Defense)`. Enemies deal contact damage to the player.
- **Player Invulnerability**: After taking a hit, the player character flashes red and enters a temporary invulnerability window ("I-Frame") to prevent instant death from multiple rapid hits.
- **Feedback**: The `CombatSystem` provides visual feedback through flashes on hit and applies knockback physics to affected entities.
- **Death**: When a player's HP reaches `0`, the `GameOverScene` is triggered, offering options to Retry or Return to the Main Menu.

Inventory and Loot System

The inventory is managed by a persistent singleton that preserves its state across scene transitions.

- **Data-Driven Items**: Items are defined in `items.json` with properties such as `type` (e.g., CONSUMABLE, WEAPON), `stackable`, `rarity`, and a defined `effect`.
- **UI and Controls**: The inventory UI is toggled with the `I` key. Players can click items for details, use consumables, or drop items. The first five inventory items are mapped to Quick Slots accessible via keys `1-5`.
- **Loot Logic**: When an enemy is defeated, the `LootSystem` rolls for drops based on the `lootTable` ID assigned to that enemy in `enemies.json`. These tables, defined in `lootTables.json`, support weighted probabilities, guaranteed drops, and quantity ranges.
- **Recent Balancing**: To prevent farming, restorative item drops (e.g., Med/Energy kits) have been balanced with a "Once Per Level" drop rule. Coin drops remain randomized.

Level Progression and World Design

Level progression is managed by the `MapManager` system, which parses Tiled maps and spawns entities according to definitions in `levels.json`. The game supports a linear progression through a series of distinct biomes.

- **Current Level Flow:**
    1. **Chamber** (Tutorial/Start)
    2. **Forest** (Nature Biome)
    3. **Dungeon** (Underground)
    4. **Cave** (Deep Earth)
    5. **Cyberpunk City** (Sci-Fi Twist)
    6. **Ancient Library** (Lore/Boss Prep)
    7. **Underwater Ruins** (Endgame)
- **Level Composition**: Each level is defined by a specific `mapAsset`, unique background music, and a target `enemyCount` that scales the difficulty. All levels now have specific background images, replacing a generic placeholder.

User Interface and Experience (UI/UX)

The game features a dedicated `UIScene` that runs in parallel with the `GameScene` to manage the heads-up display and other overlays.

- **HUD Elements**: The UI Scene is responsible for rendering:
    - Health and Mana bars, which update dynamically via the `EventBus`.
    - The inventory grid overlay.
    - The dialogue box for NPC interactions.
- **Menus and Scenes**: The game includes a polished set of menu scenes for a complete user experience:
    - **Main Menu**: The entry point with "Play," "Settings," and "Controls" options.

- **Settings Scene**: Allows players to toggle Music and SFX.
- **Controls Scene**: A visual guide displaying all keybindings.
- **Pause Scene**: Accessed via the `ESC` key, it pauses the physics engine and provides "Resume," "Settings," or "Quit" options.
- **Winning Scene**: Triggers on level or quest completion, offering "Replay" or "Menu" options.

Development and Extensibility

The data-driven architecture is designed to make adding new content straightforward.

Adding a New Item

1. Open `src/data/items.json`.
2. Add a new JSON object with a unique key.
3. Define its properties, including `name`, `sprite`, `type`, and `effect`.

```
"super_potion": {
  "name": "Super Potion",
  "sprite": "item_potion_purple",
  "frame": 2,
  "type": "CONSUMABLE",
  "effect": { "type": "heal", "stat": "hp", "value": 100 }
}
```

Adding a New Enemy

1. Open `src/data/enemies.json` and add a new entry.
2. Define its stats (hp, `damage`) and assign a `lootTable`.
3. If unique behavior is required, extend the `Enemy.js` class or add a new state to its state machine.
4. Place the enemy in a Tiled map using a Point object with `type: Enemy` and a custom property `id` matching the JSON key.

```
"fire_imp": {
  "name": "Fire Imp",
  "hp": 60,
  "damage": 12,
  "sprite": "imp_red"
}
```

Creating a New Level

1. Create a map in Tiled and export it as a JSON file to `assets/maps`.
2. Register the new map asset in the game's asset loader (`BootScene.js`).
3. Add a new entry to `src/data/levels.json` defining its `name`, `mapAsset`, `music`, and other properties.

```
"volcano": {
  "name": "Volcano",
  "nextLevel": "sky_temple",
  "enemyCount": 30
}
```

Controls

Key(s)	Action
WASD / Arrow Keys	Move Character
SPACE	Attack
I	Toggle Inventory
1-5	Use Quick Slot Item
E	Interact with NPCs
R	Retry Level (on Game Over Screen)
ESC	Pause Game

Future Roadmap

The following tasks are pending on the project roadmap:

- **Save/Load System:** Implement a system to serialize full player inventory and level progress.
- **Boss Mechanics:** Develop complex AI patterns and unique mechanics for the "Null Shard" boss.
- **Audio Polish:** Add unique sound effects for different weapon types and actions.
- **Mobile Support:** Implement an overlay for touch controls.