Project Report – Roulette Mini Game

This report summarizes the design decisions, implementation details, architectural choices, and optimization techniques I used while developing the modular Roulette mini-game based on the assignment requirements.
Throughout the development process, my priority was to create a clean, extendable, and organized structure while demonstrating solid understanding of Unity workflows, UI systems, data-driven setups, and animation management.

1. Architectural Approach & Design Decisions

From the beginning, my goal was to structure the project in a way that can easily support multiple roulette variations. Because of this, I created a shared ScriptableObject base class named **RouletteVariationBase**.

Through this base:

- Slot reward data
- Visual appearance configurations
- Layout information
- Spin behavior, delays, and loop settings

are all controlled from a single centralized asset.
This approach makes future extensions extremely easy, as adding a new variation simply means creating another ScriptableObject with different visuals, slot positions, or spin animation behavior.

Slot & Layout System

Slots are defined using:

- **RouletteLayoutData** (positions, spacing, prefab)
- **SlotAppearanceConfig** (normal, glow, reward, completed visuals)
- **SlotData** (reward information)

This data-driven approach allowed me to adjust visuals and layouts directly in the Unity Editor without modifying any code. It also kept the scene clean and improved iteration speed.

2. Service Architecture & Dependency Management

To keep the gameplay logic clean and independent, I introduced a small service layer.

Services Implemented

- **WalletService**
  Manages the player's wallet and persists the value using PlayerPrefs.
- **RewardManager**
  Handles reward animations, pooling operations, and updating the wallet after animations finish.
- **ServiceLocator**
  A lightweight way to resolve services without tight coupling.

For a small project like this, Service Locator provides enough flexibility. In larger productions I would consider Zenject or Unity’s new dependency systems, but here it helped keep the code minimal and easy to follow.

3. UI, Animation & Tween Management

All UI animations were created using **DOTween**, and I centralized the animation logic inside **UITweenUtils** to keep the code reusable and consistent.

Implemented UI/FX Animations

- A soft bounce effect when a slot becomes selected
- Trail glow state during the spin
- Reward icons spawning randomly and flying toward the wallet
- Wallet icon bouncing when rewards reach it
- A small pop-up that shows the collected reward
- Entrance animation for the roulette board

By managing all tweens through a single utility class, I kept the system clean and avoided repeating animation code.

4. Event System

To keep UI and gameplay separated, I used a lightweight **EventBus**.

Events Used

- **ChangedWalletEvent**
- **RewardWonEvent**

For example, WalletUI simply listens to wallet updates — gameplay never directly touches the UI layer.
This reduces dependencies and keeps responsibilities well-defined.

5. Game Flow & State Management

To make the spin–reward–popup loop predictable and clean, I added a simple **GameStateManager**.

States

- Idle
- Spinning
- Popup
- Rewarding

These states help control:

- When the spin button should be disabled
- When reward animations are allowed to play
- Ensuring popup and reward animations don’t overlap
- Proper sequencing inside RewardSequence

Waiting on state changes inside the coroutine made the gameplay flow much easier to manage and understand.

6. Optimizations

Although the project is small, I still applied a couple of performance-focused techniques.

Object Pooling

Reward icons spawn frequently.
Using instantiate/destroy repeatedly would cause GC allocations and performance drops, especially on mobile.
I created a **RewardImagePool** to reuse spawned reward objects efficiently.

Sprite Atlas

To minimize draw calls and benefit from Unity's batching system, I used a sprite atlas for UI elements.
This helps keep mobile performance stable and reduces rendering overhead.