Mini Project Completion Guide

Steal-a-Brainrot Style Character Buyer + Base Earner Loop (Roblox)

This document describes a complete mini project that demonstrates core Roblox game programming skills: client/server separation, purchases, replication, movement, earnings over time, claiming, saving, and anti-exploit validation. The end goal is a playable prototype that can be expanded into a full "Steal a Brainrot"-style experience.

Project Summary

Characters move in a straight line from left to right in a public shop lane. Each character has a Name, Price, and Earnings Per Second (EPS). Players buy characters from a central purchase zone. Purchased characters then travel to the player's base. Once they arrive, they begin generating money per second. Players must actively claim the generated money to add it to their spendable balance and buy stronger characters.

1. Core Gameplay Loop (Must Work)

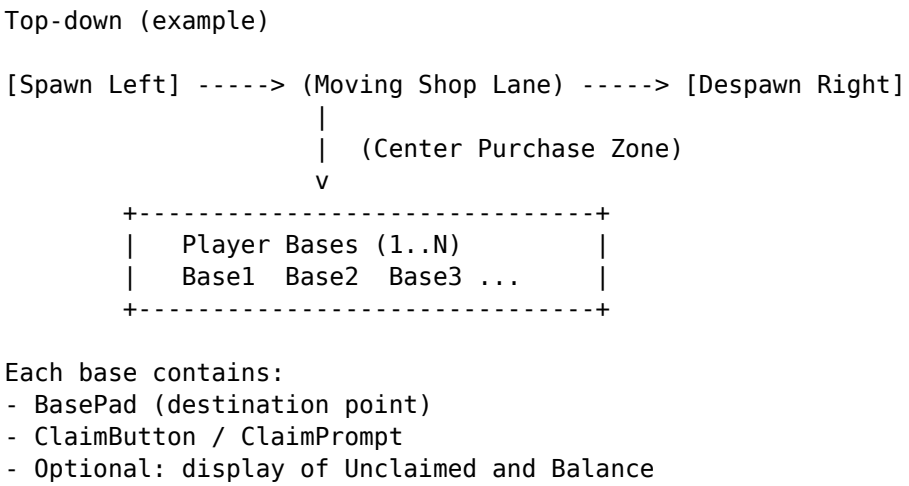
- Browse lane: Characters continuously move left -> right on a visible shop lane.
- Inspect: Each character clearly displays name, price, and EPS (e.g., BillboardGui above head).
- Purchase: Player interacts at the center zone to buy one available character (server validates).
- Deliver: The bought character leaves the lane and moves toward the player's base.
- Earn: Once the character reaches the base, it generates money into an Unclaimed pool each second.
- Claim: Player presses Claim to move Unclaimed -> Balance.
- Progress: Balance is used to purchase better characters with higher EPS.

2. Completion Criteria (Definition of Done)

- A new player can join, see the moving lane, and buy at least 5 character tiers.
- Bought characters reliably walk/move to the correct base and start generating money.
- Claiming works and updates UI immediately.
- Purchases are server-authoritative (no client-side money editing).
- Basic saving works: Balance, owned character tiers/slots, and Unclaimed state restore on rejoin.
- Code is modular (ModuleScripts) and easy to extend with more characters/mutations later.

3. World Layout (Recommended)

Keep the prototype simple: one main lane, one purchase zone, and multiple player bases in a row. All movement is deterministic and easy to debug.



Base Requirements

- BasePad: A fixed point where purchased characters stop and become earners.
- Claim interaction: ProximityPrompt or button that triggers a server claim request.
- Ownership: Each base is assigned to one player (server stores mapping).

4. Character Data Model

All characters should come from a configuration table. Never hard-code price/EPS in scripts. This lets you add new brainrots instantly and rebalance without touching logic.

Tier	Name	Price	EPS	ModelKey
1	Tiny Brainrot	0	1	Brainrot_T1
2	Better Brainrot	25	3	Brainrot_T2
3	Epic Brainrot	150	10	Brainrot_T3
4	Mythic Brainrot	800	40	Brainrot_T4
5	Legend Brainrot	3500	120	Brainrot_T5

Required per-character fields

- tier (number): progression order
- name (string): display name
- price (number): cost to purchase from shop
- eps (number): earnings per second once in base
- modelKey (string): key to load the correct model template

5. System Architecture (How to Build It Cleanly)

Use server modules for authoritative state and a small client controller for UI. The prototype should be readable and extendable.

Suggested folder structure

```
ReplicatedStorage
  Shared
    Remotes
    Config
      BrainrotCharacters.lua
    Util

ServerScriptService
  Server
    Services
      CurrencyService.lua
      ShopLaneService.lua
      PurchaseService.lua
      BaseService.lua
      SavingService.lua

StarterPlayer
  StarterPlayerScripts
    Client
      UIController.lua
      ShopLaneClient.lua (optional visuals)

StarterGui
  UI
    MainHUD (Balance, Unclaimed, Claim button, Selected character panel)
```

Module responsibilities

- CurrencyService: Holds Balance and Unclaimed for each player. Only server edits money.
- ShopLaneService: Spawns characters onto the lane, moves them left -> right, and cycles/despawns.
- PurchaseService: Validates purchase requests, deducts Balance, assigns character to player.
- BaseService: Assigns bases to players and provides destination points for delivery.
- SavingService: Loads/saves Balance, Unclaimed, and owned earners/tiers.

6. Client vs Server Rules (Applied to This Project)

- Client can request: BuyCharacter, Claim.
- Server decides: which character is purchasable, whether the player has enough money, and how much they earn.
- Client UI is driven by replicated Attributes or server events (Balance, Unclaimed, owned counts).
- Never let the client send "new balance" or "earnings amount".

Recommended RemoteEvents

- Remotes/RequestBuy (RemoteEvent): client -> server (characterId or laneSlotId)
- Remotes/RequestClaim (RemoteEvent): client -> server
- Remotes/StateUpdate (RemoteEvent): server -> client (balance/unclaimed updates if not using Attributes)

7. Movement and Delivery Implementation

Keep movement simple and deterministic. You can use either Tween-based movement or direct CFrame stepping on the server. Avoid per-character RunService connections; use a single loop that updates all lane movers.

Shop lane movement (left -> right)

- Spawn character templates from ServerStorage (or ReplicatedStorage if clients need them).
- Place each character onto a lane path at start X, then move toward end X at a constant speed.
- When a character reaches the end, despawn and replace it with a new one.

Purchase delivery to base

- On purchase: detach the character from lane logic and mark it as "Owned".
- Compute a destination point (the player's BasePad position).
- Move the character to the base using a simple straight line tween or pathing-free walk.
- When within a small radius, stop movement and snap to final CFrame.

```
-- Server pseudo-code: moving a model to a destination
local function MoveModelTo(model, destinationPos, speed)
    local primary = model.PrimaryPart
    local startPos = primary.Position
    local dist = (destinationPos - startPos).Magnitude
    local time = math.max(dist / speed, 0.05)

    -- Option A: Tween the PrimaryPart CFrame (simple and reliable)
    -- TweenService:Create(primary, TweenInfo.new(time, Enum.EasingStyle.Linear), {
    --     CFrame = CFrame.new(destinationPos)
    -- }):Play()
end
```

Arrival detection

Do not rely only on Tween.Completed (it can be cancelled). Always verify arrival by distance checks and then finalize state (set anchored, start earning).

8. Earnings, Unclaimed, and Claiming

Use two money values: Balance (spendable) and Unclaimed (generated by earners, not yet claimed). This creates the core "claim" mechanic that makes the loop feel active.

Earning logic

- Each owned character contributes EPS to the player once at base.
- Every 1 second (or 0.5s), add totalEPS to player's Unclaimed.
- Clamp to prevent absurd values (optional) and store as integers to avoid float drift.

Claim logic

- When player claims: move Unclaimed -> Balance (server-only).
- Reset Unclaimed to 0.

- Notify client to update UI (Attributes or a StateUpdate remote).

9. UI/UX Requirements (Simple but Addictive)

- Balance display: always visible.
- Unclaimed display: always visible and animates when increasing.
- Claim button: clearly clickable with cooldown feedback.
- Shop lane info: each character shows Name, Price, EPS.
- Purchase feedback: success/fail toast ("Not enough money", etc.).

BillboardGui layout (above each shop character)

```
Name:  'Epic Brainrot'
Price: $150
EPS:   +10/s
```

10. Anti-Exploit Validation (Required)

- Validate purchase requests: lane slot exists, character is still available, player is close enough to purchase zone.
- Validate currency: player Balance >= price, price comes from server config only.
- Rate limit: prevent spam buy/claim (simple cooldown per player).
- Never accept EPS or price from client; client only sends an ID.

11. Saving (Minimum Viable)

- Save Balance and Unclaimed.
- Save owned characters at base (e.g., list of tiers or count per tier).
- On join: restore owned earners and restart EPS accumulation.
- Use pcall + retries; save on leave + periodic timer.

12. Testing Checklist

- Join with 2 players: confirm each gets their own base and purchases go to the correct base.
- Spam buy: ensure server cooldown stops rapid purchases and no negative money occurs.
- Disconnect while Unclaimed > 0: verify it loads correctly on rejoin (or design choice: reset).
- Server performance: ensure lane movement uses a single loop, not 50 separate Heartbeat connections.
- UI correctness: Balance and Unclaimed always match server state.

13. Stretch Goals (Optional Enhancements)

- Rarity / roll: lane characters have random rarity and special visuals.
- Mutations: apply a Highlight/Particles layer to purchased characters (no cloning).
- Steal mechanic: allow players to steal unclaimed money or steal characters (requires strong validation).
- Upgrades: faster delivery speed, extra base slots, higher claim multiplier.
- Offline earnings: grant catch-up based on time away (careful with exploits).

Final Deliverables

- A Roblox place file with the lane + bases + purchase zone.
- All server logic in ModuleScripts with clear APIs.
- Client UI that reads server state and sends only requests.
- Config file for characters (easy to add more brainrots).
- Basic saving implemented and tested.