# App Outline: Crypto Clicker - A Seamless Web3 Game

## 1. High-Level Concept

**Crypto Clicker** is a simple yet addictive idle/clicker game where players earn in-game currency by clicking. The core innovation is its **seamless in-game store**. Using the Base Account SDK, players can make instant, one-click purchases of power-ups with real crypto, without the constant interruption of wallet signature pop-ups. This is achieved by creating an app-specific **Sub Account** for the player and using a one-time **Spend Permission** to create a "spending budget".

**The Goal:** To demonstrate a frictionless Web3 user experience that feels as smooth as a traditional Web2 game. The game itself is the stage; the seamless transaction flow is the main feature.

## 2. Core Features & Functionality

### Gameplay Mechanics

* **Click to Earn:** A large, central on-screen element (e.g., a spinning coin) that the player clicks to earn an in-game, non-crypto currency called "Gems".
* **Gem Counter:** A visible display of the player's total Gems.
* **Upgrades:** A list of upgrades the player can buy *with Gems* to increase their earning rate.
  + **Example Upgrade 1:** "Stronger Clicks" - Permanently increases Gems earned per click.
  + **Example Upgrade 2:** "Auto-Miner Bot" - Automatically generates Gems per second, even when not clicking.

### Web3 Integration & Store

* **Connect Wallet Button:** The initial action for a user. This connects their main Base Account.
* **Sub Account Management:**
  + On first connect, the app will automatically create and associate a new, app-specific Sub Account for the player. This will be their "in-game wallet".
  + On subsequent visits, the app will automatically retrieve the existing Sub Account.
* **Spend Permission "Budget":**
  + The store will have a "Top Up Balance" or "Add Budget" button.
  + Clicking this prompts the user **only once** to sign a Spend Permission, allowing their Sub Account to pull a specific amount (e.g., $5.00 in USDC) from their main Base Account for in-game purchases.
  + The UI must clearly display the remaining budget from this permission.
* **One-Click In-Game Store:**
  + A section of the UI that sells temporary "Power-Ups" for real crypto (paid from the Spend Permission budget).
  + **Example Power-Up 1:** "Click Frenzy" (Cost: $0.10) - For 30 seconds, all clicks yield 10x Gems.
  + **Example Power-Up 2:** "Gem Shower" (Cost: $0.25) - Instantly grants 5,000 Gems.
  + When a player with a budget clicks "Buy", the purchase is instant. No signature, no pop-up. The funds are transferred from their main account via the Sub Account, and the power-up is activated.

## 3. User Interface (UI) Layout

The application should be a single-page interface.

* **Top Bar:**
  + Title: "Crypto Clicker"
  + Status Indicator: Shows connection status, current Gem balance, and remaining Spend Permission budget (e.g., "Budget: $4.90").
  + Connect/Disconnect Wallet Button.
* **Main Game Area (Left Side):**
  + The large, clickable coin element.
  + A visual effect should occur on each click.
* **Upgrades & Store Area (Right Side):**
  + A tabbed interface with two sections: "Upgrades" and "Store".
  + **Upgrades Tab:** A scrollable list of upgrades purchasable with Gems. Each item should show its name, cost in Gems, effect, and a "Buy" button.
  + **Store Tab:** A list of power-ups purchasable with real money. Each item should show its name, cost in USD, effect, and a "Buy Now (1-Click)" button. This section must also contain the prominent "Top Up Balance" button.
* **Account Info Display:**
  + A small, non-intrusive section at the bottom showing the connected Main Account address and the managed Sub Account address.

## 4. User Flow

1. **New User:** Clicks "Connect Wallet". The Base Account connection modal appears.
2. User connects their wallet. The app detects it's a new player and checks the database for existing data.
3. A modal appears: "Welcome! We're creating a secure in-game wallet for you." The app requests the signature to create the Sub Account (if one doesn't already exist).
4. The user is now on the main game screen. If they had prior progress, it's loaded from the database. They can start clicking to earn Gems.
5. User navigates to the "Store" and clicks "Buy" on a power-up. A message appears: "You need a budget first!"
6. User clicks "Top Up Balance". A clear, friendly modal explains: "To enable one-click purchases, please approve a one-time spending budget of $5.00 for your in-game wallet."
7. The Base Account modal appears, asking the user to sign the EIP-712 Spend Permission.
8. Once approved, the UI updates to show "Budget: $5.00".
9. Now, the user clicks "Buy" on the "$0.10 Click Frenzy" power-up.
10. The transaction happens instantly in the background. The user's budget display updates to "$4.90", and the "Click Frenzy" power-up activates in the game. **No new signature is required.**
11. The user can continue making one-click purchases until their budget runs out.

## 5. Technical Stack & Implementation Details

* **Framework:** React.js (using Vite or Create React App).
* **Styling:** Tailwind CSS for a clean, modern, and responsive layout.
* **Web3 Library:** Use the @base-org/account SDK.
* **Blockchain:** Target a Base testnet like Base Sepolia for development and demonstration.
* **Database & Persistence:**
  + **Service:** Firebase Firestore (NoSQL Cloud Database).
  + **Data Structure:** Create a users collection. Each document in this collection will be identified by the player's main wallet address (e.g., users/{user\_wallet\_address}).
  + **Schema:** The document for each user should store their game state, for example:  
    {  
     "gems": 1500,  
     "upgrades": {  
     "strongerClicksLevel": 2,  
     "autoMinerLevel": 1  
     },  
     "lastLogin": "timestamp"  
    }
  + **Functionality:**
    - **On Connect:** After the user connects their wallet, query Firestore for a document with their address as the ID. If it exists, load the game state from that document.
    - **On State Change:** Any time the user's Gem count or upgrade levels change, save the updated state back to their document in Firestore.
* **Logic:**
  + Use useState and useEffect hooks extensively to manage game state (Gem count, upgrades) and wallet state (provider, addresses, budget).
  + All Web3 logic (SDK initialization, connecting, creating/getting Sub Account, requesting Spend Permission, sending transactions) should be abstracted into a dedicated utility file or a custom hook for cleanliness.
  + The transaction for purchasing a power-up should be a simple transfer of a stablecoin (like USDC on Base Sepolia) from the user's account to a designated treasury address. Use the wallet\_sendCalls method from the Sub Account address.