docs.cartridge.gg — Snapshot (- -)

> *Clean documentation content extracted from sitemap.*

Cartridge Documentation

Source: https://docs.cartridge.gg/ vocs-content

High Performance Tooling and Infrastructure for Provable Games and Applications

[Play with

Providing seamless player onboarding with self-custodial embedded wallets with Passkeys, Session Tokens, Paymaster and more. Start playing games in seconds!](/controller/overview)[Scale with

Horizontally scalable execution sharding for ephemeral and persistent rollups. Providing low latency execution contexts with fixed costs.](/slot/getting-started)[Discover with

Bring all your favorite onchain games together in one central hub where players and developers connect.](/arcade/overview)

Arcade Overview – Cartridge Documentation

Source: https://docs.cartridge.gg/arcade/overview

TL;DR: Arcade is:

- A unified hub for all your favorite onchain games

- A frictionless bridge between developers and players
- A permissionless platform for game registration and publishing
- Designed to enhance discoverability, engagement, and player experience
- Leverage NFTs for ownership and control of your games and editions

Key Features

🎮 Register a Game

Game studios can register a game freely, without needing permission. When creating a game for the first time, you'll be asked to provide its metadata and define at least one Game Edition.

When you register a game or an edition, an NFT representing its ownership is minted. This NFT grants you admin rights over the game or edition. You can update the metadata of your game or any edition at any time — including name, description, icon, image gallery, video, and more.

🚀 Publish a Game

Once your game is registered, you can publish it to request a review. After approval, the game will be whitelisted and made publicly visible to all users on the platform.

🧩 Create and update Game Editions

You can create new editions of your game at any time — each edition gets its own ownership NFT. As the game owner, you control which editions are visible or hidden. As the edition owner, you can choose to publish or hide your edition from public view.

🚀 Publish and whitelist an Edition

As the edition owner you have the ability to publish your edition once created. Once published, the game owner has the ability to whitelist you Edition to make it public.

Arcade Setup – Cartridge Documentation

Source: https://docs.cartridge.gg/arcade/setup

Torii Configuration

To provide a rich user experience in Arcade, we recommend enhancing your Torii configuration to enable live activity feeds, asset indexing, and leaderboards.

⚡ Activity Feed

Display live player activity on your Arcade edition page.

```
[indexing] transactions = true
```

💄 Player Asset Indexing

Index and display custom in-game assets tied to your players.

```
[indexing] contracts = [     "ERC20:0x1234...5678",
"ERC721:0x1234...5678", ]
```

✨ Leaderboard Integration

Enable live leaderboards based on progression events or achievements (if implemented).

```
[sql] historical = ["<YOUR-NAMESPACE>-TrophyProgression"]
```

Only add TrophyProgression if your game emits progression events through achievements.

Achievements – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/achievements

The Cartridge Achievements is a system for games to reward players for completing achievements.

Key Features

- **Packages**: Games can define achievements thanks to the provided packages.
- **Rewards**: Games can reward players with Cartridge points for completing achievements.
- **Profile**: Players can view their achievements and scores whitout leaving the game.

Benefits for Game Developers

- **Simplicity**: Easy integration with existing Starknet smart contracts and Dojo.
- **Cost-effectiveness**: Achievements are events based, no additional storage is required.
- **Performance** (coming soon): Plugin attached to Torii to improve the performances of the achievements computation.

How It Works?

For detailed implementation and usage, refer to the GitHub repository.

Creation

The game world describes the achievements and the corresponding tasks to unlock them. Each achievement is defined by (not exhaustive) a unique `identifier`, a `title`, a `description` and a set of `tasks`. Each task is defined by an `identifier`, a `total` and a `description`. The completion of a task is done when enough progression has been made by a player regarding a specific task. The achievement is completed when all included tasks are completed.

Progression

The progression of each individual task is done by the game by emitting events associated to a `task` and a `player`. Each progression provides a `counter` to add to the player progression. A task completion is the sum of all the progression events emitted for a specific `task` (defined by the `identifier`).

Integration

The status of the achievement is computed off-chain by the controller, it starts when the controller is initialized on the client.

Controller Configuration – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/configuration

Controller provides several configuration options related to chains, sessions, and theming.

ControllerOptions

``` export type Chain = { rpcUrl: string; };

export type ControllerOptions = { // Chain configuration chains?: Chain[]; // Custom RPC endpoints (takes precedence over default chains) defaultChainId?: string; // Default chain to use (hex encoded). If using Starknet React, this gets overridden by the same param in StarknetConfig

```
// Session options
policies?: SessionPolicies; // Optional: Session policies for
pre-approved transactions
propagateSessionErrors?: boolean; // Propagate transaction
errors back to caller

// Customization options
preset?: string; // Preset name for custom themes and
verified policies
slot?: string; // Slot project name for custom indexing
```

}; ```

# Chain Configuration

Controller provides default Cartridge RPC endpoints for Starknet mainnet and sepolia networks:

- `https://api.cartridge.gg/x/starknet/mainnet`
- `https://api.cartridge.gg/x/starknet/sepolia`

When you provide custom chains via the `chains` option, they take precedence over the default Cartridge chains if they specify the same network. This allows you to:

- Use custom RPC endpoints for mainnet or sepolia
- Add support for additional networks (like Slot katana instances)
- Override default chain configurations

**Example:**

```
const controller = new Controller({   chains: [{ rpcUrl:
"https://api.cartridge.gg/x/my-game/sepolia" }, // Overrides
default sepolia   { rpcUrl: "http://localhost:5050" }, // Adds
```
```

```
local development chain   ],   chainId:
constants.StarknetChainId.SN_SEPOLIA, });
```

Configuration Categories

The configuration options are organized into several categories:

- **Chain Options**: Core network configuration and chain settings
- **Session Options**: Session policies and transaction-related settings
- **Customization Options**: Presets for themes and verified policies, Slot for custom indexing

When to Use Policies

Policies are optional in Cartridge Controller. Choose based on your application's needs:

Use Policies When:

- Building games that need frequent, seamless transactions
- You want gasless transactions via Cartridge Paymaster
- Users should not be interrupted with approval prompts during gameplay
- You need session-based authorization for better UX

Skip Policies When:

- Building simple applications with occasional transactions
- Manual approval for each transaction is acceptable
- You don't need gasless transaction capabilities
- You want minimal setup complexity

``` // Without policies - simple setup, manual approvals const simpleController = new Controller();

// With policies - session-based, gasless transactions const sessionController = new Controller({ policies: { // ... policy definitions } }); ```

---

## Controller Getting Started – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/getting-started
```

Controller implements a standard StarkNet account interface and can be seamlessly integrated into your application like any other wallet.

Quick Start

The fastest way to get started is to install the controller package and connect to Cartridge:

``` import Controller from "@cartridge/controller";

const controller = new Controller({}); const account = await controller.connect();

// You're ready to execute transactions! ```

## Installation

npmpnpmyarnbun

npm

```
npm install @cartridge/controller starknet
```

## Basic Usage

Here's a simple example of how to initialize and use the controller:

``` import Controller from "@cartridge/controller";

// Initialize the controller without policies // This requires manual approval for each transaction const controller = new Controller();

// Connect to get an account instance const account = await controller.connect();

// Execute transactions - user will see approval dialog const tx = await account.execute([ { contractAddress: "  x...", entrypoint: "my_function", calldata: ["  x  ", "  x  "], }]); ```

> ***Note:*** *When no policies are provided, each transaction requires manual user approval through the Cartridge interface. This is suitable for simple applications or testing, but games typically benefit from using session policies for a smoother experience.*

## Configuration with Session Policies

For games that need gasless transactions and session management:

``` import Controller from "@cartridge/controller";

const controller = new Controller({ policies: { contracts: { // Your game contract " x ...": { name: "My Game Contract", methods: [ { name: "move*player", entrypoint: "move*player", description: "Move player character", }, { name: "attack", entrypoint: "attack", description: "Attack enemy", }, ], }, }, }, }); ```

## Usage with Starknet React

``` import React from "react"; import { sepolia, mainnet } from "@starknet-react/chains"; import { StarknetConfig, jsonRpcProvider, starkscan, } from "@starknet-react/core"; import ControllerConnector from "@cartridge/connector/controller";

// Create the connector outside the component to avoid recreation on renders const connector = new ControllerConnector();

// Configure the JSON RPC provider const provider = jsonRpcProvider({ rpc: (chain) => { switch (chain) { case mainnet: return { nodeUrl: "https://api.cartridge.gg/x/starknet/mainnet" }; case sepolia: default: return { nodeUrl: "https://api.cartridge.gg/x/starknet/sepolia" }; } }, });

export function StarknetProvider({ children }: { children: React.ReactNode }) { return ( <StarknetConfig defaultChainId={sepolia.id} chains={[mainnet, sepolia]} provider={provider} connectors={[connector]} explorer={starkscan} > {children} ); } ```

## Examples

For more detailed examples of how to use Cartridge Controller in different environments, check out our integration guides:

. React

- Integration with `starknet-react`
- Hooks and components
- State management

. Svelte

- Svelte stores and reactivity
- Component lifecycle

    - Event handling

. Rust

    - Native integration
    - Error handling
    - Async operations

Each guide provides comprehensive examples and best practices for integrating Cartridge Controller in your preferred environment.

## Next Steps

- Learn about Session Keys
- Set up Multiple Signers for backup authentication
- Customize your Controller
- Set up Usernames
- Configure Paymaster

---

## Controller Inventory Management – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/inventory

Cartridge Controller provides Inventory modal to manage account assets ( `ERC-20` , `ERC-721` ).

## Configure tokens

By default, commonly used tokens are indexed and automatically shown. Full list of default tokens are listed in `torii-config/public-tokens/mainnet.tom` . This list can be extended by configuring Torii hosted on Slot.

### Configure additional token to index

```

torii-config.toml

[indexing] contracts = ["erc :", "erc :"] ```
```

Create or update Torii instance on Slot

```
slot d create <project> torii --config <path/to/torii-config.toml>
```

Configure Controller

Provide Slot project name to `ControllerOptions`.

``` const controller = new Controller({ slot: "" });

// or via connector const connector = new CartridgeConnector({ slot: "" }) ```

### Open Inventory modal

```
controller.openProfile("inventory");
```

---

# Controller Overview – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/overview

### TL;DR: Cartridge Controller is:

- A gaming-focused smart contract wallet for Starknet
- Makes Web gaming accessible and fun via Session Keys and gasless transactions
- Handles seamless player onboarding with Passkey authentication
- Provides identity, achievements, and customization features for games
- Compatible with popular frameworks like Starknet React and can be integrated across platforms

# Key Features

### Simple & Secure

- Passwordless authentication using Passkeys for one-click onboarding
- Multi-signer support with Passkeys, social login (Google, Discord), and external wallets (MetaMask, Rabby, WalletConnect)
- Self-custodial embedded wallets that put players in control
- Built-in security features to protect player assets
```

Designed for Fun

- Session keys eliminate transaction popups during gameplay
- Secure transaction delegation lets games submit actions on behalf of players
- Free transactions through the Cartridge Paymaster so players focus on playing

Customizable

- Flexible architecture adapts to your game's specific requirements
- Full theme customization to match your game's branding
- Dynamic UI components for displaying game assets, quests and achievements
- Extensible plugin system for adding custom functionality

Identity and Reputation

- Universal player identity that works across all Cartridge-enabled games
- Built-in achievement system for tracking player accomplishments
- Reputation system that grows as players engage with games
- Social features to connect players and build communities

Controller Passkey Support – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/passkey-support

Credential Security

Controller leverages Passkeys to securely provision credentials for signing blockchain transactions. Passkeys can be generated by:

- Platform authenticators (Face ID, Touch ID)
- Password managers (Bitwarden, Password)

Platform Support

Passkeys are generally well supported across modern platforms. You can use them with:

- Device authenticators directly
- Mobile device pairing via QR code flow

If you device does not natively support them, there are several password managers that support Passkey creation and management:

- Bitwarden (free)
- 1Password
- Dashlane

When using a password manager, be sure to install the browser extension as well.

Backup Solutions

Passkey backup is handled differently depending on your platform or password manager:

Apple Devices

Passkeys are backed up with your keychain in iCloud. Learn more

Android Devices

Passkeys are backed up with your Google account. Learn more

Windows Devices

Passkeys can be created and managed as part of your Windows account. Learn more

Multi-Signer Support

Passkeys can be used as part of Controller's multi-signer functionality, allowing you to add multiple authentication methods to your account. This provides backup access and additional security options.

Learn more about adding and managing multiple signers in the Signer Management guide.

Controller Presets – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/presets

This guide provides a comprehensive overview of how to create and apply custom themes, provide verified session policies, and configure Apple App Site Association for iOS integration with the Cartridge Controller.

Creating a Theme

To create a theme, teams should commit their theme config to the `configs` folder in `@cartridge/presets` with the icon and banner included.

```
{   "origin": "https://flippyflop.gg",   "theme":
{     "colors": {       "primary": "#F38332"     },     "cover":
"cover.png",     "icon": "icon.png",     "name":
"FlippyFlop"   } }
```

See an example pull request `here`

Verified Sessions

Session Policies can be provided in the preset configuration, providing a smoother experience for your users. In order to submit verified policies, create a commit with them to your applications `config.json` in `@cartridge/presets` .

For an example, see dope-wars:

```
{   "origin": "dopewars.game",   "chains": {     "SN_MAIN":
{       "policies": {         "contracts":
{           "0x051Fea4450Da9D6aeE758BDEbA88B2f665bCbf549D2C61421
AA724E9AC0Ced8F": {             "name": "VRF
Provider",             "description": "Provides verifiable
random functions",             "methods":
[               {                 "name": "Request
Random",                 "description": "Request a random
number",                 "entrypoint":
"request_random"               }             ]           }
  }       }     }   }, }
```

Apple App Site Association

The Apple App Site Association (AASA) configuration enables iOS app integration with Cartridge Controller, allowing for usage of Web Credentials (Passkeys) in native applications.

Configuration

To add your iOS app to the AASA file, include the `apple-app-site-association` section in your game's `config.json` :

JSON Configuration

```
{   "apple-app-site-association": {     "webcredentials":
{       "apps": ["ABCDE12345.com.example.yourgame"]     }   }, }
```

Sessions and Policies – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/sessions

Cartridge Controller supports session-based authorization and policy-based transaction approvals. When policies are pre-approved by the user, games can execute transactions seamlessly without requesting approval for each interaction, creating a smooth gaming experience.

How Sessions Work

. **Policy Definition**: Games define which contract methods they need to call
. **User Approval**: Users approve these policies once during initial connection
. **Session Creation**: Controller creates a session with approved transaction permissions
. **Gasless Execution**: Games can execute approved transactions without user prompts
. **Paymaster Integration**: Transactions can be sponsored through Cartridge Paymaster

Transactions Without Policies

Cartridge Controller can execute transactions **without** defining policies. When no policies are provided:

• Each transaction requires manual user approval via the Cartridge interface
• Users will see a confirmation screen for every transaction
• No gasless transactions or paymaster integration
• Suitable for simple applications that don't need session-based authorization

``` // Controller without policies - requires manual approval for each transaction const controller = new Controller(); const account = await controller.connect();

// This will prompt the user for approval const tx = await account.execute([ { contractAddress: "  x  ...", entrypoint: "transfer", calldata: ["  x  ...", "  "], }]); ```
```

Sessions vs. Manual Approval

| Feature | With Policies (Sessions) | Without Policies (Manual) | | --- | --- | --- | | Transaction Approval | Pre-approved via policies | Manual approval each time | | User Experience | Seamless gameplay | Confirmation prompts | | Gasless Transactions | Yes (via Paymaster) | No | | Setup Complexity | Higher (policy definition) | Lower (basic setup) | | Best For | Games, frequent transactions | Simple apps, occasional transactions |

Session Options

```
export type SessionOptions = {   rpc: string;                // RPC endpoint URL   chainId: string;            // Chain ID for the session   policies: SessionPolicies;  // Approved transaction policies   redirectUrl: string;        // URL to redirect after registration };
```

Defining Policies

Policies allow your application to define permissions that can be pre-approved by the user:

``` type SessionPolicies = { contracts: { [address: string]: ContractPolicy; // Contract interaction policies }; messages?: TypedDataMessage[]; // Optional signed message policies };

type ContractPolicy = { name?: string; // Human-readable name of the contract description?: string; // Description of the contract methods: Method[]; // Allowed contract methods };

type ContractMethod = { name: string; // Method name entrypoint: string; // Contract method entrypoint description?: string; // Optional method description };

type SignMessagePolicy = TypedDataPolicy & { name?: string; // Human-readable name of the policy description?: string; // Description of the policy };

type TypedDataPolicy = { types: Record<string, StarknetType[]>; primaryType: string; domain: StarknetDomain; }; ```
```

Usage Examples

Contract Interaction Policies Example

``` const policies: SessionPolicies = { contracts:
{ " x ed a c f c e eaf b bd e a c e a edf e fc ac ddc ": { name: "Pillage", description: "Allows you to raid a structure and pillage resources", methods: [ { name: "Battle Pillage", description: "Pillage a structure", entrypoint: "battle_pillage" } ] },
" x f aa fd d ada ee a df da f ccb fabfebc ": { name: "Battle contract", description: "Required to engage in battles", methods: [ { name: "Battle Start", description: "Start a battle", entrypoint: "battle*start" }, { name: "Battle Join", description: "Join a battle", entrypoint: "battle*join" }, { name: "Battle Leave", description: "Leave a battle", entrypoint: "battle_leave" }, ] }, // Include other contracts as needed } };

// Using the controller directly const controller = new Controller({ policies, // other options });

// Using starknet-react connector const connector = new CartridgeConnector({ policies, // other options }); ```

## Signed Message Policy Example

Signed Message policies allow the application to sign a typed message without manual approval from the user.

```
 const policies: SessionPolicies = {   messages:
[{   name: "Eternum Message Signing",
description: "Allows signing messages for Eternum",   types:
{   StarknetDomain: [{ name: "name", type:
"shortstring" },   { name: "version", type:
"shortstring" },   { name: "chainId", type:
"shortstring" },   { name: "revision", type:
"shortstring" }   ],   "s0_eternum-Message":
[{ name: "identity", type:
"ContractAddress" },   { name: "channel", type:
"shortstring" },   { name: "content", type:
"string" },   { name: "timestamp", type:
"felt" },   { name: "salt", type:
"felt" }   ]   },   primaryType: "s0_eternum-
Message",   domain: {   name: "Eternum",
```
```

```
version: "1",          chainId: "SN_MAIN",          revision:
"1"        }      }    ] };
```

Verified Sessions

Verified session policies provide a better user experience by attesting to the validity of a games session policy configuration, providing confidence to it's players.

Verified configs can be committed to the `configs` folder in `@cartridge/presets` .

Before they are merged, the team will need to collaborate with Cartridge to verify the policies.

Signer Management – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/signer-management

Cartridge Controller supports **multi-signer** functionality, allowing you to add multiple authentication methods to your account for enhanced security and convenience. This feature enables you to sign in using different methods while maintaining access to the same Controller account and assets.

Overview

Multi-signer support provides several benefits:

- **Backup Authentication**: Add multiple ways to access your account in case you lose access to your primary authentication method
- **Convenience**: Use different authentication methods depending on your device or context
- **Security**: Distribute access across multiple secure authentication methods
- **Flexibility**: Choose the authentication method that works best for each situation

Supported Signer Types

Controller supports three types of signers:

. Passkey (WebAuthn)

- **Biometric authentication** using Face ID, Touch ID, or hardware security keys
- **Platform-native** security with device-based credential storage

- **Cross-platform** compatibility with password managers like Bitwarden, Password
- See Passkey Support for detailed setup information

. Social Login

Google Login

- **Social authentication** using your Google account
- **Familiar experience** for users with existing Google accounts
- **Secure integration** via Turnkey wallet infrastructure
- Requires existing Google account

Discord Login

- **Social authentication** using your Discord account
- **Streamlined onboarding** for users already active in gaming communities
- **Secure integration** via Turnkey wallet infrastructure
- Requires existing Discord account

. External Wallets

- **MetaMask**: Popular browser extension wallet
- **Rabby**: Security-focused multi-chain wallet
- **WalletConnect**: Protocol supporting + wallets via QR code or deep linking
- Leverages existing wallet setup and seed phrases

Adding Signers

Accessing Signer Management

> ***Important**: The "Add Signer" functionality is currently disabled while under development. This feature will be re-enabled in a future update.*

. Connect to your Controller account using any existing authentication method
. Open the **Settings** panel within the Controller interface
. Navigate to the **Signer(s)** section

. ~~Click **Add Signer** to begin adding a new authentication method~~ (Currently disabled)

> ***Note**: When re-enabled, signer management will be available on **Mainnet only**. The "Add Signer" button will be disabled on testnet environments.*

Adding a Passkey

> ***Currently Disabled**: This functionality is temporarily unavailable while under development.*

~~ . In the Add Signer interface, select **Passkey**~~ ~~ . Your browser will prompt you to create a new Passkey using:~~ ~~- Device biometrics (Face ID, Touch ID, Windows Hello)~~ ~~- Hardware security key (USB, NFC, or Bluetooth)~~ ~~- Password manager (if configured for Passkey storage)~~ ~~ . Follow your device's authentication flow~~ ~~ . Once created, the Passkey will be added to your account~~

Adding Social Login

Adding Google Login

. Select **Google** from the signer options
. You'll be redirected to Google's OAuth authorization page
. Sign in to your Google account if not already logged in
. Authorize Cartridge Controller to access your Google identity
. The Google login will be linked to your Controller account

Adding Discord Login

> ***Currently Disabled**: This functionality is temporarily unavailable while under development.*

~~ . Select **Discord** from the signer options~~ ~~ . You'll be redirected to Discord's OAuth authorization page~~ ~~ . Sign in to your Discord account if not already

logged in~~ ~~ . Authorize Cartridge Controller to access your Discord identity~~ ~~ . The Discord login will be linked to your Controller account~~

Adding External Wallets

> ***Currently Disabled**: This functionality is temporarily unavailable while under development.*

~~ . Select **Wallet** to see external wallet options~~ ~~ . Choose from the supported wallet types:~~ ~~- **MetaMask**: Ensure MetaMask extension is installed and unlocked~~ ~~- **Rabby**: Ensure Rabby extension is installed and unlocked~~ ~~- **WalletConnect**: Use QR code or deep link to connect mobile/desktop wallets~~ ~~ . Follow the wallet-specific connection flow~~ ~~ . Sign the verification message to link the wallet to your account~~

Managing Existing Signers

Viewing Your Signers

The Signer(s) section displays all authentication methods associated with your account:

- **Signer type** with recognizable icons (fingerprint for Passkey, Discord logo, wallet icons)
- **Current status** indicating which signer you're currently using
- **Identifying information** such as wallet addresses (partially masked for privacy)

Signer Information Display

Each signer card shows:

- **Type**: Passkey, Google, Discord, MetaMask, Rabby, or WalletConnect
- **Status**: "(current)" label for the active authentication method
- **Identifier**: Shortened wallet address for external wallets, or authentication type for others

Switching Between Signers

When connecting to your Controller:

- The connection interface will show all available authentication methods
- Select any of your registered signers to authenticate
- Your account and assets remain the same regardless of which signer you use

Security Considerations

Best Practices

- **Multiple Backups**: Add at least - different signer types to ensure account recovery
- **Secure Storage**: For Passkeys, ensure your device backup (iCloud, Google) is secure
- **External Wallet Security**: Keep your wallet seed phrases secure and never share them
- **Regular Access**: Periodically test each authentication method to ensure they work

Account Recovery

If you lose access to your primary authentication method:

. Use any other registered signer to access your account
. Consider adding additional backup authentication methods
. Remove compromised signers if necessary (feature coming soon)

Current Limitations

- **Deletion**: Signer removal functionality is planned but not yet available
- **Mainnet Only**: Signer management is currently restricted to Mainnet
- **No Hierarchy**: All signers have equal access; there's no primary/secondary distinction

Troubleshooting

Common Issues

"Must be on Mainnet" message

- Signer management is only available on Mainnet
- Switch to Mainnet network to add or manage signers

Passkey creation fails

- Ensure your device supports WebAuthn/FIDO
- Try using a password manager with Passkey support
- Check that your browser is up to date

External wallet connection fails

- Verify the wallet extension is installed and unlocked
- Ensure you're on a supported network
- Check that the wallet isn't connected to another dApp

Google login issues

- Verify you're logged into Google in the same browser
- Check that third-party cookies are enabled
- Try clearing browser cache and cookies

Discord login issues

- Verify you're logged into Discord in the same browser
- Check that third-party cookies are enabled
- Try clearing browser cache and cookies

Getting Help

If you encounter issues with signer management:

- Check the Passkey Support guide for WebAuthn-specific help
- Verify your wallet setup in the respective wallet's documentation
- Ensure you're using a supported browser and have the latest wallet extensions installed

Next Steps

- Learn about Session Keys for gasless gaming transactions
- Explore Controller Configuration options
- Set up Usernames for your account

Username Lookup – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/usernames

A service for looking up usernames and addresses in the Cartridge ecosystem. You can use either the helper methods from the SDK or query the endpoint directly.

Direct API Access

The lookup endpoint can be accessed directly via HTTP POST:

```
curl -X POST \   -H "Content-Type: application/json" \   -d
'{"usernames": ["shinobi","sensei"]}' \   https://
api.cartridge.gg/accounts/lookup
```

Request Format

```
{   usernames?: string[];  // Look up addresses for usernames
addresses?: string[];  // Look up usernames for addresses }
```

Response Format

```
{   "results": [     {       "username": "shinobi",
"addresses": ["0x123..."]  // Array of addresses - future
support for multiple controllers/signers     },
{       "username": "sensei",       "addresses":
["0x456..."]     }   ] }
```

> ***Note**: The API response includes an array of addresses per username to support multiple controllers/signers in the future. Currently, the helper methods assume a : relationship and use only the first address.*

Helper Methods

For convenience, you can also use the helper methods which include caching:

```
npm install @cartridge/controller
```

``` import { lookupUsernames, lookupAddresses } from '@cartridge/controller';

// Look up addresses for usernames const userMap = await lookupUsernames(['shinobi']); console.log(userMap.get('shinobi')); // Returns address: '  x       ...'

// Look up usernames for addresses const addressMap = await lookupAddresses(['   x          ...']); console.log(addressMap.get('   x          ...')); // Returns username: 'shinobi' ```

```
lookupUsernames(usernames: string[]):
Promise<Map<string, string>>
```

- Fetches addresses for given usernames.
- Input: Array of usernames
- Returns: Map of username to address
- Caching: Results are automatically cached

```
lookupAddresses(addresses: string[]):
Promise<Map<string, string>>
```

- Fetches usernames for given addresses.
- Input: Array of addresses
- Returns: Map of address to username
- Caching: Results are automatically cached

Limitations and Rate Limiting

When using the lookup methods or directly via the API, be aware of the following limitations and rate limiting measures:

. **Maximum Items**: You can fetch up to items total in a single call, combining both addresses and usernames. For example:

  - addresses OR
  - usernames OR
  - Any combination (e.g., addresses + usernames)
```

. **Rate Limiting**: The API is rate-limited to requests per second to prevent overloading the server.

. **Address Format Requirements**:

  ◦ Addresses must be lowercase non-zero-padded hex
  ◦ The helper methods handle address formatting automatically

Error Handling

The lookup methods may throw errors in the following cases:

• If you provide more than addresses in a single call.
• If you exceed the rate limit of requests per second.
• If there are network issues or the API is unavailable.

Always wrap your calls to lookup methods in a try-catch block to handle potential errors gracefully.

Performance Considerations

To optimize performance when fetching usernames:

. Batch your requests: Instead of making multiple calls for individual addresses, group them into a single call (up to addresses).
. Utilize the built-in caching of the helper methods: Previously fetched usernames are cached, so subsequent requests for the same addresses will be faster.
. Be mindful of the rate limit: If you need to fetch usernames for more than addresses, implement your own throttling mechanism to avoid hitting the rate limit.

By following these guidelines, you can efficiently fetch and display usernames for controller addresses in your Cartridge-powered application.

Achievement Creation – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/achievements/creation

Getting Started

Emit events to define your achievements.

The package provides a way to define achievements leveraging Starknet components.

‵‵‵

[dojo::contract]

pub mod Actions { use achievement::components::achievable::AchievableComponent; use achievement::types::task::{Task, TaskTrait}; component!(path: AchievableComponent, storage: achievable, event: AchievableEvent); impl AchievableInternalImpl = AchievableComponent::InternalImpl;

```
#[storage]
struct Storage {
    #[substorage(v0)]
    achievable: AchievableComponent::Storage,
}

#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    #[flat]
    AchievableEvent: AchievableComponent::Event,
}
  // Constructor

fn dojo_init(self: @ContractState) {
    // [Event] Emit all Achievement creation events
    let world = self.world("<YOUR-NAMESPACE>");
    let task_id = 'TASK_IDENTIFIER';
    let task_target = 100;
    let task = TaskTrait::new(task_id, task_target, "Do
something 100 times");
    let tasks: Span<Task> = array![task].span();

    self.achievable
        .create(
            world,
            id: 'ACHIEVEMENT_IDENTIFIER',
            hidden: false,
            index: 0,
            points: 10,
            start: 0,
```

```
            end: 0,
            group: 'Group',
            title: "Achievement title",
            description: "The achievement description",
            tasks: tasks,
            data: "",
            icon: 'fa-trophy',
        );
    }
 }

}```
```

API References

`AchievableComponent.create`

```
AchievableComponent.create(     self:
@ComponentState<TContractState>,     world: WorldStorage,
id: felt252,     hidden: bool,     index: u8,     points:
u16,     start: u64,     end: u64,     group: felt252,     icon:
felt252,     title: felt252,     description: ByteArray,
tasks: Span<Task>,     data: ByteArray, )
```

See also AchievableComponent

Parameters

- `self` : The component state.
- `world` : The world storage.
- `id` : The achievement identifier, it should be unique.
- `hidden` : Speicify if you want the achievement to be hidden in the controller UI.
- `index` : The achievement index which is the page in which the achievement will be displayed within the group.
- `points` : The achievement points to reward the player.
- `start` : The achievement start timestamp, it should be used for ephemeral achievements, `0` for everlasting achievements.
- `end` : The achievement end timestamp, it should be used for ephemeral achievements, `0` for everlasting achievements.
- `group` : The achievement group, it should be used to group achievements together (see also `index` to define multiple pages).

- `icon` : The achievement icon, it should be a FontAwesome icon name (e.g. `fa-trophy`).
- `title` : The achievement title.
- `description` : The achievement global description.
- `tasks` : The achievement tasks (see also `Task` type).
- `data` : The achievement data, not used yet but could have a future use.

`Task`

```
pub struct Task {     id: felt252,     total: u32,
description: ByteArray, }
```

See also Task

Parameters

- `id` : The task identifier, it should be unique but used in several achievements.
- `total` : The task target, once reached the achievement task is completed.
- `description` : The task description.

Gallery

- DopeWars

Controller Achievements Integration – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/achievements/integration

Configure the controller

The controller needs to be configured with the additional parameters:

- `namespace` : The namespace of the game.
- `slot` : The slot name associated to the torii instance.

```
new ControllerConnector({   url,   rpc,   profileUrl,
namespace: "dopewars",    slot: "ryomainnet",    theme,
colorMode,   policies, });
```

Open the achievements page

Integrate the achievements page in your game client

You can add this following callback to a button to open the achievements page.

``` const { connector } = useAccount();

const handleClick = useCallback(() => { if (!connector?.controller)
{ console.error("Connector not initialized"); return; }
connector.controller.openProfile("achievements"); }, [connector]); ```

## Gallery

- DopeWars

---

## Achievement Progression – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/achievements/progression

## Getting Started

Emit events to track the progress of the player.

The package provides also a way to be used as a cairo package.

```
#[dojo::contract] pub mod Actions {   use achievement::store::
{Store, StoreTrait};   // ...   #[abi(embed_v0)]   impl
ActionsImpl of IActions<ContractState> {   fn play(ref
self: ContractState, do: felt252) {   let world =
self.world(@"<YOUR-NAMESPACE>")   // If the player
meets the task requirement, emit an event to track the
progress   if do === 'something' {   let
store = StoreTrait::new(world);   let player_id =
starknet::get_caller_address();   let task_id =
'TASK_IDENTIFIER';   let count =
1;   let time =
starknet::get_block_timestamp();
store.progress(player_id.into(), task_id, count,
time);   }   }   } }
```
```

API References

`AchievableComponent.progress`

```
AchievableComponent.create(     self:
@ComponentState<TContractState>,     world: WorldStorage,
player_id: felt252,     task_id: felt252,     count: u32, )
```

See also AchievableComponent

Parameters

- `self` : The component state.
- `world` : The world storage.
- `player_id` : The player identifier.
- `task_id` : The task identifier.
- `count` : The progression count to add.

Gallery

- DopeWars

Achievement Setup – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/achievements/setup

Getting Started

Add the Cartridge package `achievement` as a dependency in your Scarb.toml

``` [dependencies] starknet = "  .  .  " dojo = { git = "https://github.com/dojoengine/dojo", tag = "v  .  .  " } achievement = { git = "https://github.com/cartridge-gg/arcade", tag = "v  .  .  " }

[[target.starknet-contract]] build-external-contracts = [ "dojo::world::world*contract::world*", "*achievement::events::index::e*TrophyCreation", "achievement::events::index::e_TrophyProgression", ] ```

## Torii configuration

The progression events require to be managed as historical events by Torii.
```

It means that every single events will remain available in the torii database and accessible in the `event_messages_historical` table.

``` rpc = world_address =

[indexing]

[sql] historical = ["-TrophyProgression"] ```

## Gallery

- DopeWars

---

## Achievement Testing – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/achievements/testing

Do not forget to add the corresponding events to your namespace definition while you setup your tests.

```
 fn namespace_def() -> NamespaceDef {   NamespaceDef
{   namespace: "namespace", resources:
[// ...
TestResource::Event(achievement::events::index::e_TrophyCreation
::TEST_CLASS_HASH),
TestResource::Event(achievement::events::index::e_TrophyProgress
ion::TEST_CLASS_HASH),
TestResource::Contract(Actions::TEST_CLASS_HASH),   ].span
()   }; }
```

---

## Controller Node.js Integration – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/examples/node

This guide demonstrates how to integrate the Cartridge Controller with a Node.js application.

## Installation

npmpnpmyarnbun
```

npm

```
npm install @cartridge/controller starknet
```

Basic Setup

``` import SessionProvider, { ControllerError, } from "@cartridge/controller/session/node"; import { constants } from "starknet"; import path from "path";

export const STRK*CONTRACT*ADDRESS = " x d d e f e bd fcc ddd b f c b b f e dc ";

async function main() { // Path to store session const storagePath = process.env.CARTRIDGE*STORAGE*PATH || path.join(process.cwd(), ".cartridge");

// Create a session provider const provider = new SessionProvider({ rpc: "https://api.cartridge.gg/x/starknet/sepolia", chainId: constants.StarknetChainId.SN_SEPOLIA, policies: { contracts: { [STRK*CONTRACT*ADDRESS]: { methods: [ { name: "approve", entrypoint: "approve", description: "Approve spending of tokens", }, { name: "transfer", entrypoint: "transfer" }, ], }, }, }, basePath: storagePath, });

try { // Connect and create session const account = await provider.connect(); console.log("Session initialized!");

```
if (account) {
  console.log("Account address:", account.address);

  // Example: Transfer STRK
  const amount = "0x0";
  const recipient = account.address; // Replace with actual
recipient address

  const result = await account.execute([
    {
      contractAddress: STRK_CONTRACT_ADDRESS,
      entrypoint: "transfer",
      calldata: [recipient, amount, "0x0"],
    },
  ]);

  console.log("Transaction hash:", result.transaction_hash);
} else {
  console.log("Please complete the session creation in your
```
```

```
browser");
}
```

} catch (error: unknown) { const controllerError = error as ControllerError; if (controllerError.code) { console.error("Session error:", { code: controllerError.code, message: controllerError.message, data: controllerError.data, }); } else { console.error("Session error:", error); } } }

main().catch(console.error); ```

Important Notes

. The `basePath` parameter specifies where session data will be stored. Make sure the directory is writable.
. When running the application for the first time, you'll need to complete the session creation in your browser. The application will provide instructions.
. Session data is persisted between runs, so you don't need to create a new session each time.
. The example includes proper error handling for Controller-specific errors, which include additional context through the `code` and `data` fields.
. Keep your RPC endpoints and contract addresses secure, preferably in environment variables.

Controller React Integration – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/examples/react

This guide demonstrates how to integrate the Cartridge Controller with a React application.

Installation

npmpnpmyarnbun

npm

```
npm install @cartridge/connector @cartridge/controller @starknet-react/core @starknet-react/chains starknet npm install -D tailwindcss vite-plugin-mkcert
```

Basic Setup

. Configure the Starknet Provider

First, set up the Starknet provider with the Cartridge Controller connector:

You can customize the `ControllerConnector` by providing configuration options during instantiation. The `ControllerConnector` accepts an options object that allows you to configure various settings such as policies, RPC URLs, theme, and more.

> ⚠️ ***Important**: The `ControllerConnector` instance must be created outside of any React components. Creating it inside a component will cause the connector to be recreated on every render, which can lead to connection issues.*

``` import { sepolia, mainnet } from "@starknet-react/chains"; import { StarknetConfig, jsonRpcProvider, starkscan, } from "@starknet-react/core"; import ControllerConnector from "@cartridge/connector/controller"; import { SessionPolicies } from "@cartridge/controller";

// Define your contract addresses const ETH*TOKEN*ADDRESS = ' x d d e f e bd fcc ddd b f c b b f e dc '

// Define session policies const policies: SessionPolicies = { contracts: { [ETH*TOKEN*ADDRESS]: { methods: [ { name: "approve", entrypoint: "approve", description: "Approve spending of tokens", }, { name: "transfer", entrypoint: "transfer" }, ], }, }, }

// Initialize the connector const connector = new ControllerConnector({ policies, // With the defaults, you can omit chains if you want to use: // - chains: [ // { rpcUrl: "https://api.cartridge.gg/x/starknet/sepolia" }, // { rpcUrl: "https://api.cartridge.gg/x/starknet/mainnet" }, // ] })

// Configure RPC provider const provider = jsonRpcProvider({ rpc: (chain: Chain) => { switch (chain) { case mainnet: return { nodeUrl: 'https://api.cartridge.gg/x/starknet/mainnet' } case sepolia: default: return { nodeUrl: 'https://api.cartridge.gg/x/starknet/sepolia' } } }, })

export function StarknetProvider({ children }: { children: React.ReactNode }) { return ( <StarknetConfig autoConnect defaultChainId={mainnet.id} chains={[mainnet, sepolia]} provider={provider} connectors={[connector]} explorer={starkscan} > {children} ) } ```
```

. Create a Wallet Connection Component

Use the `useConnect`, `useDisconnect`, and `useAccount` hooks to manage wallet connections:

``` import { useAccount, useConnect, useDisconnect } from '@starknet-react/core' import { useEffect, useState } from 'react' import ControllerConnector from '@cartridge/connector/controller' import { Button } from '@cartridge/ui'

export function ConnectWallet() { const { connect, connectors } = useConnect() const { disconnect } = useDisconnect() const { address } = useAccount() const controller = connectors[ ] as ControllerConnector const [username, setUsername] = useState()

useEffect(() => { if (!address) return controller.username()?.then((n) => setUsername(n)) }, [address, controller])

return (

{address && ( <>

Account: {address}

{username &&

Username: {username}

} </> )} {address ? ( <Button onClick={() => disconnect()}>Disconnect ) : ( <Button onClick={() => connect({ connector: controller })}> Connect )}
) } ```

## . Performing Transactions

Execute transactions using the `account` object from `useAccount` hook:

``` import { useAccount, useExplorer } from '@starknet-react/core' import { useCallback, useState } from 'react'

const ETH_CONTRACT =
' x d d e f e bd fcc ddd b f c
b b f e dc '

export const TransferEth = () => { const [submitted, setSubmitted] = useState(false) const { account } = useAccount() const explorer = useExplorer() const [txnHash, setTxnHash] = useState()

const execute = useCallback( async (amount: string) => { if (!account) return setSubmitted(true) setTxnHash(undefined) try { const result = await account.execute([ { contractAddress: ETH*CONTRACT, entrypoint: 'approve', calldata: [account?.address, amount, '  x  '], }, { contractAddress: ETH*CONTRACT, entrypoint: 'transfer', calldata: [account?.address, amount, '  x  '], }, ]) setTxnHash(result.transaction_hash) } catch (e) { console.error(e) } finally { setSubmitted(false) } }, [account], )

if (!account) return null

return (

## Transfer ETH

<button onClick={() => execute('  x  C  BF  ')} disabled={submitted} > Transfer  .  ETH {txnHash && (

Transaction hash:{' '} {txnHash}

)}
) } ```

### . Add Components to Your App

``` import { StarknetProvider } from './context/StarknetProvider' import { ConnectWallet } from './components/ConnectWallet' import { TransferEth } from './components/TransferEth'

function App() { return ( ) } export default App ```

## Important Notes

Make sure to use HTTPS in development by configuring Vite:

``` import { defineConfig } from 'vite' import react from '@vitejs/plugin-react' import mkcert from 'vite-plugin-mkcert'

export default defineConfig({ plugins: [react(), mkcert()], }) ```

---

## Controller Rust Integration – Cartridge Documentation

**Source:** https://docs.cartridge.gg/controller/examples/rust

## Installation

Add the `account_sdk` crate to your `Cargo.toml`:

```
[dependencies] account_sdk = { git = "https://github.com/
cartridge-gg/controller-rs.git", package = "account_sdk" }
starknet = "0.10" # Make sure to use a compatible version
```

Importing Necessary Modules

```
use account_sdk::{     controller::Controller,
signers::Signer, }; use starknet::{     accounts::Account,
providers::Provider,     signers::SigningKey,
core::types::FieldElement, };
```

Setting Up the Controller

Initialize the controller with necessary parameters:

‴

[tokio::main]

async fn main() { // Create a signer (replace with your own private key) let owner = Signer::Starknet(SigningKey::from*secret*scalar(FieldElement::from*hex*be(" xYourPrivateKey").unwra

```
// Initialize the provider (replace with your RPC URL)
let provider = Provider::try_from("http://
localhost:5050").unwrap();
let chain_id = provider.chain_id().await.unwrap();

// Create a new Controller instance
let username = "testuser".to_string();
let controller = Controller::new(
    "your_app_id".to_string(),
    username.clone(),
    FieldElement::from_hex_be("0xYourClassHash").unwrap(), //
Class hash
    "http://localhost:5050".parse().unwrap(), // RPC URL
    owner.clone(),

FieldElement::from_hex_be("0xYourControllerAddress").unwrap(),
```

```
 // Controller address
     chain_id,
 );

 // Deploy the controller
 controller.deploy().await.unwrap();

 // Interact with the controller
 // For example, execute a transaction
 let call = your_function_call(); // Define your function call
 controller.execute(vec![call], None).await.unwrap();

}```
```

Performing Transactions

Define function calls and execute them:

```
 fn your_function_call() -> starknet::core::types::FunctionCall
{     starknet::core::types::FunctionCall
{         contract_address:
FieldElement::from_hex_be("0xYourContractAddress").unwrap(),
    entry_point_selector:
starknet::core::utils::get_selector_from_name("yourEntryPoint").
unwrap(),         calldata: vec![FieldElement::from(123)], //
Replace with your calldata     } }
```

Execute the function call using the controller:

```
 controller.execute(vec![your_function_call()],
None).await.unwrap();
```

Controller Svelte Integration – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/examples/svelte

Installation

npmpnpmyarnbun

npm

```
npm install @cartridge/controller starknet
```

Setting Up the Controller

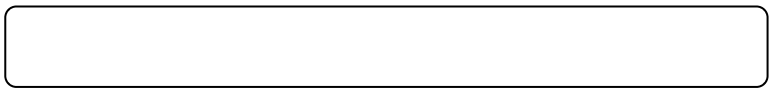
Import the `Controller` and create an instance:

``` // src/routes/+page.svelte import { onMount } from "svelte"; import Controller from "@cartridge/controller"; import { account, username } from "../stores/account"; import { ETH_CONTRACT } from "../constants";

let controller = new Controller({ policies: { contracts: { [ETH_CONTRACT]: { methods: [ { name: "approve", entrypoint: "approve", description: "Approve spending of tokens", }, { name: "transfer", entrypoint: "transfer", description: "Transfer tokens", }, ], }, }, }, }); ```

## Connecting a Wallet

Use the `connect` method to establish a connection:

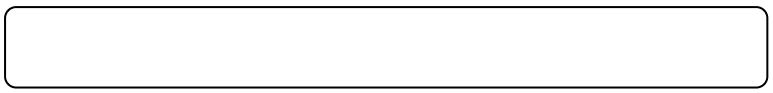


``` Connect ```

## Disconnecting a Wallet

Implement a disconnect function:

```



Disconnect ```

Displaying User Information

Create a `UserInfo` component to show account details:

```

# User Information

{ if accountAddress}

Account Address: {accountAddress}

{:else}

No account connected

{/if} {  if username}

Username: {username}

{/if}

```

Performing Transactions

Create a `TransferEth` component for executing transactions:

```

## Transfer Eth

<button on:click={() => execute('  x  ', false)}>Transfer  ETH to self <button on:click={() => execute('  x  C  BF  ', false)}>Transfer  . ETH to self <button on:click={() => execute('  x  ', true)}>Manual: Transfer  ETH to self ```

### Full Example

Here's how your main `+page.svelte` might look:

```

SvelteKit + Controller Example

{ if loading}

Loading

{:else if $account} Disconnect {:else}

Connect {/if}

{ if $account && !loading} {/if} ```
```

This example demonstrates how to set up the Controller, connect/disconnect a wallet, display user information, and perform transactions in a Svelte application using the Cartridge Controller.

Controller Telegram Integration – Cartridge Documentation

Source: https://docs.cartridge.gg/controller/examples/telegram

Controller Integration Flow

. Generate local Stark key pair and store private key in Telegram cloud storage
. Open session controller page with user's public key
. Controller registers session public key and returns account info
. Create controller session account on client
. Store account info in Telegram cloud storage

Setting up the Session Provider

. Define your configuration:

``` // config.ts export const RPC_URL = "https://api.cartridge.gg/x/starknet/mainnet";

// Define your session policies export const SESSION_POLICIES = { contracts: { " x fc f e c a b b b d bd a e a e b bf eec f": { name: "Beast Game Contract", description: "Contract for beast game interactions", methods: [ { name: "attack", entrypoint: "attack", description: "Attack the beast", }, { name: "claim", entrypoint: "claim", description: "Claim your tokens", }, ], }, }, };

export const REDIRECT_URI = "https://t.me/hitthingbot/hitthing"; ```

### . Create the SessionProvider:

``` import { PropsWithChildren, createContext, useContext, useEffect, useState } from "react"; import { constants } from "starknet"; import SessionConnector from "@cartridge/connector/session"; import { StarknetConfig, jsonRpcProvider } from "@starknet-react/core"; import { useLaunchParams, cloudStorage } from "@telegram-apps/sdk-react"; import { RPC*URL, SESSION*POLICIES, REDIRECT_URI } from "./config";

const connector = new SessionConnector({ policies: SESSION*POLICIES, rpc: RPC*URL, chainId: constants.StarknetChainId.SN*MAINNET, redirectUrl: REDIRECT*URI, });

const provider = jsonRpcProvider({ rpc: () => ({ nodeUrl: RPC_URL }), });

export function SessionProvider({ children }: PropsWithChildren) { return ( {children} ); } ```

### . Use the SessionProvider in your app:

``` // App.tsx import { SessionProvider } from "./SessionProvider";

function App() { return ( ); } ```

. Use the session in your components:

``` function GameComponent() { const { isConnected, connect, disconnect } = useSession();

const handleConnect = async () => { try { await connect(); } catch (error) { console.error("Failed to connect:", error); } };

return (

{!isConnected ? ( Connect Wallet ) : ( Disconnect )}

```
    {isConnected && (
      <div>
        {/* Your game content */}
      </div>
)}
  </div>

); } ```
```

. Error Handling

``` function GameComponent() { const { connect } = useSession(); const [error, setError] = useState();
```

const handleConnect = async () => { try { await connect(); } catch (err) { setError(err instanceof Error ? err.message : "Failed to connect"); } };

return (

{error &&
{error}
}

Connect

); } ```

See the full example here.

Next.js Configuration for WebAssembly

If you're using Next.js, you'll need to configure it to properly handle WebAssembly modules used by the SessionController and SessionConnector. Create or update your `next.config.js`:

``` / @type {import('next').NextConfig} / const nextConfig = { reactStrictMode: true, webpack: (config, { isServer, dev }) => { // Enable WebAssembly config.experiments = { asyncWebAssembly: true, topLevelAwait: true, };

```
// Fix for WebAssembly in production builds
if (!dev && isServer) {
  config.output.webassemblyModuleFilename = "chunks/
[id].wasm";
  config.plugins.push(new WasmChunksFixPlugin());
}

return config;
```

}, };

// Plugin to fix WASM chunk loading in production class WasmChunksFixPlugin { apply(compiler) { compiler.hooks.thisCompilation.tap("WasmChunksFixPlugin", (compilation) => { compilation.hooks.processAssets.tap( { name: "WasmChunksFixPlugin" }, (assets) => Object.entries(assets).forEach(([pathname, source]) => { if (!pathname.match(/.wasm$/)) return; compilation.deleteAsset(pathname);

```
const name = pathname.split("/")[1];
const info = compilation.assetsInfo.get(pathname);
```
```

```
          compilation.emitAsset(name, source, info);
        }),
    );
  });

}}

module.exports = nextConfig; ```
```

Slot Getting Started – Cartridge Documentation

Source: https://docs.cartridge.gg/slot/getting-started

Slot is the execution layer of Dojo, supporting rapid provisioning of low latency, dedicated, provable execution contexts, bringing horizontal scalability to the blockchain. It manages the sequencing, proving, and efficient settlement of its execution.

Installation

Install slotup to manage slot installations and follow the outputted directions.

```
curl -L https://slot.cartridge.sh | bash
```

Usage

Authenticate with Cartridge

```
slot auth login
```

Programmatic usage

First, authenticate as mentioned above. Then, run:

```
slot auth token
```

Follow instructions and save the output to set the SLOT_AUTH env var. You can set this environment variable in CI, scripts, or deployment platforms to run slot without having to login.

Create service deployments

```
slot deployments create <Project Name> katana slot deployments create <Project Name> torii --world 0x3fa481f41522b90b3684ecfab7650c259a76387fab9c380b7a959e3d4ac69f
```

Update a service

```
slot deployments update <Project Name> torii --version v0.3.5
```

Delete a service

```
slot deployments delete <Project Name> torii
```

Read service logs

```
slot deployments logs <Project Name> <katana | torii>
```

List all deployments

```
slot deployments list
```

View deployments configuration

```
slot deployments describe <Project Name> <katana | torii>
```

View predeployed accounts

```
slot deployments accounts <Project Name> katana
```

Manage collaborators with teams

The name of the team is the same as the project name used to create a service. A team is automatically created when you create a new project.

```
slot teams <Team Name> list slot teams <Team Name> add <Account Name> slot teams <Team Name> remove <Account Name>
```

Fund teams

Teams need credits to run services and paymasters. You can fund teams using CLI commands or the web interface:

CLI:

```
slot auth fund slot auth transfer <Team Name> --credits <amount>
```

Web Interface: Navigate to `https://x.cartridge.gg/slot/fund` to fund teams through a user-friendly interface with credit card or crypto payments.

Paymaster Management – Cartridge Documentation

Source: https://docs.cartridge.gg/slot/paymaster

Paymasters in Slot allow you to sponsor transaction fees for your applications, enabling gasless experiences for your users. The Cartridge Paymaster is a powerful feature that enables gasless transactions for your users, creating a more seamless Web experience. When enabled, the paymaster automatically covers transaction fees on behalf of your users, eliminating the need for them to hold ETH / STRK for gas fees. You can manage budgets, policies, and monitor usage through the Slot CLI.

Availability

The paymaster service is available across all networks with different activation requirements:

- **Testnet Networks**
  - Automatically enabled, no additional setup required
- **Mainnet**
  - Available and fully self-served
  - Manage everything through the Slot CLI
  - Define your own usage scopes and spending limits

Integration

One of the key benefits of the Cartridge Paymaster is that it requires zero additional integration work. When the paymaster is enabled for your application, it will automatically activate for all eligible transactions. No code changes or configuration are needed.

Prerequisites

Make sure you are authenticated with Slot:

```
slot auth login
```

Team Setup

Before creating a paymaster, you need a team with sufficient credits.

Create a Team (if it doesn't exist)

```
slot teams <team-name> create --email <your-email@example.com>
```

Fund Your Account and Transfer to Team

Paymasters automatically deduct from your team's account balance when created. If you don't have sufficient credits:

```
# Buy credits for your account (opens browser) slot auth fund
```

. Select the team you want to fund from the list
. Choose your payment method (credit card or crypto)
. Complete the purchase

Creating a Paymaster

Create a new paymaster with an initial budget:

```
slot paymaster <paymaster-name> create --team <team-name> --
budget <amount> --unit CREDIT
```

Example

```
slot paymaster my-game-pm create --team my-team --budget 1000 --
unit CREDIT
```

Output:

``` ✅ Paymaster Created Successfully

---

---

🏢 Details: • Name: my-game-pm • Team: my-team

💰 Initial Budget: • Amount: CREDIT ($ . USD) ```
```

Managing Budget

Increase Budget

Add funds to your paymaster:

```
 slot paymaster <paymaster-name> budget increase --amount
<amount> --unit CREDIT
```

Output:

``` ✅ Budget Increased Successfully
---

---

🏦 Paymaster: my-game-pm

📈 Operation: • Action: Increased • Amount: CREDIT

💰 New Budget: • Amount: CREDIT ($ . USD) ```
```

Decrease Budget

Remove funds from your paymaster:

```
 slot paymaster <paymaster-name> budget decrease --amount
<amount> --unit CREDIT
```

Output:

``` ✅ Budget Decreased Successfully
---

---

🏦 Paymaster: my-game-pm

📉 Operation: • Action: Decreased • Amount: CREDIT

💰 New Budget: • Amount: CREDIT ($ . USD) ```
```

Policy Management

Policies define which contracts and entry points your paymaster will sponsor.

Add Policies from Preset (Recommended)

The preferred way to add policies is using verified contract presets for your games:

```
slot paymaster <paymaster-name> policy add-from-preset --name <preset-name>
```

Example:

```
slot paymaster my-game-pm policy add-from-preset --name dope-wars
```

Add a Single Policy

For individual contract policies or custom contracts not yet in presets:

```
slot paymaster <paymaster-name> policy add --contract <contract-address> --entrypoint <entry-point>
```

Add Policies from JSON File

For bulk adding multiple custom policies:

```
slot paymaster <paymaster-name> policy add-from-json --file <path-to-json>
```

JSON Format:

```
[   {     "contractAddress": "0x1234...abcd",     "entryPoint": "move_player"   },   {     "contractAddress": "0x5678...efgh",     "entryPoint": "attack"   } ]
```

Remove a Policy

```
slot paymaster <paymaster-name> policy remove --contract <contract-address> --entrypoint <entry-point>
```

Output:

```
Successfully removed policy: PolicyArgs { contract: "0x1234...abcd", entrypoint: "move_player" }
```

Remove All Policies

```
slot paymaster <paymaster-name> policy remove-all
```

List Policies

```
slot paymaster <paymaster-name> policy list
```

Paymaster Information

Get comprehensive information about your paymaster:

```
slot paymaster <paymaster-name> info
```

Output:

``` 🔍 Paymaster Info for 'my-game-pm'

---

———————— 🏢 Details: • Team: my-team • Active: ✅ Yes

💰 Budget: • Total: CREDIT ($ . USD) • Spent: . CREDIT ($ . USD) • Usage: [████░░░░░░░░░░░░░░░░] . %

📋 Policies: • Count: ```

## Updating Paymaster Configuration

Update your paymaster's basic configuration settings:

```
slot paymaster <current-name> update [OPTIONS]
```

### Update Paymaster Name

```
slot paymaster my-game-pm update --name new-game-pm
```

Change Team Association

Transfer the paymaster to a different team:

```
slot paymaster my-game-pm update --team new-team
```

Enable/Disable Paymaster

Toggle the active state of your paymaster:

```

# Disable paymaster (stops sponsoring transactions)

slot paymaster my-game-pm update --active false

# Re-enable paymaster

slot paymaster my-game-pm update --active true ```

## Statistics and Monitoring

View usage statistics for your paymaster:

```
slot paymaster <paymaster-name> stats --last <time-period>
```

Example:

```
slot paymaster my-game-pm stats --last 24hr
```

Time Period Options:

- `1hr` , `2hr` , `24hr`
- `1day` , `2day` , `7day`
- `1week`

Output:

``` 📊 Paymaster Stats for 'my-game-pm' (Last hr)

---

——————— 📈 Transactions: • Total: , • Successful: , • Reverted: • Success Rate: . %

💰 Fees (USD): • Total ( hr): $ . • Average: $ . • Minimum: $ . • Maximum: $ .

👥 Users: • Unique Users: ```

## Transaction History

View detailed transaction history for your paymaster with filtering and sorting options:

```
slot paymaster <paymaster-name> transactions [OPTIONS]
```
```

Basic Usage

View recent transactions:

```
slot paymaster my-game-pm transactions
```

Output:

```
📊 Paymaster Transactions for 'my-game-pm' (Last 24hr)
──────────────────────────────────────────────────────────────────────────────
──────────── Transaction
Hash
Executed     Status       USD Fee
──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────
0x50c2dd556593564fe2b814d61b3b1592682de83702552a993d24f9e897710e
7  11s ago      SUCCESS      $0.0026
0x41b0f547741bd1fdc29dd4c82a80da2a452314e710ae7cbe0e05cb4cb1e6c0
e  22s ago      SUCCESS      $0.0025
0x4b74ee2ab7764cb3d11f3319b64c2698b868727fdf99728bdf74aa023b5e77
d  32s ago      REVERTED     $0.0028
0x2af69b9798355e91119c6a9adb1363b2f533f0557601e4687dcfe9725e8fea
a  42s ago      SUCCESS      $0.0025
0x25dfc115dabda89a2027366790ee5cfcfefb861fe1b584c6fb15dc1588e081
6  47s ago      REVERTED     $0.0032
```

Filtering Options

Filter by Status:

```

# Show only successful transactions

slot paymaster my-game-pm transactions --filter SUCCESS

# Show only reverted transactions

slot paymaster my-game-pm transactions --filter REVERTED
```

Show all transactions (default)

slot paymaster my-game-pm transactions --filter ALL ```

Time Period:

```
# Last hour slot paymaster my-game-pm transactions --last 1hr
```

Sorting:

```

# Sort by fees (ascending)

slot paymaster my-game-pm transactions --order-by FEES_ASC

# Sort by fees (descending)

slot paymaster my-game-pm transactions --order-by FEES_DESC

# Sort by execution time (most recent first - default)

slot paymaster my-game-pm transactions --order-by EXECUTED*AT*DESC

# Sort by execution time (oldest first)

slot paymaster my-game-pm transactions --order-by EXECUTED*AT*ASC ```

**Limit Results:**

```
Show up to 50 transactions (max 1000) slot paymaster my-game-pm transactions --limit 50
```

## Dune Analytics Queries

Generate Dune Analytics queries to analyze your paymaster's transaction data:
```

```
slot paymaster <paymaster-name> dune [OPTIONS]
```

Example Dashboard

See a live example of paymaster analytics at Blob Arena Stats on Dune Analytics.

Query Types

Fast Query (Default)

```
slot paymaster my-game-pm dune
```

- Quick execution suitable for long time periods
- Matches direct execute_from_outside_v calls and simple vRNG patterns
- Does not catch complex nested vRNG calls
- Best for initial analysis and long-term trends

Exact Query

```
slot paymaster my-game-pm dune --exact
```

- Exhaustive search of all transaction patterns
- Uses execute_from_outside_v selector as anchor
- Catches all patterns including nested vRNG calls
- May timeout on long time periods
- Best for exact metrics

Time Period Options

By default, queries use the paymaster's creation time. You can specify a custom time period:

‵‵‵

Last hours

slot paymaster my-game-pm dune --last hr

Last week

slot paymaster my-game-pm dune --last week

Combine with exact query

slot paymaster my-game-pm dune --exact --last hr ```

Time Period Options:

- `1hr , 2hr , 24hr`
- `1day , 2day , 7day`
- `1week`

Query Output

The command generates a SQL query that you can copy and run in Dune Analytics. The query includes:

- Daily transaction counts
- Unique user counts
- Fee amounts in USD
- Overall totals

Quick Debugging Use Cases

The transaction history is useful for identifying issues:

View expensive transactions that might indicate inefficient contract calls:

```
slot paymaster my-game-pm transactions --order-by FEES_DESC --limit 10
```

Investigate failed transactions to debug contract issues:

```
slot paymaster my-game-pm transactions --filter REVERTED --last 24hr
```

Best Practices

Budget Management

- Start with a conservative budget and increase as needed
- Monitor spending through the stats command
- Keep sufficient team balance for paymaster operations

Policy Management

- Be specific with your policies to avoid sponsoring unintended transactions
- **Use presets whenever possible** for verified game contracts in the Dojo ecosystem
- Contribute your game contracts to the preset repository for community verification
- Regularly review and update policies as your application evolves
- Test policies with small budgets before scaling up

Security

- Only add policies for contracts you trust
- Keep your team membership limited to necessary collaborators

Common Workflows

Setting up a new game paymaster

```
# Create team if it doesn't exist
slot teams my-team create --email developer@mygame.com

# Fund your account and transfer to team
slot auth fund
slot auth transfer my-team --credit

# Create paymaster
slot paymaster my-game-pm create --team my-team --budget  --unit CREDIT
```

Add game contract policies

slot paymaster my-game-pm policy add --contract x ...abc --entrypoint move*player slot paymaster my-game-pm policy add --contract x ...abc --entrypoint attack*enemy slot paymaster my-game-pm policy add --contract x ...abc --entrypoint use_item

Check initial setup

slot paymaster my-game-pm info ```

Monitoring and maintenance

```

# Check daily stats

slot paymaster my-game-pm stats --last hr

# Check current status

slot paymaster my-game-pm info

# Add more budget if needed (ensure team has credits)

slot paymaster my-game-pm budget increase --amount --unit CREDIT ```

### Insufficient Credits Error

If you encounter insufficient credits when creating or funding a paymaster:

```

Check team balance first

slot teams my-team info

Fund your account if needed

slot auth fund

Transfer more credits to team

slot auth transfer my-team --credit

Retry your paymaster operation

slot paymaster my-game-pm create --team my-team --budget --unit CREDIT ```

Scale your deployments – Cartridge Documentation

Source: https://docs.cartridge.gg/slot/scale

Slot instances are launched with the `basic` instance by default. This instance type is only suitable for testing and development purposes and comes with limited CPU & memory. deployments of the basic tier are free.

To prepare your deployments for production, you can set up billing and upgrade to a paid instance tier.

Instances

| Tier | Description | Storage | Old cost | Cost from July st | | --- | --- | --- | --- | --- | | Basic | First are free. for dev & tests | GB | $ /month | $ /month | | Common | (removed) | auto | $ /month | – | | Pro | vCPU and GB RAM | auto | - | $ /month | | Epic | vCPU and GB RAM | auto | $ /month | $ /month | | Legendary | vCPU and GB RAM | auto | $ /month | $ /month | | Insane | (removed) | auto | $ /month | - |

Note: basic instances are scaled down automatically after a few hours of no activity. To revive deployments, simply send a single request to the instance URL, and it'll be revived on the spot. If unused without any activity for more than days, it will get deleted.

Premium tiers

Pro and higher tiers are never scaled down or deleted as long as there are enough credits on the related team. They also come with auto storage scaling, which means your deployment can never run out of disk space.

Storage is billed at $. /GB/month.

Replicas

For torii, with premium tiers, you can choose to deploy your instances with multiple replicas using the `--replicas <n>` flag.

```
slot d create --tier epic my-project torii --replicas 3
```

Replicas are billed as how many replicas you have. For example, if you have replicas, you will be billed times the monthly cost of the tier you are using.

Regions

For torii, with premium tiers, you can choose to deploy your instances in multiple regions.

Region
`us-east`
`europe-west`
`asia-southeast`

To deploy a slot service in multiple regions, you can use the `--regions` flag.

Katana is only supported in `us-east` at this time.

```
slot d create --tier pro my-project torii --regions us-east,asia-southeast,europe-west
```

Multi-region deployments are billed based on the number of regions you choose and the tier you are using by multiplying the monthly cost of the tier by the number of regions times replicas. For example, if you have two replicas in three regions, you will be billed six times the monthly cost of the tier you are using.

Set up billing

To set up slot billing, you need to buy credits and transfer them to a slot team.

If you have existing deployments, a team of the same name as your account will be created for you, and you can transfer credits to it.

``` slot teams my-team create --email my-email@example.com email is required for billing alerts slot teams my-team update --email my-email@example.com if you want to update an existing team's email

slot auth fund buy credits for your account, this opens the browser

slot auth transfer my-team --usd transfer $ from your controller to my-team

# or

slot auth transfer my-team --credits transfer credits ($ ) from your controller to my-team ```

Once you create paid slot instances, funds are deducted on a daily basic with a minimum charge of one day. For example, if you create a deployment and delete it after one hour, you will be charged / th of the monthly cost.

## Create an instance with a paid tier

Make sure to use the team flag of the team you previously transferred credits to.

```
slot d create --tier epic --team my-team my-instance torii
```

## Update an existing instance to a paid tier

Note that you can only upgrade tiers; downgrading to a lower tier is not possible.

Make sure that you created this instance with the team flag of the team you previously transferred credits to, or make sure the team it belongs to has credits available.

```
slot d update --tier epic my-instance torii
```
```

vRNG Overview – Cartridge Documentation

Source: https://docs.cartridge.gg/slot/vrng

This Cartridge Verifiable Random Number Generator (vRNG) is designed to provide cheap, atomic verifiable randomness for fully onchain games.

Key Features

. **Atomic Execution**: The vRNG request and response are processed within the same transaction, ensuring synchronous and immediate randomness for games.
. **Efficient Onchain Verification**: Utilizes the Stark curve and Poseidon hash for optimized verification on Starknet.
. **Fully Onchain**: The entire vRNG process occurs onchain, maintaining transparency and verifiability.
. **Improved Player Experience**: The synchronous nature of the vRNG allows for instant resolution of random events in games, enhancing gameplay fluidity.

How It Works

. A game calls `request_random(caller, source)` as the first call in their multicall.
. A game contract calls `consume_random(source)` on the vRNG contract.
. The vRNG server generates a random value using the vRNG algorithm for the provided entropy source.
. The Cartridge Paymaster wraps the players multicall with a `submit_random` and `assert_consumed` call.
. The `submit_random` call submit a vRNG Proof for the request, the vRNG Proof is verified onchain, ensuring the integrity of the random value which is immediately available and must be used within the same transaction.
. The `assert_consumed` call ensures that `consume_random(source)` has been called, it also reset the storage used to store the random value during the transaction to .

Benefits for Game Developers

- **Simplicity**: Easy integration with existing Starknet smart contracts and Dojo.
- **Performance**: Synchronous randomness generation without waiting for multiple transactions.
- **Cost-effectiveness**: Potential cost savings through Paymaster integration.
- **Security**: Cryptographically secure randomness that's fully verifiable onchain.

Deployments

| Network | Contract Address | Class Hash | | --- | --- | --- | | Mainnet | x fea da d aee bdeba b f bcbf d c aa e ac ced f | x be edf dd aa c b a bcee c a ed db a c | | Sepolia | x fea da d aee bdeba b f bcbf d c aa e ac ced f | x be edf dd aa c b a bcee c a ed db a c |

For detailed implementation and usage, refer to the GitHub repository.

Using the vRNG Provider

To integrate the Verifiable Random Function (vRNG) into your Starknet contract, follow these steps:

. Define the vRNG Provider interface:

```
# [starknet::interface]

trait IVrfProvider { fn request*random(self: @TContractState, caller: ContractAddress, source: Source); fn consume*random(ref self: TContractState, source: Source) -> felt ; }

# [derive(Drop, Copy, Clone, Serde)]

pub enum Source { Nonce: ContractAddress, Salt: felt , } ```

. Define the vRNG Provider address in your contract:

```
const VRF_PROVIDER_ADDRESS: starknet::ContractAddress = starknet::contract_address_const::<0x123>();
```

. Create a dispatcher for the vRNG Provider:
```

```
let vrf_provider = IVrfProviderDispatcher { contract_address:
VRF_PROVIDER_ADDRESS };
```

. To consume random values, use the following pattern in your contract functions:

``` fn roll_dice(ref self: ContractState) { // Your game logic here...

```
// Consume random value
let player_id = get_caller_address();
let random_value =
vrf_provider.consume_random(Source::Nonce(player_id));

// Use the random value in your game logic
// ...
```

}```

. You can use either `Source::Nonce(ContractAddress)` or `Source::Salt(felt252)` as the source for randomness:

- `Source::Nonce(ContractAddress)` : Uses the provided contract address internal nonce for randomness.
  Each request will generate a different seed ensuring unique random values.
- `Source::Salt(felt252)` : Uses a provided salt value for randomness.
  Two requests with same salts will result in same random value.

## Executing vRNG transactions

In order to execute a transaction that includes a `consume_random` call, you need to include a `request_random` transaction as the first transaction in the multicall. The `request_random` call allows our server to efficiently parse transactions that include a `consume_random` call internally.

```
const call = await account.execute([// Prefix the multicall
with the request_random call   {   contractAddress:
VRF_PROVIDER_ADDRESS,   entrypoint: 'request_random',
calldata: CallData.compile({   caller:
GAME_CONTRACT,   // Using Source::Nonce(address)
source: {type: 0, address: account.address},   // Using
Source::Salt(felt252)   // source: {type: 1, salt:
0x123}   }),   },   {   contractAddress: GAME_CONTRACT,
entrypoint: 'roll_dice',   },]);
```
```

Ensure that you call `consume_random` with the same `Source` as used in `request_random`.

Important: Adding vRNG to Policies

When using the Cartridge Controller with vRNG, make sure to add the vRNG contract address and the `request_random` method to your policies. This allows the controller to pre-approve vRNG-related transactions, ensuring a seamless experience for your users.

Add the following policy to your existing policies:

```
const policies: Policy[] = [   // ... your existing policies ...   {     target: VRF_PROVIDER_ADDRESS,     method: "request_random",     description: "Allows requesting random numbers from the VRF provider",   }, ];
```

This ensures that vRNG-related transactions can be executed without requiring additional user approval each time.

By following these steps, you can integrate the vRNG Provider into your Starknet contract and generate verifiable random numbers for your onchain game or application.

Security Assumptions

During the Phase deployment, the construction assumes the Provider has not revealed the private key and does not collude with players.

In the future, we plan to move the Provider to a Trusted Execution Environment (TEE) in order to provide a more robust security model without compromising on performance.
