

CONTRIBUTING.md:

Contributing to Flow Docs

Reading this document carefully will help you understand how to contribute your own content to the Flow Docs, and avoid problems along the way.

In most cases you should be able to iterate and deploy changes to your documentation with no to little involvement the engineering team that is maintaining the Flow Docs.

Getting Prepared

Before diving into the contribution process, make sure you're set up for a smooth experience.

Account Setup

To get started, you'll need a GitHub account. If you don't have one yet, no worries! Sign up here. Once you have your account ready, you're all set to proceed.

Contributing Your Content

The contribution process is designed to be straightforward and collaborative. Here's the basic workflow:

1. Create a Pull Request: Head over to the onflow/docs repository and create a Pull Request into the main branch. This is where you'll submit your proposed changes.

1. Validation and Preview: Wait for the automated check runs to complete. Address any validation errors that pop up. You can also preview the changes using the provided link to ensure everything looks as intended.

1. Merge and Publish: Once your PR is ready and error-free, go ahead and merge it. Your updated documentation will promptly become accessible on <https://developers.flow.com>!

Note: Previews are built for each Git branch by Vercel workflow.

For additional assistance or integration support, feel free to request help in the #devex-builders channel on the Flow Discord server.

Updating Existing Content

Updating existing content is a seamless process:

1. Look for the "Edit this page" link at the bottom of each page, just before the footer.

```
<Callout type="note" title="Here is what it looks like.">
!edit page
!edit page
</Callout>
```

1. Clicking the link will take you directly to the source code of the page you're viewing.

Adding New Content

Expanding the Flow Docs with new content is straightforward:

1. **Markdown Mastery:** To display content on the Flow Docs, use Markdown format. Markdown syntax resources are available for learning and reference:

- <https://www.markdowntutorial.com/>
 - <https://www.markdownguide.org/>

1. From Google Docs to Markdown: While Markdown is perfect for single authors, Google Docs is more suited for collaborative writing. If you're using Google Docs, consider using this browser plugin to generate Markdown files from your Google Docs documents.

<https://workspace.google.com/marketplace/app/docstomarkdown/700168918607>

Adding category

Adding a new section/category is as easy as creating a new folder and populating it with Markdown files.

For instance, to add a new toolchain, the structure might look like this:

```
./docs/tools/toolchains
|   └── tools...
|       └── toolchains
|           └── new-toolchain          // new section/category
|               ├── index.md
|               ├── category.yml
|               ├── page1.md
|               ├── page2.mdx
|               └── dir1
|                   └── nested-page1.md
|                       └── ...
|                           ...
|                               ...
```

category

category file is used to customize the category/section menu items content related to it.

Possible format:

- category.json
- category.yml
- category.yaml

Example using .yml:

```
yaml category.yml
  label: Deploying Contracts
  position: 2
  customProps:
    icon: ☒
    description: How to deploy smart contracts to mainnet and testnet
```

For more information refer to the Official Docusaurus documentation

index file

The index page should contain links to the content inside of a category:

- list of available pages and categories
- short descriptions

Examples:

```
markdown index.md
Clients
```

Go SDK

Flow Go SDK provides a set of packages for Go developers to build applications that interact with the Flow network.

JavaScript (FCL)

Flow Client Library (FCL) is a package used to interact with user wallets and the Flow blockchain.

<!-- ... Other sections ... -->

```
markdown index.md
---
sidebarposition: 1
title: Tools
description: Essential tools for the Flow blockchain ecosystem
---
```

```
import DocCardList from '@theme/DocCardList';
import { isSamePath } from '@docusaurus/theme-common/internal';
import { useDocsSidebar } from '@docusaurus/plugin-content-docs/client';
```

```
import { useLocation } from '@docusaurus/router';

<DocCardList items={useDocsSidebar().items.filter(item =>
!isSamePath(item.href, useLocation().pathname))}/>

:::warning
The index page should only include information that is available
elsewhere within the category's other pages.

:::

Using DocCardList

Using DocCardList improves the layout of the index page:

:::note Example

!edit page

!edit page

:::

On the most pages you can use just <DocCardList /> component imported
from '@theme/DocCardList'

markdown
import DocCardList from '@theme/DocCardList';

Deployments

This section contains guides to deploying and testing smart contracts.

All Sections
<DocCardList />

On the top level index category pages you have to use useDocsSidebar
react hook to avoid rendering errors

markdown
import DocCardList from '@theme/DocCardList';
import { isSamePath } from '@docusaurus/theme-common/internal';
import { useDocsSidebar } from '@docusaurus/plugin-content-docs/client';
import { useLocation } from '@docusaurus/router';

<DocCardList items={useDocsSidebar().items.filter(item =>
!isSamePath(item.href, useLocation().pathname))}/>
```

:::warning

If you use `<DocCardList/>` on the top level category index page (e.g. `./docs/tools/index.mdx`), Docusaurus will throw an error:

```
> useCurrentSidebarCategory() should only be used on category index pages.
```

:::

SEO

Basic SEO metadata can be included in markdown frontmatter. The Flow Docs supports title and description metadata. If none is provided, a default will be applied.

Example frontmatter:

```
markdown
---
title: "Hello World"
description: "A Great Document"
---
```

It is not required to supply this SEO metadata. Flow's in-house SEO experts will provide custom search-engine appropriate metadata for each page. Your custom metadata will be used in lieu of metadata supplied by our SEO experts.

Page Content

Images and Other Media

Include images and media using relative URLs within the docs folder. If your media is viewable in GitHub, then it should display on the Flow Docs.

For external media or URLs, use fully qualified URLs, eg:

```
!image
```

The same rule applies to all other external media.

Links

Linking to documents within the docs folder of your repo can be done in the standard way that is acceptable to GitHub, eg.

```
link
```

Writing links in your documents is easy if you follow this rule-of-thumb: If it works in GitHub it should work on the Flow Docs, with one notable exception.

Links to content outside the docs folder in your repository, or elsewhere on the web must be in the form of a fully-qualified URL, eg:

link

:::tip

Use relative links directly to .md/.mdx files

:::

Callouts

Many pages have callouts/admonitions. It is a slightly customized version of standard admonitions

Available types:

- note (also secondary)
- tip (also success)
- info (also important)
- caution
- warning
- danger

Example:

markdown

:::tip

Use relative links directly to .md/.mdx files

:::

Code Reference

To include code from a file using a direct URL use !from operand inside a code block.

Example:

markdown

cadence DepositFees.cdc

```
!from https://github.com/onflow/flow-core-
contracts/blob/master/transactions/FlowServiceAccount/depositfees.cdc#L23
-L26
```

This method keeps your documentation synchronized with your codebase by pulling the latest code directly into your docs.

Content Validation

Content is validated each time a PR is submitted to the docs repository.

Validation status is available in the check run output for your PR on GitHub.

Currently, the Flow Docs validates content using the following conditions:

- Do pages render without errors? This check accounts for errors in your markdown syntax.
- Are all links URLs valid? Broken links make for bad user experience.

Here is an example of check run output with successful validation.

```
!successful-checks
```

Dead Links Checks

The Flow Docs automatically scans relative links in all your documents when you submit changes to your docs. This is done to ensure that cross-links are valid, within your set of documentation.

Content Previews

A special preview link is provided for content PRs as part of the GitHub PR check run.

Previews are generated against the PR branch of the Flow Docs, to ensure your content can be integrated with the latest updates to the Flow Docs itself.

Here is an example of preview output for changed documents in a PR

```
!preview-link
```

When you merge updates to documents in your PR, previews will deployed with a few minutes delay.

```
# README.md:
```

Website

This website is built using Docusaurus 2, a modern static website generator.

```
it pulls doc sections from repositories defined in  
https://github.com/onflow/developer-portal/tree/main/app/data/doc-  
collections
```

Installation

```
$ yarn
```

Build

```
$ yarn build
```

This command generates static content into the build directory and can be served using any static contents hosting service.

Local Development

```
$ yarn start
```

This command starts a local development server and opens up a browser window. Most changes are reflected live without having to restart the server.

Deployment

Using SSH:

```
$ USESSH=true yarn deploy
```

Not using SSH:

```
$ GITUSER=<Your GitHub username> yarn deploy
```

If you are using GitHub pages for hosting, this command is a convenient way to build the website and push to the gh-pages branch.

Search Indexing

```
- copy .env Flow Docusaurus Prod/.env Flow Docusaurus Prod to your local  
.env file from 1Password->Flow Team vault
```

```
# explore-more.md:
```

```

---
title: Explore More
sidebarlabel: Explore More
description: Explore more of the Flow blockchain
sidebarposition: 999
---

import DocCardList from '@theme/DocCardList';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'
import { faWindowMaximize, faCoins, faGem, faBook, faLandmark } from
'@fortawesome/free-solid-svg-icons'

```

Below are some additional tutorials to help you get started with Flow:

```

<DocCardList items={[
  {
    type: 'link',
    label: 'Flow App Quickstart',
    href: '/build/getting-started/fcl-quickstart',
    description: 'Simple walkthrough building a web3 app using the Flow
Client Library (FCL)',
    customProps: {
      icon: <FontAwesomeIcon icon={faWindowMaximize} className="h-16" />,
      author: {
        name: 'Flow Blockchain',
        profileImage:
          'https://avatars.githubusercontent.com/u/62387156?s=200&v=4',
      },
    },
  },
  {
    type: 'link',
    label: 'Fungible Token Guide',
    href: '/build/guides/fungible-token',
    description: 'Steps to create, deploy, mint, and transfer fungible
tokens on Flow',
    customProps: {
      icon: <FontAwesomeIcon icon={faCoins} className="h-16" />,
      author: {
        name: 'Flow Blockchain',
        profileImage:
          'https://avatars.githubusercontent.com/u/62387156?s=200&v=4',
      },
    },
  },
  {
    type: 'link',
    label: 'NFT Guide',
    href: 'https://academy.ecdao.org/en/quickstarts/1-non-fungible-token-
next',
    description: 'A DApp that lets users create an empty collection, mint
some pre-loaded NFTs, and transfer them to another account on Flow
testnet.',
  }
]}

```

```
customProps: {
  icon: <FontAwesomeIcon icon={faGem} className="h-16" />,
  author: {
    name: 'Emerald City DAO',
    profileImage:
      'https://pbs.twimg.com/profileimages/1687225095557632005/tUCmv8P400x400.jpg',
    },
  },
},
{
  type: 'link',
  label: 'Walkthrough Guides',
  href: '/build/getting-started/fcl-quickstart',
  description: 'Longer form guides to help you get started with Flow',
  customProps: {
    icon: <FontAwesomeIcon icon={faBook} className="h-16" />,
    author: {
      name: 'Flow Blockchain',
      profileImage:
        'https://avatars.githubusercontent.com/u/62387156?s=200&v=4',
    },
  },
},
{
  type: 'link',
  label: 'Emerald Academy',
  href: 'https://academy.ecdao.org/en/quickstarts',
  description: 'Quickstart Tutorials for Flow created by Emerald City Dao',
  customProps: {
    icon: '/images/logos/ea-logo.png',
    author: {
      name: 'Emerald City DAO',
      profileImage:
        'https://pbs.twimg.com/profileimages/1687225095557632005/tUCmv8P400x400.jpg',
    },
  },
},
{
  type: 'link',
  label: 'Basic Concepts',
  href: '/build/basics/blocks',
  description: 'Basic Concepts of Flow Blockchain',
  customProps: {
    icon: <FontAwesomeIcon icon={faLandmark} className="h-16" />,
    author: {
      name: 'Flow Blockchain',
      profileImage:
        'https://avatars.githubusercontent.com/u/62387156?s=200&v=4',
    },
  },
},
```

```
        },
    }
}] />

# flow.md:

---
title: Why Flow
sidebarlabel: Why Flow
sidebarposition: 1
---

Why Flow
```

Flow is a fast, decentralized, and developer-friendly blockchain designed to be the foundation for a new generation of games, apps, and digital assets that power them. It is based on a unique, multi-role architecture, and designed to scale without sharding, allowing for massive improvements in speed and throughput while preserving a developer-friendly, ACID-compliant environment.

What Makes Flow Unique

- **Multi-role architecture:** Flow's node architecture allows the network to scale to serve billions of users without sharding or reducing the decentralization of consensus.
- **Resource-oriented programming:** Smart contracts on Flow are written in Cadence, an easier and safer programming language for crypto assets and apps.
- **Developer ergonomics:** This network is designed to maximize developer productivity. Examples range from upgradeable smart contracts and built-in logging support to the Flow Emulator.
- **Consumer onboarding:** Flow was designed for mainstream consumers, with payment onramps catalyzing a safe and low-friction path from fiat to crypto.

The following chapters summarize the content in this section. Read on more for details.

App Development

The Flow App Quickstart covers the Flow core concepts, including:

- **App Client:** The app client is the interface through which users interact with your app. Web and mobile applications are typical examples of app clients.
- **Smart Contract:** A smart contract is a collection of code deployed to a permanent location on the blockchain that defines the core logic for a dApp.
- **User Account:** A user account is a record on the blockchain that stores the digital assets owned by a single user.
- **Transaction:** A transaction is a code submitted to the blockchain that mutates the state of one or more user accounts and smart contracts.

- User Wallet: A user wallet is software or hardware that controls access to a user's account on the blockchain.
- Script: A script is a request made to the blockchain that returns information about the state of your dApp's smart contracts.
- Flow Client Library (FCL): The Flow Client Library is a framework that provides a standard interface to connect client applications and user wallets.

Core Contracts

The Flow blockchain implements core functionality using its own smart contract language, Cadence. The core functionality is split into a set of contracts, so-called core contracts:

- Fungible Token: The FungibleToken contract implements the Fungible Token Standard. It is the second contract ever deployed on Flow.
- Flow Token: The FlowToken contract defines the FLOW network token.
- Flow Fees: The FlowFees contract is where all the collected flow fees are gathered.
- Service Account: tracks transaction fees and deployment permissions and provides convenient methods for Flow Token operations.
- Staking Table: The FlowIDTableStaking contract is the central table that manages staked nodes, delegation, and rewards.
- Epoch Contract: The FlowEpoch contract is the state machine that manages Epoch phases and emits service events.

FLOW Token

The FLOW token is the native currency for the Flow network. Developers and users can use FLOW to transact on the network. Developers can integrate FLOW directly into their apps for peer-to-peer payments, service charges, or consumer rewards. FLOW can be held, transferred, or transacted peer-to-peer.

Technical Background

- The Flow Technical Primer is a great place to start to understand how Flow works.
- The Three technical whitepapers cover the unique innovation behind the Flow blockchain network in-depth.

Tokenomics

- To understand more about Flow's Token Economics, and the FLOW token, you can read the Flow Token Economics guide.
- FLOW tokens are Flow's native Fungible Token. To learn more about how to work with them in your applications, go [here](#).

More Concepts

If you're a developer, looking to get a better understanding of working with Flow in your applications, use the ☰ left-hand navigation to explore the concepts pages.

```
# account-abstraction.md:
```

```
---
```

```
sidebarposition: 2
```

```
---
```

Account Abstraction

Flow provides native support for key use cases that are enabled by Account Abstraction, empowering developers to deliver mainstream-ready user experiences. With Cadence, Flow was designed with these use cases in mind through the separation of the contract and transaction layers. This guide demonstrates how Flow supports key use cases that are made possible with Account Abstraction.

Multi-sig Transactions

Since accounts are smart contracts, they can be defined in order to require multiple signatures in order to execute a transaction, which unlocks a range of new users that improve the user experience for Web3 apps.

```
| Account Abstraction
| Flow
|
| -----
----- | -----
-----
```

----- |
| The move from from Externally-Owned Accounts (EOAs) to smart contract accounts enables developers to build in logic to require multiple signatures to execute transactions. | Flow has native support for multi-sig transactions since all accounts are defined as smart contracts. Flow provides support for multiple keys to be added to an account and weights can be applied to denote relative priority. |

Sponsored Transactions

The concept of paying fees to execute transactions in order to use Web3 apps can be a hurdle for newcomers as they begin to explore these experiences. In order to remove this significant point of friction in requiring newcomers to acquire crypto before they can get started with an app, developers can subsidize these costs on behalf of users.

```
| Account Abstraction
| Flow
|
| -----
----- | -----
-----
```

```
| The ERC-4337 standard introduces the concept of paymasters, which can  
enable a developer to pay the fees for a transaction for their users. |  
Flow has built-in support for 3 different roles for transactions which  
provides native support for sponsored transactions. |
```

Bundled Transactions

Developers can deliver a more streamlined user experience that reduces the amount of interruptions in the form of transaction approvals by bundling multiple transactions together into a single transaction that executes the set of operations with one signature.

```
| Account Abstraction  
| Flow  
|  
| ----- |
```

```
| The ERC-4337 standard outlines support for bundled transactions through  
a new mempool that holds user operations from smart wallets. Bundlers  
package sets of these user operations into a single transaction on the  
blockchain and return the result back to each wallet. | Since Cadence has  
an explicit separation of contracts and transactions, Flow has protocol-  
level support for bundling transactions across multiple contracts into a  
single transaction. |
```

Account Recovery

Account Abstraction enables developers to build more robust account management features for users, addressing the major pain point of losing access to assets forever if the user loses their keys to their account. Apps can enable users to recover access to their accounts and enclosed assets through social recovery or pre-approved accounts.

```
| Account Abstraction  
| Flow  
|  
| ----- |
```

```
| Smart contract accounts can be defined to include account recovery  
logic that enables users to define custom recovery methods that can rely  
on specific accounts or other validated sources. | Since all accounts are  
smart contracts, Flow has native support for account recovery and cycling  
of keys to help users regain access to accounts in a secure manner. |
```

Multi-factor Authentication

Multi-factor authentication is a broadly accepted concept in Web2 apps for secure access to accounts and Account Abstraction enables developers to deliver the same benefits to Web3 users.

```
| Account Abstraction
| Flow
|
| -----
----- | -----
----- | -----
----- | |
----- | |
----- | |
| Smart contract accounts can require a secondary factor to confirm
transactions which can be delivered in the form of familiar confirmation
channels such as email or SMS. | Since all accounts are smart contracts,
Flow has native support for multi-factor authentication as developers can
implement these security mechanisms for their users. |
```

Seamless Experience

Account Abstraction brings the potential for dramatic improvements to the user experience of Web3 apps. Developers can introduce conditions under which a user can grant a smart contract account to pre-approve transactions under certain conditions, reducing interruptions for the user to explicitly sign each transaction.

These improvements are especially notable on mobile, where users are typically met with the jarring experience of switching between apps in approve transactions.

```
| Account Abstraction
| Flow
|
| -----
----- | -----
----- | -----
----- | |
----- | |
----- | |
| Developers can build new features that streamline the user experience
of Web3 apps, such as 'session keys' that pre-approve transactions for a
period of time or setting custom limits on transaction volume or network
fees. | Since all accounts are smart contracts, Flow has support for
these new controls that enable apps to sign pre-approved transactions
based on user controls and preferences. |
```

```
# flix.md:
```

```
---
sidebarposition: 3
---
```

FLIX (Flow Interaction Templates)

Flow Interaction Templates is a standard for how contract developers, wallets, users, auditors, and applications can create, audit, and verify the intent, security, and metadata of Flow scripts and transactions, with the goal to improve the understandability and security of authorizing transactions and promote patterns for change resilient composability of applications on Flow.

Interaction Templates provide a way to use and reuse existing scripts and transactions, as well as to provide more metadata such as a human-readable title and description of what the transaction or script will do, which can be used by the developer as well as the user of the application.

By using FLIX transactions and scripts, developers don't have to write their own for common operations!

Read more about the design and purpose of FLIX in the FLIP

Using FLIX

Flow makes FLIX available through an API available at flix.flow.com.

You can query a FLIX API to get an Interaction Template. An example query looks like: <https://flix.flow.com/v1/templates?name=transfer-flow>

You can read more about how to query a FLIX API in the documentation available here: <https://github.com/onflow/flow-interaction-template-service>

:::info

The FLIX working group is currently working on a protocol to publish FLIX templates on-chain.

:::

Example

How to integrate FLIX across different developer teams? For this example there are two github repositories.

- (smart contracts) <https://github.com/onflow/hello-world-flix>
- (web development) <https://github.com/onflow/hello-world-web>

The Smart contract developer creates FLIX templates and makes them available in github, these can be versioned. Example is v0.1.0 release, the templates are available for a specific version. In this example the templates are located at:

- <https://github.com/onflow/hello-world-flix/blob/v0.1.0/cadence/templates/ReadHelloWorld.template.json>
- <https://github.com/onflow/hello-world-flix/blob/v0.1.0/cadence/templates/UpdateHelloWorld.template.json>

Developers can use FLIX templates from the smart contract github to interact with their smart contracts. They simply need the FLIX template URLs to create binding files (TypeScript or JavaScript). One major benefit is the web developers don't need to learn Cadence or copy Cadence to their repository in order to integrate with existing smart contracts.

TypeScript code generated from templates:

- <https://github.com/onflow/hello-world-web/blob/main/app/cadence/readHelloWorld.ts>
- <https://github.com/onflow/hello-world-web/blob/main/app/cadence/updateHelloWorld.ts>

:::warning

manually added "@ts-ignore" in generated file because of linting error. 'template' property is typed as "object" when it should also allow strings (url to flix template file). There is current a dev effort that will fix this linting issue.

:::

See the hello-world-web README for more information on how to generate and execute FLIX templates here flow-cli flix commands

Clients

There are currently two clients that have integrated with FLIX that you can use:

Go client <https://github.com/onflow/flixkit-go>

FCL client you read how to get started tools/clients/fcl-js/interaction-templates

(Advanced) Running a FLIX API

Flow provides an implementation of the Flow interaction template service as an open-source project. If you wish to run your own API, you can find the repository here: <https://github.com/onflow/flow-interaction-template-service>

```
# metadata-views.md:  
  
---  
title: NFT Metadata Views  
sidebarlabel: NFT Metadata Views  
---  
  
NFT Metadata Views on Flow
```

MetadataViews on Flow offer a standardized way to represent onchain metadata across different NFTs. Through its integration, developers can ensure that different platforms and marketplaces can interpret the NFT metadata in a unified manner. This means that when users visit different websites, wallets, and marketplaces, the NFT metadata will be presented in a consistent manner, ensuring a uniform experience across various platforms.

:::info

It is important to understand this document so you can make meaningful decisions about how to manage your project's metadata as support for metadata views does not happen by default. Each project has unique metadata and therefore will have to define how they expose it in unique ways.

:::

A view is a standard Cadence struct that represents a specific type of metadata,

such as a Royalty specification:

```
cadence
access(all) struct Royalty {
    /// Where royalties should be paid to
    access(all) let receiver: Capability<&{FungibleToken.Receiver}>

    /// The cut of the sale that should be taken for royalties.
    access(all) let cut: UFix64

    /// Optional description of the royalty
    access(all) let description: String
}
```

or a rarity description:

```
cadence
access(all) struct Rarity {
    /// The score of the rarity as a number
    access(all) let score: UFix64?

    /// The maximum value of score
    access(all) let max: UFix64?

    /// The description of the rarity as a string.
    access(all) let description: String?
}
```

This guide acts as a specification for the correct ways to use each metadata view.

Many of the standard metadata views do not have built-in requirements

for how they are meant to be used, so it is important for developers to understand the content of this document so third party apps can integrate with their smart contracts as easily and effectively as possible.

> If you'd like to follow along while we discuss the concepts below, you can do so by referring to the ExampleNFT contract.

Additionally, here is the source code for the ViewResolver contract and the MetadataViews contract.

Flowty has also provided a useful guide for how to manage metadata views properly in order to be compatible with their marketplace. This guide is very useful because all of their advice is generally good advice for any NFT contract, regardless of what marketplace it is using.

Two Levels of Metadata: An Overview

Metadata in Cadence is structured at two distinct levels:

1. Contract-Level Metadata: This provides an overarching description of the entire NFT collection/project.

Any metadata about individual NFTs is not included here.

2. NFT-Level Metadata: Diving deeper, this metadata relates to individual NFTs.

It provides context, describes rarity, and highlights other distinctive attributes that distinguish one NFT from another within the same collection.

While these distinct levels describe different aspects of a project, they both use the same view system for representing the metadata and the same basic function calls to query the information, just from different places.

Understanding ViewResolver and MetadataViews.Resolver

When considering Flow and how it handles metadata for NFTs, it is crucial to understand two essential interfaces: ViewResolver and MetadataViews.Resolver.

Interfaces serve as blueprints for types that specify the required fields and methods that your contract or composite type must adhere to to be considered a subtype of that interface. This guarantees that any contract asserting adherence to these interfaces will possess a consistent set of functionalities that other applications or contracts can rely on.

1. ViewResolver for Contract-Level Metadata:

- This interface ensures that contracts, particularly those encapsulating NFT collections, conform to the Metadata Views standard.
 - Through the adoption of this interface, contracts can provide dynamic metadata that represents the entirety of the collection.
2. `MetadataViews.Resolver` (`ViewResolver.Resolver` in Cadence 1.0) for NFT-Level Metadata:
- Used within individual NFT resources, this interface ensures each token adheres to the Metadata standard format.
 - It focuses on the distinct attributes of an individual NFT, such as its unique ID, name, description, and other defining characteristics.

Core Functions

Both the `ViewResolver` and `MetadataViews.Resolver` utilize the following core functions:

`getViews` Function

This function provides a list of supported metadata view types, which can be applied either by the contract (in the case of `ViewResolver`) or by an individual NFT (in the case of `MetadataViews.Resolver`).

```
cadence
access(all) fun getViews(): [Type] {
    return [
        Type<MetadataViews.Display>(),
        Type<MetadataViews.Royalties>(),
        ...
    ]
}
```

`resolveView` Function

Whether utilized at the contract or NFT level, this function's role is to deliver the actual metadata associated with a given view type.

The caller provides the type of the view they want to query as the only argument, and the view is returned if it exists, and nil is returned if it doesn't.

```
cadence
access(all) fun resolveView( view: Type): AnyStruct? {
    switch view {
        case Type<MetadataViews.Display>():
            ...
        ...
    }
    return nil
}
```

As you can see, the return values of `getViews()` can be used as arguments for `resolveView()` if you want to just iterate through all the views

that an NFT implements.

NFT-Level Metadata Implementation

NFT-level metadata addresses the unique attributes of individual tokens within a collection. It provides structured information for each NFT, including its identifier, descriptive elements, royalties, and other associated metadata. Incorporating this level of detail ensures consistency and standardization among individual NFTs, making them interoperable and recognizable across various platforms and marketplaces.

Core Properties

In the code below, an NFT has properties such as its unique ID, name, description, and others. When we add the NonFungibleToken.NFT and by extension, the MetadataViews.Resolver to our NFT resource, we are indicating that these variables will adhere to the specifications outlined in the MetadataViews contract for each of these properties. This facilitates interoperability within the Flow ecosystem and assures that the metadata of our NFT can be consistently accessed and understood by various platforms and services that interact with NFTs.

```
cadence
access(all) resource NFT: NonFungibleToken.NFT {
    access(all) let id: UInt64
    access(all) let name: String
    access(all) let description: String
    access(all) let thumbnail: String
    access(self) let royalties: [MetadataViews.Royalty]
    access(self) let metadata: {String: AnyStruct}
    ...
}
```

To make this possible though, it is vital that projects all use the standard metadata views in the same way, so third-party applications can consume them in standard ways.

For example, many metadata views have String-typed fields. It is difficult to enforce that these fields are formatted in the correct way, so it is important for projects to be diligent about how they use them. Take Traits for example, a commonly misused metadata view:

```
cadence
access(all) struct Trait {
    // The name of the trait. Like Background, Eyes, Hair, etc.
    access(all) let name: String
    ...
    ...
}
```

```
}
```

The name of the trait should be formatted in a way so that it is easy to display on a user-facing website. Many projects will use something like CamelCase for the value, so it looks like "HairColor", which is not pretty on a website. The correct format for this example would be "Hair Color". This is just one of many common view uses that projects need to be aware of to maximize the chance of success for their project.

Metadata Views for NFTs

MetadataViews types define how the NFT presents its data. When invoked, the system knows precisely which view to return, ensuring that the relevant information is presented consistently across various platforms. In this section of the document, we will explore each metadata view and describe how projects should properly use them.

Display

This view provides the bare minimum information about the NFT suitable for listing or display purposes. When the Display type is invoked, it dynamically assembles the visual and descriptive information that is typically needed for showcasing the NFT in marketplaces or collections.

```
cadence
case Type<MetadataViews.Display>():
    return MetadataViews.Display(
        name: self.name,
        description: self.description,
        thumbnail: MetadataViews.HTTPFile(
            url: self.thumbnail
        )
    )
```

If the thumbnail is a HTTP resource:

```
cadence
thumbnail : MetadataViews.HTTPFile(url: Please put your url here)
```

If the thumbnail is an IPFS resource:

```
cadence
//
thumbnail : MetadataViews.IPFSFile(
    cid: thumbnail cid, // Type <String>
    path: ipfs path // Type <String?> specify path if the cid is a
    folder hash, otherwise use nil here
```

```
)  
  
!MetadataViews.Display  
  
:::info
```

Note about SVG files on-chain: SVG field should be sent as thumbnailURL, should be base64 encoded, and should have a dataURI prefix, like so:

```
data:image/svg+xml;base64,PHN2ZyB3aWR0aD0iMSIgaGVpZ2h0PSIxIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciPjxyZWN0IHdpZHRoPSIxMDALIiBoZWlnaHQ9IjEwMCUiIGZpbGw9InJlZCIvPjwvc3ZnPg==
```

```
:::
```

Editions

The Editions view provides intricate details regarding the particular release of an NFT within a set of NFTs with the same metadata. This can include information about the number of copies in an edition, the specific NFT's sequence number within that edition, or its inclusion in a limited series. When the Editions view is queried, it retrieves this data, providing collectors with the information they need to comprehend the rarity and exclusivity of the NFT they are interested in.

An NFT can also be part of multiple editions, which is why the Editions view can hold any number of Edition structs in an array.

For example, if an NFT is number 11 of 30 of an exclusive edition, the code to return the Editions view would look like this:

```
cadence  
case Type<MetadataViews.Editions>():  
    let editionInfo = MetadataViews.Edition(  
        name: "Example NFT Edition",  
        number: 11,  
        max: 30  
    )  
    return MetadataViews.Editions([editionInfo])
```

Serial Number Metadata

The Serial metadata provides the unique serial number of the NFT, akin to a serial number on a currency note or a VIN on a car. This serial number is a fundamental attribute that certifies the individuality of each NFT and is critical for identification and verification processes.

Serial numbers are expected to be unique among other NFTs from the same project.

Many projects are already using the NFT resource's globally unique UUID

as the ID already, so they will typically also use that as the serial number.

```
cadence
case Type<MetadataViews.Serial>():
    return MetadataViews.Serial(self.uuid)
```

Royalties Metadata

Royalty information is vital for the sustainable economics of the creators in the NFT space.

The Royalties metadata view defines the specifics of any royalty agreements in place, including the percentage of sales revenue that will go to the original creator or other stakeholders on secondary sales.

Each royalty view contains a fungible token receiver capability where royalties should be paid:

```
cadence
access(all) struct Royalty {

    access(all) let receiver: Capability<&{FungibleToken.Receiver}>
    access(all) let cut: UFix64
}
```

here is an example of how an NFT might return a Royalties view:

```
cadence
case Type<MetadataViews.Royalties>():
    // Assuming each 'Royalty' in the 'royalties' array has 'cut' and
    'description' fields
    let royalty =
        MetadataViews.Royalty(
            // The beneficiary of the royalty: in this case, the contract
            account
            receiver:
ExampleNFT.account.capabilities.get<&AnyResource{FungibleToken.Receiver}>
(/public/GenericFTReceiver),
            // The percentage cut of each sale
            cut: 0.05,
            // A description of the royalty terms
            description: "Royalty payment to the original creator"
        )
    }
    return MetadataViews.Royalties(detailedRoyalties)
```

```

If someone wants to make a listing for their NFT on a marketplace,
the marketplace can check to see if the royalty receiver
accepts the seller's desired fungible token by calling
the receiver.getSupportedVaultTypes(): {Type: Bool}
function via the receiver reference:
cadence
let royaltyReceiverRef = royalty.receiver.borrow()
?? panic("Could not borrow a reference to the receiver")
let supportedTypes = receiverRef.getSupportedVaultTypes()
if supportedTypes[royalty.getType()] {
    // The type is supported, so you can deposit
    receiverRef.deposit(<-royalty)
} else {
    // if it is not supported, you can do something else,
    // like revert, or send the royalty tokens to the seller instead
}

```

If the desired type is not supported, the marketplace has a few options. They could either get the address of the receiver by using the receiver.owner.address field and check to see if the account has a receiver for the desired token, they could perform the sale without a royalty cut, or they could abort the sale since the token type isn't accepted by the royalty beneficiary.

You can see example implementations of royalties in the ExampleNFT contract and the associated transactions and scripts. NFTs are often sold for a variety of currencies, so the royalty receiver should ideally be a fungible token switchboard receiver that forwards any received tokens to the correct vault in the receiving account.

Important instructions for royalty receivers

If you plan to set your account as a receiver of royalties, you'll likely want to be able to accept as many token types as possible. This is possible with the FungibleTokenSwitchboard. If you initialize a switchboard in your account, it can accept any generic fungible token and route it to the correct vault in your account.

Therefore, if you want to receive royalties, you should set up your account with the setuproyaltyaccountbypaths.cdc transaction.

This will link generic public path from MetadataViews.getRoyaltyReceiverPublicPath() to the capability paths and types that you provide as arguments. Then you can use that public path and capability for your royalty receiver.

External URL Metadata

The ExternalURL view returns to an associated webpage URL, providing additional content or information about the NFT. This can be a website, social media page, or anything else related to the project that uses a URL.

```
cadence
case Type<MetadataViews.ExternalURL>():
    return MetadataViews.ExternalURL("<https://example-nft.onflow.org/>".concat(self.id.toString()))
```

Traits Metadata

The Trait view type encapsulates the unique attributes of an NFT, like any visual aspects or category-defining properties. These can be essential for marketplaces that need to sort or filter NFTs based on these characteristics.

By returning trait views as recommended, you can fit the data in the places you want.

```
cadence
access(all) struct Trait {
    // The name of the trait. Like Background, Eyes, Hair, etc.
    access(all) let name: String

    // The underlying value of the trait
    access(all) let value: AnyStruct

    // displayType is used to show some context about what this name
    and value represent
    // for instance, you could set value to a unix timestamp, and
    specify displayType as "Date" to tell
    // platforms to consume this trait as a date and not a number
    access(all) let displayType: String?

    // Rarity can also be used directly on an attribute.
    // This is optional because not all attributes need to contribute
    to the NFT's rarity.
    access(all) let rarity: Rarity?
```

The traits view is extremely important to get right, because many third-party apps

and marketplaces are heavily reliant on it to properly display the entirety of your NFTs.

For example, the names and values of the traits are likely going to be displayed

on a user-facing website, so it is important to return them in a presentable form, such as First Name, instead of firstname or firstName.

Additionally, limit your value field to primitive types like String, Int, or Bool.

Additionally, the displayType is important as well, because it tells websites how to display the trait properly. Developers should not just default to String or Integer for all their display types. When applicable, the display types to accurately reflect the data that needs to be displayed.

!MetadataViews.Traits

Note: Always prefer wrappers over single views

When exposing a view that could have multiple occurrences on a single NFT, such as Edition, Royalty, Media or Trait the wrapper view should always be used (such as Editions, Royalties, etc), even if there is only a single occurrence. The wrapper view is always the plural version of the single view name and can be found below the main view definition in the MetadataViews contract.

When resolving the view, the wrapper view should be the returned value, instead of returning the single view or just an array of several occurrences of the view.

```
cadence
access(all) fun resolveView( view: Type): AnyStruct? {
    switch view {
        case Type<MetadataViews.Editions>():
            let editionInfo = MetadataViews.Edition(name: "Example NFT
Edition", number: self.id, max: nil)
            let editionList: [MetadataViews.Edition] = [editionInfo]
            // return the wrapped view
            return MetadataViews.Editions(
                editionList
            )
    }
}
```

Contract-Level Metadata Implementation

Contract-level metadata provides a holistic view of an NFT collection, capturing overarching attributes and contextual information about the entire set, rather than specifics of individual tokens. These views describe attributes at the collection or series level rather than individual NFTs. These views should still be queryable via individual NFTs though. One can accomplish this by just forwarding the call from the NFT's resolveView() method to the contract's resolveView() method, like so:

```
cadence
```

```

/// this line is in ExampleNFT.NFT.resolveView()
case Type<MetadataViews.NFTCollectionDisplay>():
    return ExampleNFT.getCollectionDisplay(nftType:
Type<@ExampleNFT.NFT>())

NFTCollectionData

This view provides paths and types related to the NFT collection's
storage
and access within the smart contract. The information in this view
is critical for understanding how to interact with a collection.

cadence
case Type<MetadataViews.NFTCollectionData>():
    return MetadataViews.NFTCollectionData(
        // where should the collection be saved?
        storagePath: ExampleNFT.CollectionStoragePath,
        // where to borrow public capabilities from?
        publicPath: ExampleNFT.CollectionPublicPath,
        // Important types for how the collection should be linked
        publicCollection: Type<&ExampleNFT.Collection>(),
        publicLinkedType: Type<&ExampleNFT.Collection>(),
        // function that can be accessed to create an empty collection
for the project
        createEmptyCollectionFunction: (fun():
@{NonFungibleToken.Collection} {
            return <-ExampleNFT.createEmptyCollection(nftType:
Type<@ExampleNFT.NFT>())
        })
    )

```

Here, NFTCollectionData is specifying several important elements related to how the collection is stored and accessed on the Flow blockchain.

It provides information on storage paths and access control paths for both public and private data, as well as linked types that specify what capabilities are publicly available (like collection, receiver, or provider interfaces).

NFTCollectionDisplay

This view describes the collection with visual elements and metadata that are useful for display purposes, such as in a marketplace or gallery.

Many third party apps need this in order to display high-level information about an NFT project properly.

```

cadence
case Type<MetadataViews.NFTCollectionDisplay>():
    let media = MetadataViews.Media(
        file: MetadataViews.HTTPFile(

```

```

        url: "<https://assets.website-
files.com/5f6294c0c7a8cdd643b1c820/5f6294c0c7a8cda55cb1c936FlowWordmark.s
vg>" ,
        mediaType: "image/svg+xml"
    )
    return MetadataViews.NFTCollectionDisplay(
        name: "The Example Collection",
        description: "This collection is used as an example to help you
develop your next Flow NFT.",
        externalURL: MetadataViews.ExternalURL("<https://example-
nft.onflow.org>"),
        squareImage: media,
        bannerImage: media,
        socials: {
            "twitter": MetadataViews.ExternalURL("<https://twitter.com/flowblockchain>")
        }
    )
}

```

In the example above, the `NFTCollectionDisplay` not only offers fundamental metadata like the collection's name and description but also provides image URLs for visual representations of the collection (`squareImage` and `bannerImage`) and external links, including social media profiles.

`!MetadataViews.CollectionDisplay`

Contract-borrowing Metadata

With the contract borrowing feature, the `ViewResolver` interface on contracts can be borrowed directly without needing to import the contract first.

Views can be resolved directly from there.

As an example, you might want to allow your contract to resolve `NFTCollectionData` and `NFTCollectionDisplay` so that platforms do not need to find an NFT that belongs to your contract to get information about how to set up or show your collection.

```

cadence
import ViewResolver from 0xf8d6e0586b0a20c7
import MetadataViews from 0xf8d6e0586b0a20c7

access(all) fun main(addr: Address, name: String): StoragePath? {
    let t = Type<MetadataViews.NFTCollectionData>()
    let borrowedContract =
getAccount(addr).contracts.borrow<&ViewResolver>(name: name) ??
panic("contract could not be borrowed")

    let view = borrowedContract.resolveView(t)
    if view == nil {
        return nil
    }
}

```

```
}

let cd = view! as! MetadataViews.NFTCollectionData
return cd.storagePath
}
```

Will Return
cadence
{"domain":"storage","identifier":"exampleNFTCollection"}

More

Understanding MetadataViews and the core functions associated with it is crucial for developers aiming to deploy NFTs on Flow. With these views and functions, NFTs can maintain a consistent presentation across various platforms and marketplaces and foster interoperability between contracts and applications in the Flow ecosystem. To gain a deeper understanding of implementing the MetadataView standard, check out our documentation on "How to Create an NFT Project on Flow". It provides an introduction to integrating these standards into your NFT contracts.

- See the API reference for a complete list of Metadata functions
- Check out an Example NFT project implementing MetadataViews
- Read the NFT Guide for an introduction to implementation

randomness.md:

```
---
title: Flow On-chain Randomness in Cadence
sidebarlabel: VRF (Randomness) in Cadence
---
```

Randomness on FLOW

Flow enhances blockchain functionality and eliminates reliance on external oracles by providing native onchain randomness at the protocol level. This secure, decentralized feature empowers developers to build a variety of applications with truly unpredictable, transparent, and fair outcomes, achieved with greater efficiency.

Flow's onchain randomness delivers immediate random values within smart contracts, bypassing the latency and complexity of oracle integration. Developers can obtain verifiably random results with a single line of Cadence code, streamlining the development process and enhancing the performance of decentralized applications.

Use Cases of Onchain Randomness

- Gaming: Integrates fairness and unpredictability into gameplay, enhancing user engagement without delays.

- NFTs: Facilitates the creation of uniquely randomized traits in NFTs quickly, adding to their rarity and value.
- Lotteries & Draws: Offers instant and verifiably fair random selection for lotteries, solidifying trust in real-time.
- DeFi Protocols: Enables rapid and innovative random reward systems within decentralized finance.
- DAOs: Assists in unbiased voting and task assignments through immediate randomness.
- Broad Applications: Extends to any domain requiring impartial randomization, from asset distribution to security mechanisms, all with the added benefit of on-demand availability.

History of the Distributed Randomness Beacon

Within the Flow protocol, the heart of randomness generation lies in the "Distributed Randomness Beacon".

This module generates randomness that is distributed across the network while adhering to established cryptographic and security standards.

The output from the randomness beacon is a random source for each block that is unpredictable and impartial.

For over three years, the beacon has ensured protocol security by selecting which consensus node gets to propose the next block and assigning verification nodes to oversee block computations. For those interested in a more detailed exploration of the randomness beacon and its inner workings, you can read the technical deep dive on the Flow forum.

The History and Limitations of unsafeRandom (Now Deprecated)

Cadence has historically provided the unsafeRandom function to return a pseudo-random number. The stream of random numbers produced was potentially unsafe in the following two regards:

1. The sequence of random numbers is potentially predictable by transactions within the same block and by other smart contracts calling into your smart contract.
2. A transaction calling into your smart contract can potentially bias the sequence of random numbers which your smart contract internally generates. Currently, the block hash seeds unsafeRandom. Consensus nodes can easily bias the block hash and influence the seed for unsafeRandom.

```
<Callout type="warning">
⚠ Note unsafeRandom is deprecated since the Cadence 1.0 release.
</Callout>
```

Guidelines for Safe Usage

For usage of randomness where result abortion is not an issue, it is recommended to use the Cadence built-in function revertibleRandom. revertibleRandom returns a pseudo-random number and is based on the Distributed Randomness Beacon.

cadence

```

// Language reference:
// https://cadence-lang.org/docs/language/built-in-
functions#revertiblerandom
// Run the snippet here: https://academy.ecdao.org/en/snippets/cadence-
random
access(all) fun main(): UInt64 {
    // Simple assignment using revertibleRandom - keep reading docs for
safe usage!
    let rand: UInt64 = revertibleRandom()
    return rand
}

```

It is notable that the random number generation process is unpredictable (for miners unpredictable at block construction time and for cadence logic unpredictable at time of call), verifiable, uniform, as well as safe from bias by miners and previously-running Cadence code.

Protocol improvements (documented in FLIP 120) expose the randomness beacon to the FVM and Cadence where it can be used to draw safe randoms without latency.

Although Cadence exposes safe randomness generated by the Flow protocol via revertibleRandom, there is an additional safety-relevant aspect that developers need to be mindful about.

The revertibleRandom function can be used safely in some applications where the transaction results are not deliberately reverted after the random number is revealed (i.e. a trusted contract distributing random NFTs to registered users or onchain lucky draw).

However, if applications require a non-trusted party (for instance app users) to submit a transaction calling a randomized (non-deterministic) contract, the developer must explicitly protect the stream of random numbers to not break the security guarantees:

```

<Callout type="warning">
  ⚠ A transaction can atomically revert all its action during its runtime
  and abort. Therefore, it is possible for a transaction calling into your
  smart contract to post-select favorable results and revert the
  transaction for unfavorable results.
</Callout>

```

In other words, transactions submitted by a non-trusted party are able to reject their results after the random is revealed.

```

<Callout type="info">
  ⚡ Post-selection - the ability for transactions to reject results they
  don't like - is inherent to any smart contract platform that allows
  transactions to roll back atomically. See this very similar Ethereum
  example.
</Callout>

```

The central aspect that a contract developer needs to think about is the following scenario:

- Imagine an adversarial user that is sending a transaction that calls your smart contract.
- The transaction includes code that runs after your smart contract returns and inspects the outcome.
- If the outcome is unfavorable (based on some criteria codified in the transaction), the transaction aborts itself.

As an example, consider a simple coin toss randomized contract where users can bet any amount of tokens against a random binary output. If the coin toss contract outputs 1, the user doubles their bet. If the coin toss contract outputs 0, the user loses their bet in favor of the coin toss.

Although the user (or the honest coin toss contract) cannot predict or bias the outcome, the user transaction can check the randomized result and cancel the transaction if they are losing their bet. This can be done by calling an exception causing the transaction to error. All temporary state changes are cancelled and the user can repeat the process till they double their bet.

Commit-Reveal Scheme

The recommended way to mitigate the problems above is via a commit-reveal scheme. The scheme involves two steps: commit and reveal. During the commit phase, the user transaction commits to accepting the future output of a smart contract where the last remaining input is an unknown random source. The smart contract stores this commitment on the blockchain. At the current level of optimization, the reveal phase can start as early as the next block, when the "future" beacon's source of randomness becomes available. The reveal phase can be executed at any block after that, now that the commitment to a past block is stored on-chain. With a second transaction, the smart contract can be executed to explicitly generate the random outputs.

There are ideas how to further optimize the developer experience in the future. For example, a transaction could delegate part of its gas to an independent transaction it spawns. Conceptually, also this future solution would be a commit-and-reveal scheme, just immediately happening within the same block. Until we eventually get to this next level, developers may need to implement their own commit-reveal. In Cadence, it is clean and short.

FLIP 123

On Flow, we have absorbed all security complexity into the platform.

FLIP 123: On-chain Random beacon history for commit-reveal schemes was introduced to provide a safe pattern to use randomness in transactions so that it's not possible to revert unfavorable randomized transaction results.

We recommend this approach as a best-practice example for implementing a commit-reveal scheme in Cadence. The `RandomBeaconHistory` contract provides a convenient archive, where for each past block height (starting

Nov 2023) the respective “source of randomness” can be retrieved. The RandomBeaconHistory contract is automatically executed by the system at each block to store the next source of randomness value.

```
<Callout type="info">
```

💡 While the commit-and-reveal scheme mitigates post-selection of results by adversarial clients, Flow's secure randomness additionally protects against any pre-selection vulnerabilities (like biasing attacks by byzantine miners).

```
</Callout>
```

A commit-reveal scheme can be implemented as follows. The coin toss example described earlier will be used for illustration:

- When a user submits a bidding transaction, the bid amount is transferred to the coin toss contract, and the block height where the bid was made is stored. This is a commitment by the user to use the SoR at the current block. Note that the current block's SoRA isn't known to the transaction execution environment, and therefore the transaction has no way to inspect the random outcome and predict the coin toss result. The current block's SoRA is only available once added to the history core-contract, which only happens at the end of the block's execution. The user may also commit to using an SoR of some future block, which is equally unknown at the time the bid is made.
- The coin toss contract may grant the user a limited window of time (i.e a block height range) to send a second transaction for resolving the results and claim any winnings. Failing to do so, the bid amount remains in the coin toss contract.
- Within that reveal transaction, the user calls the coin toss contract, looks up the block height at which the block was committed and checks that it has already passed. The contract queries that block's SoRA from the core-contract RandomBeaconHistory via block height.
- The coin toss contract uses a PRG seeded with the queried SoRA and diversified using a specific information to the use-case (a user ID or resource ID for instance). Diversification does not add new entropy, but it avoids generating the same outcome for different use-cases. If a diversifier (or salt) isn't used, all users that committed a bid on the same block would either win or lose.
- The PRG is used to generate the random result and resolve the bid. Note that the user can make the transaction abort after inspecting a losing result. However, the bid amount would be lost anyway when the allocated window expires.

The following lines of code illustrate a random coin toss that cannot be gamed or biased. The reveal-and-commit scheme prevent clients from post-selecting favorable outcomes.

```
cadence
// The code below is taken from the example CoinToss contract found in
this project repo
// Source: https://github.com/onflow/random-coin-toss

/// --- Commit ---
```

```

/// In this method, the caller commits a bet. The contract takes note of
the
/// block height and bet amount, returning a Receipt resource which is
used
/// by the better to reveal the coin toss result and determine their
winnings.
access(all) fun commitCoinToss(bet: @FungibleToken.Vault): @Receipt {
    let receipt <- create Receipt(
        betAmount: bet.balance
    )
    // commit the bet
    // self.reserve is a @FungibleToken.Vault field defined on the app
contract
    // and represents a pool of funds
    self.reserve.deposit(from: <-bet)

    emit CoinTossBet(betAmount: receipt.betAmount, commitBlock:
receipt.commitBlock, receiptID: receipt.uuid)

    return <- receipt
}

/// --- Reveal ---
/// Here the caller provides the Receipt given to them at commitment. The
contract
/// then "flips a coin" with randomCoin(), providing the committed block
height
/// and salting with the Receipts unique identifier.
/// If result is 1, user loses, if it's 0 the user doubles their bet.
/// Note that the caller could condition the revealing transaction, but
they've
/// already provided their bet amount so there's no loss for the contract
if
/// they do
access(all) fun revealCoinToss(receipt: @Receipt): @FungibleToken.Vault {
    pre {
        receipt.commitBlock <= getCurrentBlock().height: "Cannot reveal
before commit block"
    }

    let betAmount = receipt.betAmount
    let commitBlock = receipt.commitBlock
    let receiptID = receipt.uuid
        // self.randomCoin() errors if commitBlock <= current block
height in call to
        // RandomBeaconHistory.sourceOfRandomness()
    let coin = self.randomCoin(atBlockHeight: receipt.commitBlock, salt:
receipt.uuid)

    destroy receipt

    if coin == 1 {
        emit CoinTossReveal(betAmount: betAmount, winningAmount: 0.0,
commitBlock: commitBlock, receiptID: receiptID)
    }
}

```

```

        return <- FlowToken.createEmptyVault()
    }

    let reward <- self.reserve.withdraw(amount: betAmount 2.0)

    emit CoinTossReveal(betAmount: betAmount, winningAmount:
reward.balance, commitBlock: commitBlock, receiptID: receiptID)

    return <- reward
}

```

Which random function should be used:

While both are backed by Flow's Randomness Beacon it is important for developers to mindfully choose between revertibleRandom or seeding their own PRNG utilizing the RandomBeaconHistory smart contract:

- With revertibleRandom a developer is calling the transaction environment,
which has the power to abort and revert if it doesn't like revertibleRandom's outputs.
revertibleRandom is only suitable for smart contract functions that exclusively run within the trusted transactions.
- In contrast, the RandomBeaconHistory contract is key for effectively implementing a commit-reveal scheme, where the transaction is non-trusted and may revert the random outputs.

During the commit phase, the user commits to proceed with a future source of randomness,

which is only revealed after the commit transaction concluded.

For each block, the RandomBeaconHistory automatically stores the generated source of randomness.

At the time of revealing the source, the committed source becomes a past-block source that can be queried through the history contract.

Adding a safe pattern to reveal randomness without the possibility of conditional transaction reversion unlocks applications relying on randomness. By providing examples of commit-reveal implementations we hope to foster a more secure ecosystem of decentralized applications and encourage developers to build with best practices.

In simpler terms, the native secure randomness provided by the protocol can now be safely utilized within Cadence smart contracts and is available to all developers on Flow and the FVM.

An Invitation to Build

Flow's onchain randomness opens new doors for innovation in web3, offering developers the tools to create fair and transparent decentralized applications. With this feature, new possibilities emerge—from enhancing gameplay in decentralized gaming to ensuring the integrity of smart contract-driven lotteries or introducing novel mechanisms in DeFi.

This is an invitation for builders and creators: leverage Flow's onchain randomness to distinguish your projects and push the boundaries of what's possible. Your imagination and code have the potential to forge new paths in the web3 landscape. So go ahead and build; the community awaits the next big thing that springs from true randomness.

Learn More

If you'd like to dive deeper into Flow's onchain randomness, here's a list of resources:

- To learn more about how randomness works under the hood, see the [forum post](#).
- These documents provide a more in-depth technical understanding of the updates and enhancements to the Flow blockchain.
 - FLIP 120: Update unsafeRandom function
 - FLIP 123: On-chain Random beacon history for commit-reveal schemes
- To see working Cadence code, explore the coin toss example on [GitHub](#).

```
# index.md:  
---  
sidebarposition: 5  
title: App Architecture  
description: Describes building self-custody and app custody applications  
on Flow Blockchain.  
sidebarcustomprops:  
  icon: 🏛  
---
```

App Architecture on Flow Blockchain

The Flow blockchain, with its focus on scalability and user-centric design, offers a unique environment for app development. Designed from the ground up, Flow prioritizes user experience, aiming to bridge the gap to mainstream adoption without compromising on decentralization or security.

While the Flow blockchain offers various architectural patterns, the recommended approaches for developers are Self-Custody and App Custody. These choices align with Flow's ethos of user-centric design and flexibility.

Self-Custody Architecture

In a self-custody architecture, users retain direct control over their private keys and assets. This model emphasizes user sovereignty and decentralization, placing the responsibility of asset management squarely on the user's shoulders. While it offers the highest degree of control, it also demands a certain level of technical knowledge and awareness from the user, which can sometimes lead to a more complex user experience.

!self-custody.png

Architectural Elements:

- Wallet: A wallet where users store their private keys and sign transactions.
- Frontend: Interfaces directly with the user and their wallet for signing transactions.
- Method of Talking to Chain: Through FCL directly.

Benefits:

- Control: Users maintain full ownership of their assets and transactions.
- Security: Direct management of private keys reduces potential points of failure.

Considerations:

- User Experience: The direct control model can lead to a more complex user experience, especially for those unfamiliar with blockchain. Typically, all transactions must be approved by the user, which can be cumbersome.
- Key Management: Users are solely responsible for managing, backing up, and ensuring the safe generation and storage of their keys.

Ideal Use Cases:

- Decentralized Finance (DeFi): Users interacting with financial protocols while maintaining full control.
- Web3 Native Users: Those familiar with blockchain technology and key management.

App Custody Architecture

App custody on Flow offers a unique approach to key management and user experience. Unlike traditional app custody solutions on other blockchains, Flow's App Custody architecture introduces features like account linking and walletless onboarding. These features ensure that while users enjoy a seamless experience, they still have the option to link their accounts and move their assets freely around the Flow ecosystem, providing a balanced approach to key management.

!app-custody.png

Architectural Elements:

- Wallet: Both user custody and app custody wallets coexist.
- Frontend: Interfaces for both wallet types and features for account linking and walletless onboarding.
- Backend: Manages app-custody user keys and assets. Also supports direct blockchain interactions.
- Method of Interacting with the Chain: Both direct FCL calls and backend-managed interactions.

- Payment Rails: Flexible methods, accommodating both direct and managed transactions.

Benefits:

- Walletless Onboarding: Users can start interacting with the app using traditional, familiar web2 mechanics without the initial need for a blockchain wallet.
- Streamlined Experience: Transactions can be processed without constant user approval, offering a smoother user journey.
- Open Ecosystem with Account Linking: Users can link their accounts, ensuring they can move their assets freely around the Flow ecosystem without being locked into a single application.
- Flexibility: Cater to both tech-savvy users and newcomers without compromising on security.
- Platform Versatility: The abstraction of the user wallet allows for integration with various platforms, including Unity games, consoles, and mobile applications.

Considerations:

- Complexity: Implementing app custody can be more intricate.
- Trust: Users need to trust the dApp with certain aspects of their data and assets.
- Legal Implications: Operating with app custody may come with legal considerations, depending on jurisdiction and the nature of the dApp. It's essential to consult with legal professionals to ensure compliance.

Ideal Use Cases:

- Gaming: Seamless gaming without constant transaction approvals.
- Social Media Platforms: Earning tokens for content without initial blockchain familiarity.
- Loyalty Programs: Earning rewards without deep blockchain setup.

Wrapping Up

Selecting the right architecture is crucial when developing an app on the Flow blockchain. Your choice will influence not only the technical aspects but also the user experience and overall trust in your application. While Flow offers the tools and flexibility to cater to various needs, it's up to developers to harness these capabilities effectively. Whether you opt for a self-custody or app custody approach, ensure that your decision aligns with the core objectives of your app and the expectations of your users. Making informed architectural decisions will lay a strong foundation for your app's success.

```
# accounts.md:
```

```
---
```

```
sidebarposition: 2
```

```
title: Accounts
```

```
--
```

:::info

Are you an EVM developer looking for information about EVM Accounts on Flow? If so, check out the EVM specific documentation here

:::

Accounts

An account on Flow is a record in the chain state that holds the following information:

- Address: unique identifier for the account
- Public Keys: public keys authorized on the account
- Code: Cadence contracts deployed to the account
- Storage: area of the account used to store resource assets.

Accounts and their keys are needed to sign transactions that change the Flow blockchain state. To execute a transaction, a small amount of Flow, called a "Fee" must be paid by the account or subsidized by a wallet or service. Flow allocates a fixed amount of storage to each account for saving data structures and Resources. Flow allocates a fixed amount of storage to each account for saving data structures and Resources.

An account may also contain contract code which transactions and scripts can interact with to query or mutate the state of the blockchain.

A simple representation of an account:

!Screenshot 2023-08-16 at 16.43.07.png

Address

A Flow address is represented as 16 hex-encoded characters (usually prefixed with 0x to indicate hex encoding). Unlike Bitcoin and Ethereum, Flow addresses are not derived from cryptographic public keys. Instead, each Flow address is assigned by the Flow protocol using an on-chain deterministic sequence. The sequence uses an error detection code to guarantee that all addresses differ with at least 2 hex characters. This makes typos resulting in accidental loss of assets not possible.

This decoupling is a unique advantage of Flow, allowing for multiple public keys to be associated with one account, or for a single public key to be used across several accounts.

Balance

Each Flow account created on Mainnet will by default hold a Flow vault that holds a balance and is part of the FungibleToken standard. This balance is used to pay for transaction fees and storage fees. More on that in the fees document.

:::warning

The minimum amount of FLOW an account can have is 0.001.

:::

This minimum storage fee is provided by the account creator and covers the cost of storing up to 100kB of data in perpetuity. This fee is applied only once and can be "topped up" to add additional storage to an account. The minimum account reservation ensures that most accounts won't run out of storage capacity if anyone deposits anything (like an NFT) to the account.

Maximum available balance

Due to the storage restrictions, there is a maximum available balance that user can withdraw from the wallet. The core contract `FlowStorageFees` provides a function to retrieve that value:

```
cadence
import "FlowStorageFees"

access(all) fun main(accountAddress: Address): UFix64 {
    return FlowStorageFees.defaultTokenAvailableBalance(accountAddress)
}
```

Alternatively developers can use `availableBalance` property of the `Account`

```
cadence
access(all) fun main(address: Address): UFix64 {
    let acc = getAccount(address)
    let balance = acc.availableBalance

    return balance
}
```

Contracts

An account can optionally store multiple Cadence contracts. The code is stored as a human-readable UTF-8 encoded string which makes it easy for anyone to inspect the contents.

Storage

Each Flow account has an associated storage and capacity. The account's storage used is the byte size of all the data stored in the account's storage. An account's storage capacity is directly tied to the balance of Flow tokens an account has. An account can, without any additional cost, use any amount of storage up to its storage capacity. If a transaction puts an account over storage capacity or drops an account's balance below the minimum 0.001 Flow tokens, that transaction fails and is reverted.

Account Keys

Flow accounts can be configured with multiple public keys that are used to control access. Owners of the associated private keys can sign transactions to mutate the account's state.

During account creation, public keys can be provided which will be used when interacting with the account. Account keys can be added, removed, or revoked by sending a transaction. This is radically different from blockchains like Ethereum where an account is tied to a single public/private key pair.

Each account key has a weight that determines the signing power it holds.

:::warning

A transaction is not authorized to access an account unless it has a total signature weight greater than or equal to 1000, the weight threshold.

:::

For example, an account might contain 3 keys, each with 500 weight:

!Screenshot 2023-08-16 at 16.28.58.png

This represents a 2-of-3 multi-sig quorum, in which a transaction is authorized to access the account if it receives signatures from at least 2 out of 3 keys.

An account key contains the following attributes:

- ID used to identify keys within an account
- Public Key raw public key (encoded as bytes)
- Signature algorithm (see below)
- Hash algorithm (see below)
- Weight integer between 0-1000
- Revoked whether the key has been revoked or it's active
- Sequence Number is a number that increases with each submitted transaction signed by this key

Signature and Hash Algorithms

The signature and hashing algorithms are used during the transaction signing process and can be set to certain predefined values.

There are two curves commonly used with the ECDSA algorithm, secp256r1 (OID 1.2.840.10045.3.1.7, also called the "NIST P-256." this curve is common for mobile secure enclave support), and secp256k1 (OID 1.3.132.0.10, the curve used by "Bitcoin"). Please be sure to double-check which parameters you are using before registering a key, as presenting a key using one of the curves under the code and format of the other will generate an error.

Algorithm	Curve	ID	Code
-----------	-------	----	------

----- ----- ----- ----
ECDSA P-256 ECDSAP256 2
ECDSA secp256k1 ECDSAssecp256k1 3

Please note that the codes listed here are for the signature algorithms as used by the node API, and they are different from the ones defined in Cadence

Algorithm	Output Size	ID	Code
SHA-2	256	SHA2256	1
SHA-3	256	SHA3256	3

Both hashing and signature algorithms are compatible with each other, so you can freely choose from the set.

Locked / Keyless Accounts

An account on Flow doesn't require keys in order to exist, but this makes the account immutable since no transaction can be signed that can change the account. This can be useful if we want to freeze an account contract code and it elegantly solves the problem of having multiple account types (as that is the case for Ethereum).

!Screenshot 2023-08-16 at 18.59.10.png

You can achieve keyless accounts by either removing an existing public key from an account signing with that same key and repeating that action until an account has no keys left, or you can create a new account that has no keys assigned. With account linking you can also have a child account that has no keys but is controlled by the parent.

:::danger

Be careful when removing keys from an existing account, because once an account's total key weights sum to less than 1000, it can no longer be modified.

:::

Multi-Sig Accounts

Creating a multi-signature account is easily done by managing the account keys and their corresponding weight. To repeat, in order to sign a transaction the keys used to sign it must have weights that sum up to at least 1000. Using this information we can easily see how we can achieve the following cases:

2-of-3 multi-sig quorum

!Screenshot 2023-08-16 at 19.34.44.png

3-of-3 multi-sig quorum

!Screenshot 2023-08-16 at 19.34.55.png

1-of-2 signature

!Screenshot 2023-08-16 at 19.34.51.png

Key Format

We are supporting ECDSA with the curves P-256 and secp256k1. For these curves, the public key is encoded into 64 bytes as X||Y where || is the concatenation operator.

- X is 32 bytes and is the big endian byte encoding of the x-coordinate of the public key padded to 32, i.e. $X=x31||x30||\dots||x0$ or $X = x31256^{31} + \dots + xi256^i + \dots + x0$.
- Y is 32 bytes and is the big endian byte encoding of the y-coordinate of the public key padded to 32, i.e. $Y=y31||y30||\dots||y0$ or $Y = y31256^{31} + \dots + yi256^i + \dots + y0$

Account Creation

Accounts are created on the Flow blockchain by calling a special create account Cadence function. Once an account is created we can associate a new key with that account. Of course, all that can be done within a single transaction. Keep in mind that there is an account creation fee that needs to be paid. Account creation fees are relatively low, and we expect that wallet providers and exchanges will cover the cost when a user converts fiat to crypto for the first time.

For development purposes, you can use Flow CLI to easily create emulator, testnet and mainnet accounts. The account creation fee is paid by a funding wallet so you don't need a pre-existing account to create it.

Key Generation

Keys should be generated in a secure manner. Depending on the purpose of the keys different levels of caution need to be taken.

:::warning

Anyone obtaining access to a private key can modify the account the key is associated with (assuming it has enough weight). Be very careful how you store the keys.

:::

For secure production keys, we suggest using key management services such as Google key management or Amazon KMS, which are also supported by our CLI and SDKs. Those services are mostly great when integrated into your application. However, for personal use, you can securely use any existing wallets as well as a hardware Ledger wallet.

Service Accounts

Flow Service Account

The Service Account is a special account in Flow that has special permissions to manage system contracts. It is able to mint tokens, set fees, and update network-level contracts.

Tokens & Fees

The Service Account has administrator access to the FLOW token smart contract, so it has authorization to mint and burn tokens. It also has access to the transaction fee smart contract and can adjust the fees charged for transactions execution on Flow.

Network Management

The Service Account administers other smart contracts that manage various aspects of the Flow network, such as epochs and (in the future) validator staking auctions.

Governance

Besides its special permissions, the Service Account is an account like any other in Flow.

The service account is currently controlled by a smart contract governed by the Flow community.

No single entity has the ability to unilaterally execute a transaction from the service account because it requires four signatures from controlling keys.

The Flow foundation only controls 3 of the keys and the others are controlled by trusted community members and organizations.

Accounts Retrieval

You can use the Flow CLI to get account data by running:

```
sh
flow accounts get 0xf919ee77447b7497 -n mainnet
```

Find more about the command in the CLI docs.

Accounts can be obtained from the access node APIs, currently, there are two gRPC and REST APIs. You can find more information about them here:

gRPC API [building-on-flow/nodes/access-api#accounts](#)

REST API [http-api#tag/Accounts](#)

There are multiple SDKs implementing the above APIs for different languages:

Javascript SDK [tools/clients/fcl-js](#)

Go SDK [tools/clients/flow-go-sdk](#)

Find a list of all SDKs here: tools/clients

```
# blocks.md:
```

```
---
```

```
sidebarposition: 1
```

```
---
```

Blocks

Overview

Blocks are entities that make up the Flow blockchain. Each block contains a list of transactions that were executed and as a result, changed the global blockchain state. Each block is identified by a unique ID which is a cryptographic hash of the block contents. Block also includes a link to the parent block ID creating a linked list of blocks called the Flow blockchain.

The unique block ID serves as proof of the block contents which can be independently validated by any observer. Interesting cryptographic properties of the hash that make up the block ID guarantee that if any change is made to the block data it would produce a different hash and because blocks are linked, a different hash would break the link as it would no longer be referenced in the next block.

A very basic representation of blocks is:

!Screenshot 2023-08-16 at 15.16.38.png

Blocks are ordered starting from the genesis block 0 up to the latest block. Each block contains an ordered list of transactions. This is how the Flow blockchain preserves the complete history of all the changes made to the state from the beginning to the current state.

Each block contains more data which is divided into block header and block payload. There are many representations of block data within the Flow protocol. APIs, node types, and specific components within the node may view a block from differing perspectives. For the purpose of this documentation, we will talk about block data we expose through APIs to the clients.

!Screenshot 2023-08-16 at 10.50.53.png

Block Header

The Block header contains the following fields:

- ID represents the block's unique identifier, which is derived from the hashing block header including the payload hash. The algorithm used on Flow to hash the content and get an identifier is SHA3 256. This ID is a commitment to all the values in the block staying the same.

- Parent ID is a link to the previous block ID in the list making up the blockchain.
- Height is the block sequence number, where block 0 was the first block produced, and each next block increments the value by 1.
- Timestamp is the timestamp at which this block was proposed by the consensus node. Depending on your use case this time might not be accurate enough, read more about measuring time on the Flow blockchain.
- Payload Hash represents the payload hash that is included when producing the ID of the block. Payload hash is calculated by taking Merkle root hashes of collection guarantees, seals, execution receipts, and execution results and hashing them together. More on each of the values in the block payload section.

Block Payload

The block payload contains the following fields:

- Collection Guarantees is a list of collection IDs with the signatures from the collection nodes that produced the collections. This acts as a guarantee by collection nodes that transaction data in the collection will be available on the collection node if requested by other nodes at a later time. Flow purposely skips including transaction data in a block, making blocks as small as possible, and the production of new blocks by consensus nodes fast, that is because consensus nodes have to sync the proposed block between nodes, and that data should be the smallest possible. The consensus nodes don't really care what will a transaction do as long as it's valid, they only need to define an order of those transactions in a block.
- Block Seals is the attestation by verification nodes that the transactions in a previously executed block have been verified. This seals a previous block referenced by the block ID. It also references the result ID and execution root hash. It contains signatures of the verification nodes that produced the seal.

Lifecycle and Status

Block status is not a value stored inside the block itself but it represents the lifecycle of a block. We derive this value based on the block inclusion in the Flow blockchain and present it to the user as it acts as an important indicator of the finality of the changes the block contains.

Here we'll give an overview of the different phases a block goes through. More details can be found in the whitepaper. Also, a lot of the block states are not necessarily important to the developer but only important to the functioning of the Flow blockchain.

New blocks are constantly being proposed even if no new transactions are submitted to the network. Consensus nodes are in charge of producing blocks. They use a consensus algorithm (an implementation of HotStuff) to agree on what the new block will be. A block contains the ordered list of collections and each collection contains an ordered list of transactions. This is an important fact to reiterate. A block serves as a list of

transitions to the Flow state machine. It documents, as an ordered list, all the changes transactions will make to the state.

A block that is agreed upon by the consensus nodes using an implementation of HotStuff consensus algorithm to be the next block is finalized. This means the block won't change anymore and it will next be executed by the execution node. Please be careful because until a block is sealed the changes are not to be trusted. After verification nodes validate and agree on the correctness of execution results, a block is sealed and consensus nodes will include these seals in the new block.

In summary, a block can be either finalized which guarantees transactions included in the block will stay the same and will be executed, and sealed which means the block execution was verified.

!Screenshot 2023-08-16 at 10.48.26.png

Block Retrieval

You can use the Flow CLI to get the block data by running:

```
sh
flow blocks get latest -network mainnet
```

Find more about the command in the CLI docs.

Blocks can be obtained from the access node APIs, currently, there are two gRPC and REST APIs. You can find more information about them here:

[gRPC Block API](#)

[REST Block API](#)

There are multiple SDKs implementing the above APIs for different languages:

[Javascript SDK](#)

[Go SDK](#)

Find a list of all SDKs [here](#)

```
# collections.md:
```

```
---
sidebarposition: 1
---
```

Collections

Collections link blocks and transactions together. Collection node clusters make these collections (using the HotStuff consensus algorithm),

made up of an ordered list of one or more hashes of signed transactions. In order to optimize data, blocks don't contain transactions (as they do on Ethereum). The benefits are transaction data does not get transferred to consensus nodes on the network which optimizes transfer speed and this architecture allows scaling of ingestion speed by adding collection clusters. Consensus nodes need to only agree on the order of transactions to be executed, they don't need to know the transaction payload, thus making blocks and collections lightweight. Collection nodes hold transaction payloads for anyone who requests them (e.g. execution nodes).

!Screenshot 2023-08-17 at 19.50.39.png

Collection Retrieval

You can use the Flow CLI to get the collection data by running:

```
sh
flow collections get
caff1a7f4a85534e69badcda59b73428a6824ef8103f09cb9eaeaa216c7d7d3f -n
mainnet
```

Find more about the command in the CLI docs.

Collections can be obtained from the access node APIs, currently, there are two gRPC and REST APIs. You can find more information about them here:

[gRPC Collection API](#)

[REST Collection API](#)

There are multiple SDKs implementing the above APIs for different languages:

[Javascript SDK](#)

[Go SDK](#)

Find a list of all SDKs [here](#)

```
# events.md:
```

```
---
sidebarposition: 6
---
```

Events

Flow events are special values that are emitted on the network during the execution of a Cadence program and can be observed by off-chain observers.

Events are defined as Cadence code and you should read Cadence documentation to understand how to define them.

Since transactions don't have return values you can leverage events to broadcast certain changes the transaction caused. Clients listening on Flow networks (apps) can listen to these events being emitted and react.

!Screenshot 2023-08-18 at 14.09.33.png

There are two types of events emitted on the Flow network:

- Core events
- User-defined events

Events consist of the event name and an optional payload.

!Screenshot 2023-08-18 at 13.59.01.png

Core Events

Core events are events emitted directly from the FVM (Flow Virtual Machine). The events have the same name on all networks and do not follow the same naming as user-defined events (they have no address).

A list of events that are emitted by the Flow network is:

Event Name	Description
flow.AccountCreated	Event that is emitted when a new account gets created.
flow.AccountKeyAdded	Event that is emitted when a key gets added to an account.
flow.AccountKeyRemoved	Event that is emitted when a key gets removed from an account.
flow.AccountContractAdded	Event that is emitted when a contract gets deployed to an account.
flow.AccountContractUpdated	Event that is emitted when a contract gets updated on an account.
flow.AccountContractRemoved	Event that is emitted when a contract gets removed from an account.
flow.InboxValuePublished	Event that is emitted when a Capability is published from an account.
flow.InboxValueUnpublished	Event that is emitted when a Capability is unpublished from an account.
flow.InboxValueClaimed1	Event that is emitted when a Capability is claimed by an account.

For more details on the core events, you can read Cadence reference documentation.

User-defined events

Events that are defined inside contracts and when emitted follow a common naming schema. The schema consists of 4 parts:

```
cadence
A.{contract address}.{contract name}.{event type}
```

An example event would look like:

!Screenshot 2023-08-18 at 14.30.36.png

The first A means the event is originating from a contract, which will always be the case for user-defined events. The contract address as the name implies is the location of a contract deployed on the Flow network. Next, is the name of the contracted event originates from, and last is the event type defined in the contract.

There is an unlimited amount of events that can be defined on Flow, but you should know about the most common ones.

Fungible Token Events

All fungible token contracts, including The FLOW Token contract, use the fungible token standard on Flow. As with any contract, the standard emits events when interacted with. When any fungible token is transferred, standard events are emitted. You can find a lot of details on the events emitted in the Fungible Token documentation.

The most common events are when tokens are transferred which is accomplished with two actions: withdrawing tokens from the payer and depositing tokens in the receiver. Each of those actions has a corresponding event:

Withdraw Tokens

```
Event name: FungibleToken.Withdrawn
cadence
event Withdrawn(type: String,
                 amount: UFix64,
                 from: Address?,
                 fromUUID: UInt64,
                 withdrawnUUID: UInt64,
                 balanceAfter: UFix64)
```

Mainnet event: A.f233dcee88fe0abe.FungibleToken.Withdrawn

Testnet event: A.9a0766d93b6608b7.FungibleToken.Withdrawn

Deposit Tokens

```
cadence
event Deposited(type: String,
```

```

        amount: UFix64,
        to: Address?,
        toUUID: UInt64,
        depositedUUID: UInt64,
        balanceAfter: UFix64)

```

Event name: FungibleToken.Deposited

Mainnet event: A.f233dcee88fe0abe.FungibleToken.Deposited

Testnet event: A.9a0766d93b6608b7.FungibleToken.Deposited

Fee Events

Since fees are governed by a contract deployed on the Flow network, that contract also emits events when fees are deducted.

Charging fees consists of a couple of steps:

- Calculate and deduct fees
- Withdraw Flow tokens from the payer account
- Deposit Flow tokens to the fees contract

These events are very common since they accommodate all transactions on Flow. Each fee deduction will result in three events: the withdrawal of Flow tokens, the deposit of Flow tokens, and the fee deduction.

An example of fee events:

```

yml
Events:
  - Index: 0
    Type: A.f233dcee88fe0abe.FungibleToken.Withdrawn
    Tx ID:
    1ec90051e3bc74fc36cbd16fc83df08e463dda8f92e8e2193e061f9d41b2ad92
    Values:
      - type (String): "1654653399040a61.FlowToken.Vault"
      - amount (UFix64): 0.00000100
      - from (Address?): b30eb2755dca4572

  - Index: 1
    Type: A.f233dcee88fe0abe.FungibleToken.Deposited
    Tx ID:
    1ec90051e3bc74fc36cbd16fc83df08e463dda8f92e8e2193e061f9d41b2ad92
    Values:
      - type (String): "1654653399040a61.FlowToken.Vault"
      - amount (UFix64): 0.00000100
      - to (Address?): f919ee77447b7497

  - Index: 2
    Type: A.f919ee77447b7497.FlowFees.FeesDeducted
    Tx ID:
    1ec90051e3bc74fc36cbd16fc83df08e463dda8f92e8e2193e061f9d41b2ad92

```

Values:

- amount (UFix64): 0.00000100
- inclusionEffort (UFix64): 1.00000000
- executionEffort (UFix64): 0.00000000

fees.md:

```
---
```

sidebarposition: 5
title: Fees

:::info

Are you an EVM developer looking for information about EVM Accounts on Flow? If so, check out the EVM specific documentation here

:::

Fees

Transaction Fees

A transaction fee is a cost paid in Flow by the payer account and is required for a transaction to be included in the Flow blockchain. Fees are necessary for protecting the network against spam/infinite running transactions and to provide monetary incentives for participants that make up the Flow network.

A transaction fee is paid regardless of whether a transaction succeeds or fails. If the payer account doesn't have sufficient Flow balance to pay for the transaction fee, the transaction will fail. We can limit the transaction fee to some extent by providing the gas limit value when submitting the transaction.

Understanding the need for transaction fees

Segmented transaction fees are essential to ensure fair pricing based on the impact on the network. For instance, more heavy operations will require more resources to process and propagate transactions. Common operations, however, will stay reasonably priced.

Fees will improve the overall security of the network by making malicious actions (eg spam) on the network less viable.

The unique Flow architecture is targeted at high throughput. It makes it easier to have slack in the system, so short spikes can be handled more gracefully.

Fee Structure

Each transaction fee consists of three components: execution fee, inclusion fee, and network surge factor.

!Screenshot 2023-08-17 at 17.16.32.png

Execution Fee

The execution effort for a transaction is determined by the code path the transaction takes and the actions it does. The actions that have an associated execution effort cost can be separated into four broad buckets:

- Normal lines of cadence, loops, or function calls
- Reading data from storage, charged per byte read
- Writing data to storage, charged per byte written
- Account creation

Transaction Type (FLOW)	Estimated cost
FT transfer	0.00000185
Mint a small NFT (heavily depends on the NFT size)	0.0000019
Empty Transaction	0.000001
Add key to an account	0.000001
Create 1 Account	0.00000315
Create 10 accounts	0.00002261
Deploying a contract that is 50kb	0.00002965

Inclusion Fee

The inclusion effort of a transaction represents the work needed for:

- Including the transaction in a block
- Transporting the transaction information from node to node
- Verifying transaction signatures

Right now, the inclusion effort is always 1.0 and the inclusion effort cost is fixed to 0.000001.

Surge Factor

In the future, a network surge will be applied when the network is busy due to an increased influx of transactions required to be processed or a decrease in the ability to process transactions. Right now, the network surge is fixed to 1.0.

Currently, both the inclusion fee and surge factor don't represent any significant Flow fees. Keep in mind this can change in the future.

Estimating transaction costs

Cost estimation is a two-step process. First, you need to gather the execution effort with either the emulator, on testnet, or on mainnet. Second, you use the execution effort for a transaction to calculate the final fees using one of the JavaScript or Go FCL SDKs.

Storage

Flow's approach to storage capacity is a bit similar to some banks' pricing models, where maintaining a minimum balance prevents monthly account fees. Here, the amount of data in your account determines your minimum balance. If you fall below the minimum balance, you cannot transact with your account, except for deposits or deleting data. The essence of storage fee model is that it ensures data availability without continuously charging fees for storage, while also preventing abuses that could burden the network's storage resources. This distinction between current state and blockchain history is crucial for understanding storage requirements and limitations.

Each Flow account has associated storage used. The account's storage used is the byte size of all the data stored in the account's storage. Accounts also have a storage capacity, which is directly tied to the amount of Flow tokens an account has. The account can, without any additional cost, use any amount of storage up to its storage capacity.

:::warning

If a transaction puts an account over storage capacity, that transaction fails and is reverted. Likewise, if a transaction would drop an account's balance below 0.001 Flow tokens, which is the minimum an account can have, the transaction would also fail.

:::

Storage Capacity

The storage capacity of an account is dictated by the amount of FLOW it has.

:::danger

The minimum amount of FLOW an account can have is 0.001. This minimum is provided by the account creator at account creation.

:::

The minimum account reservation ensures that most accounts won't run out of storage capacity if anyone deposits anything (like an NFT) to the account.

Currently, the amount required to store 100 MB in account storage is 1 Flow.

!Screenshot 2023-08-17 at 17.27.50.png

Please note that storing data in an account on Flow doesn't charge tokens from the account, it just makes sure you will keep the tokens as a reserve. Once the storage is freed up you can transfer the Flow tokens.

Storage Capacity of the Payer

The storage capacity of the Payer of a transaction is generally computed the same way as the capacity of any other account, however, the system needs to account for the transaction fees the payer will incur at the end of the transaction. The final transaction fee amount is not fully known at this step, only when accounts are checked for storage compliance. If their storage used is more than their storage capacity, the transaction will fail.

Because of this, the payer's balance is conservatively considered to be lower by the maximum possible transaction fees, when checking for storage compliance. The maximum transaction fee of a specific transaction is the transaction fee as if the transaction would have used up all of its execution effort limit.

Storage Used

All data that is in an account's storage counts towards storage used. Even when an account is newly created it is not empty. There are already some items in its storage:

- Metadata that marks that the account exists.
- An empty FLOW vault, and stored receiver capability.
- Public keys to the account if the account was created with keys.
- Smart contracts deployed on the account if the account was created with contracts.
- The value of the account's storage used as an unsigned integer.

Adding additional keys, smart contracts, capabilities, resources, etc. to the account counts towards storage used.

Data stored on the Flow blockchain is stored in a key-value ledger. Each item's key contains the address that owns the item and the path to the item. An account can have many keys, therefore flow considers the account key items are stored with. This means that the storage used by each item is the byte length of the item plus the byte length of the item's key.

Maximum available balance

Due to the storage restrictions, there is a maximum available balance that user can withdraw from the wallet. The core contract FlowStorageFees provides a function to retrieve that value:

cadence

```
import "FlowStorageFees"

access(all) fun main(accountAddress: Address): UFix64 {
    return FlowStorageFees.defaultTokenAvailableBalance(accountAddress)
}
```

Alternatively developers can use availableBalance property of the Account

```
cadence
access(all) fun main(address: Address): UFix64 {
    let acc = getAccount(address)
    let balance = acc.availableBalance

    return balance
}
```

Practical Understanding of Fees

Using Flow Emulator

You can start the emulator using the Flow CLI. Run your transaction and take a look at the events emitted:

```
shell
0|emulator | time="2022-04-06T17:13:22-07:00" level=info msg="★
Transaction executed" computationUsed=3
txID=a782c2210c0c1f2a6637b20604d37353346bd5389005e4bff6ec7bcf507fac06
```

You should see the computationUsed field. Take a note of the value, you will use it in the next step.

On testnet or mainnet

Once a transaction is completed, you can use an explorer like Flowdiver to review the transaction details and events emitted. For Flowdiver, you can open the transaction in question and look for the event FeesDeducted from the FlowFees contract:

```
!flowscan-fees
```

In the event data on the right side, you will see a set of fields representing the fees for a specific transaction.:

- Total Fees Paid
- Inclusion Effort
- Execution Effort

Take a note of the last value in the list - the executionEffort value. You will use it in the next step.

Calculating final costs

The cost for transactions can be calculated using the following FCL scripts on mainnet/testnet respectively.

On mainnet

```
cadence
import FlowFees from 0xf919ee77447b7497
access(all) fun main(
    inclusionEffort: UFix64,
    executionEffort: UFix64
): UFix64 {
    return FlowFees.computeFees(inclusionEffort: inclusionEffort,
executionEffort: executionEffort)
}
```

On testnet

```
cadence
import FlowFees from 0x912d5440f7e3769e
access(all) fun main(
    inclusionEffort: UFix64,
    executionEffort: UFix64
): UFix64 {
    return FlowFees.computeFees(inclusionEffort: inclusionEffort,
executionEffort: executionEffort)
}
```

Configuring execution limits

FCL SDKs allow you to set the execution effort limit for each transaction. Based on the execution effort limit determined in the previous step, you should set a reasonable maximum to avoid unexpected behavior and protect your users. The final transaction fee is computed from the actual execution effort used up to this maximum.

> Note: Keep in mind that the limits are not for the final fees that the user will have to pay. The limits are for the execution efforts specifically.

It is important to set a limit that isn't too high or too low. If it is set too high, the payer needs to have more funds in their account before sending the transaction. If it is too low, the execution could fail and all state changes are dropped.

Using FCL JS SDK

You need to set the limit parameter for the mutate function, for example:

```
js
import as fcl from "@onflow/fcl"
```

```

const transactionId = await fcl.mutate({
  cadence:
    transaction {
      execute {
        log("Hello from execute")
      }
    }
  ,
  proposer: fcl.currentUser,
  payer: fcl.currentUser,
  limit: 100
})

const transaction = await fcl.tx(transactionId).onceSealed();
console.log(transaction);

```

Using FCL Go SDK

You need to call the `SetComputeLimit` method to set the fee limit, for example:

```

go
import (
  "github.com/onflow/flow-go-sdk"
  "github.com/onflow/flow-go-sdk/crypto"
)

var (
  myAddress     flow.Address
  myAccountKey flow.AccountKey
  myPrivateKey crypto.PrivateKey
)

tx := flow.NewTransaction().
  SetScript([]byte("transaction { execute { log(\"Hello, World!\") } }")).
  SetComputeLimit(100).
  SetProposalKey(myAddress, myAccountKey.Index,
myAccountKey.SequenceNumber).
  SetPayer(myAddress)

```

Maximum transaction fees of a transaction

The maximum possible fee imposed on the payer for a transaction can be calculated as the inclusion cost plus the execution cost. The execution cost is the fee calculated for running the transaction based on the execution effort limit maximum specified.

The payer will never pay more than this amount for the transaction.

Optimizing Cadence code to reduce effort

Several optimizations can lead to reduced execution time of transactions. Below is a list of some practices. This list is not exhaustive but rather exemplary.

Limit functions calls

Whenever you make function calls, make sure these are absolutely required. In some cases, you might be able to check prerequisites and avoid additional calls:

```
cadence
for obj in sampleList {
    /// check if call is required
    if obj.id != nil {
        functionCall(obj)
    }
}
```

Limit loops and iterations

Whenever you want to iterate over a list, make sure it is necessary to iterate through all elements as opposed to a subset. Avoid loops to grow in size too much over time. Limit loops when possible.

```
cadence
// Iterating over long lists can be costly
access(all) fun sum(list: [Int]): Int {
    var total = 0
    var i = 0
    // if list grows too large, this might not be possible anymore
    while i < list.length {
        total = total + list[i]
    }
    return total
}

// Consider designing transactions (and scripts) in a way where work can
// be "chunked" into smaller pieces
access(all) fun partialSum(list: [Int], start: Int, end: Int): Int {
    var partialTotal = 0
    var i = start
    while i < end {
        partialTotal = partialTotal + list[i]
    }
    return partialTotal
}
```

Understand the impact of function calls

Some functions will require more execution efforts than others. You should carefully review what function calls are made and what execution they involve.

```
cadence
// be aware functions that call a lot of other functions
// (or call themselves) might cost a lot
access(all) fun fib( x: Int): Int {
    if x == 1 || x== 0 {
        return x
    }
    // + 2 function calls each recursion
    return fib(x-1) + fib(x-2)
}

// consider inlining functions with single statements, to reduce costs
access(all) fun add( a: Int, b: Int): Int {
    // single statement; worth inlining
    return a + b
}
```

Avoid excessive load and save operations

Avoid costly loading and storage operations and borrow references where possible, for example:

```
cadence
transaction {

    prepare(acct: auth(BorrowValue) &Account) {

        // Borrows a reference to the stored vault, much less costly
        operation that removing the vault from storage
        let vault <- acct.storage.borrow<&ExampleToken.Vault>(from:
        /storage/exampleToken)

        let burnVault <- vault.withdraw(amount: 10)

        destroy burnVault

        // No save required because we only used a reference
    }
}
```

> Note: If the requested resource does not exist, no reading costs are charged.

Limit accounts created per transaction

Creating accounts and adding keys are associated with costs. Try to only create accounts and keys when necessary.

Check user's balance before executing transactions

You should ensure that the user's balance has enough balance to cover the highest possible fees. For FT transfers, you need to cover the amount to transfer in addition to the highest possible fees.

Educating users

Wallets will handle the presentation of the final transaction costs but you can still facilitate the user experience by educating them within your application.

If your user is using self-custody wallets, they may have to pay the transaction and want to understand the fees. Here are some suggestions.

Explain that costs can vary depending on the network usage

Suggested message: "Fees improve the security of the network. They are flexible to ensure fair pricing based on the impact on the network."

Explain that waiting for the network surge to pass is an option

Inevitably, network surges will cause higher fees. Users who might want to submit a transaction while the network usage is surging should consider sending the transaction at a later time to reduce costs.

Explain that the wallet might not allow the transaction due to a lack of funds

If dynamic fees increase to the highest possible level, the user's fund might not be enough to execute the transaction. Let the users know that they should either add funds or try when the network is less busy.

How to learn more

There are several places to learn more about transaction fees:

- FLIP-660
- FLIP-753
- Flow Fees Contract

> Note: If you have thoughts on the implementation of transaction fees on Flow, you can leave feedback on this forum post.

FAQs

When will the fee update go into effect?

The updates were rolled out with the Spork on April 6, 2022, and were enabled on June 1st during the weekly epoch transition.

Why are fees collected even when transactions fail?

Broadcasting and verifying a transaction requires execution, so costs are deducted appropriately.

What execution costs are considered above average?

There is no average for execution costs. Every function will vary significantly based on the logic implemented. You should review the optimization best practices to determine if you could reduce your costs.

Do hardware wallets like Ledger support segmented fees?

Yes.

What is the lowest execution cost?

The lowest execution cost is 1. This means your transaction included one function call or loop that didn't read or write any data.

Can I determine how much a transaction will cost on mainnet without actually paying?

You can estimate the costs in a two-way process: 1) determine execution costs for transactions (emulator or testnet) and 2) use an FCL SDK method to calculate the final transaction fees.

How accurate will testnet fees be to mainnet fees?

Final fees are determined by the surge factor on the network. The surge factor for the testnet will be different from the factor for the mainnet, so you need to expect a variation between mainnet and testnet estimates.

I use Blocto and I haven't paid any fees yet. Why is that?

That is because Blocto is acting as the payer for transactions. Self-custody wallets may have the user pay the transaction. Additionally, apps can sponsor the transaction if they choose.

Why would the same transaction have different fees when executed for different accounts?

Execution costs, among other things, include the cost to read data from account storage and since the data stored varies from account to account, so does the execution costs, and subsequently the transaction fees.

Additional Details:

- The most expensive operations in Cadence are reading and writing to storage. This isn't punitive! Every read needs to be sent to all Verification nodes for verification (with Merkle proofs), and every write requires a path of Merkle hashes to be updated. Reading and writing to storage is inherently expensive on any blockchain.
- The way data is stored in accounts is as a tree (the hint is in the name "atree" :wink:). So, the more elements in the account, the more levels of the tree, and therefore the more nodes of that tree that need

to be read and updated. So, looking at the byte size of an account is a decent proxy for figuring out how much it's going to cost.

- Because it's a tree, the cost of reads and writes grows with $\log(n)$, but does scale.
- atree has an update queued up for Crescendo that will improve this. The previous version erred on the side of adding new levels to the tree (to keep the code simple), while the new version tries to pack more data at each level. This should result in fewer levels for the same byte size. Additionally, it includes a more compact encoding leading to a reduction in the byte size of most accounts.
- Even with these improvements, this relationship is likely to remain indefinitely. The bigger the account, the more bookkeeping the nodes have to do, which will result in somewhat larger tx fees.

```
# flow-token.md:
```

```
---
```

title: FLOW Coin
sidebarposition: 10

Introduction

This section contains information about the FLOW Coin for individual backers, wallet providers, custodians and node operators.

FLOW as a Native Coin

FLOW is the default coin for the Flow protocol, meaning it is used for all protocol-level fee payments, rewards and staking transactions. FLOW implements the standard Flow Fungible Token interface, which all other on-chain fungible tokens also conform to. This interface is defined in Cadence, Flow's native smart-contract programming language, which makes it easy to write applications that interact with FLOW.

How to Get FLOW

There are two ways to acquire FLOW Coins as yield:

1. Earn FLOW as a Validator or Delegator: Receive newly-minted FLOW as a reward for running a node.
1. Earn FLOW as a Community Contributor: Flow offers grants for selected proposals as well as RFPs for teams to submit proposals for funded development

How to Use FLOW

With FLOW, you can:

- Spend

- Stake
- Delegate
- Hold
- Vote
- Send and share
- Create, develop, and grow your dapp

Spending FLOW

All you need to spend FLOW is an account and a tool for signing transactions (a wallet, custodian, or other signing service). The FCL (Flow Client Library) makes it super duper easy to go to any dapp, login with your account, have a great time, and then sign with the wallet of your choice only once you decide to make a purchase.

Staking FLOW

You can use FLOW to operate a staked node. Node operators receive newly-minted FLOW as a reward for helping to secure the network.

Delegating FLOW

You can use FLOW for stake delegation. Delegators receive newly-minted FLOW as a reward for helping to secure the network.

Holding FLOW

If you have already purchased FLOW and wish to hold it, you have a couple of options:

- For relatively small, short term holdings - most people use a wallet. Wallets are used to help you sign transactions (verify your actions) when using your FLOW tokens.
- For larger, long term holdings - you may want to use a custody provider to keep your funds safe.

You can find wallets and custodians supporting Flow in the Flow Port

Voting with FLOW

Participating in the Flow community is more than just running a node or building a dapp. It's also about engaging in discussion, debate, and decision making about the protocol, the content on it, and the people impacted by it. You can use your Flow account to submit votes to community polls and other governance related activities.

Sending and Sharing FLOW

If you simply want to share the love and bring your friends to Flow, it's easier than an edible arrangement.

It is possible to use the Flow blockchain without holding any FLOW coins yourself.

Free to play games, trials, community polls, and other community activities can all take place with only an account (which may be created on a person's behalf) and a small fixed fee which may be paid by a user agent.

The protocol requires some FLOW coins to process these transactions, but (and this is the cool part!) a product can support users who do not themselves hold FLOW while still providing that user with all the underlying security guarantees the Flow protocol provides.

Transferring FLOW, creating accounts, and updating keys are all actions made easy on Flow Port

Submitting Transactions and Updating Users

Transactions are submitted using a Flow SDK via the Access API.

On Flow, a transaction is identified by its hash - the hash that exists as soon as that transaction is signed and submitted to an Access or Collection node.

Results of transactions can be queried by transaction hash through the Access API.

A user can check the status of a transaction at any time via the Flow Block Explorer.

To expose these results natively in your app, you can use a Flow SDK to fetch transaction results, for example using the Flow Go SDK.

Using a Flow SDK you can also fetch account state by address from a Flow Access API, for example using the Flow Go SDK.

Once the transaction is sealed, an event is emitted and you will be able to read transaction events and update the user.

The Flow SDKs also allow polling for events using the Flow Access API, for example using the Flow Go SDK.

How to Build with FLOW

To get started building on Flow, please see the Flow App Quickstart

```
# scripts.md:
```

```
---  
sidebarposition: 4  
---
```

Scripts

A script provides a light-weight method to query chain data.

It is executable Cadence code that can query for Flow execution state data but cannot modify it in any way.

Unlike a Flow transaction, a script is not signed and requires no transaction fees. Also unlike a transaction, a script can return a value back to the caller.

You can think of executing a script as a read-only operation, very similar to the ethcall RPC method on Ethereum.

Scripts are currently executed on either the Access Nodes or the Execution Nodes based on the Access node configuration.

Scripts are defined by the following the Cadence code:

```
cadence  
// The 'main' function is the entry point function and every script needs  
to have one.  
access(all) fun main() {  
    // Cadence statements to be executed go here  
}
```

Scripts can return a typed value:

```
cadence  
access(all) fun main(): Int {  
    return 1 + 2  
}
```

Scripts can also accept arguments:

```
cadence  
access(all) fun main(arg: String): String {  
    return "Hello ".concat(arg)  
}
```

Scripts can call contract functions and query the state of a contract. To call a function on another contract, import it from its address and invoke the function:

```
cadence  
import World from 0x01  
  
access(all) fun main(): String {
```

```
        return World.hello()  
    }
```

Scripts can also be run against previous blocks, allowing you to query historic data from the Flow network. This is particularly useful for retrieving historical states of contracts or tracking changes over time.

When to use a script?

Scripts can be used for the following:

1. Validating a transaction before submitting it e.g. checking if the payer has sufficient balance, the receiver account is setup correctly to receive a token or NFT etc.
2. Collecting chain data over time.
3. Continuously verifying accounts through a background job e.g. a Discord bot that verifies users by their Flow account.
4. Querying core contracts e.g. see staking scripts and events for querying staking and epoch related information, see the scripts directory under each of the core contract transactions for other core contracts related scripts.

Executing Scripts

Access API

A script can be executed by submitting it to the Access API provided by access nodes. Currently, there are three API endpoints that allow a user to execute scripts at the latest sealed block, a previous block height, or a previous block ID.

gRPC Script API

REST Script API

There are multiple SDKs implementing the above APIs for different languages:

Javascript SDK

Go SDK

Find a list of all SDKs [here](#)

Flow CLI

You can also execute a script by using the Flow CLI:

```
sh  
flow scripts execute ./helloWorld.cdc
```

A user can define their own scripts or can use already defined scripts by the contract authors that can be found by using the FLIX service.

Best Practices

Following are some recommendations on how to write efficient scripts:

1. Simpler and shorter scripts: Scripts, like transactions, are subject to computation limits (see limitations). It is recommended to run shorter and simpler scripts which have low time complexity for a faster response. If you have a script with several nested loops, long iteration, or that queries many onchain fields, consider simplifying the script logic.
2. Fewer state reads: A script reads execution state and to get a faster response, it is best to limit the amount of state that is read by the script.
3. Smaller length of array or dictionary type arguments: If your script requires an array or a dictionary as an argument where each element causes a state lookup, instead of making a single script call passing in a long list, make multiple calls with a smaller subset of the array or dictionary.
4. NFTCatalog: If your script uses the NFTCatalog functions, ensure that you use the latest functions and do not use any of the deprecated functions such as `getCatalog()`.

Limitations

1. Rate limit - Script execution is subjected to API rate-limits imposed by the Access nodes and the Execution nodes. The rate limits for the Public Access nodes hosted by QuickNode are outlined here.
2. Computation limit - Similar to a transaction, each script is also subjected to a computation limit. The specific value can be configured by individual Access and Execution node operators. Currently, the default compute (gas) limit for a script is 100,000.
3. Historic block data limit
 1. Script execution on execution nodes is restricted to approximately the last 100 blocks. Any request for script execution on an execution node on a past block (specified by block ID or block height) will fail if that block is more than 100 blocks in the past.
 2. Script execution on an access node can go much beyond the last 100 blocks but is restricted to the height when the last network upgrade (HCU or spork) occurred.

```
# smart-contracts.md:

---
slug: /build/basics/smart-contracts
redirect: /build/smart-contracts/overview
title: Smart Contracts ✓
---

Smart Contracts

Go to Smart Contracts

import {Redirect} from '@docusaurus/router';

<Redirect to="/build/smart-contracts/overview" />

# transactions.md:

---
sidebarposition: 3
keywords:
- Flow Transaction Speed
- Flow Transaction Time
- Transactions on Flow
---

Transactions

Transactions are cryptographically signed data messages that contain a set of instructions that update the Flow state. They are a basic unit of computation that gets executed by execution nodes. In order for a transaction to be included in the Flow blockchain a fee is required from the payer.

!Screenshot 2023-08-17 at 13.57.36.png

:::tip

Transactions on Flow are fundamentally different from those on Ethereum. The main purpose of a transaction is not to send funds but to contain code that gets executed. This makes transactions very flexible and powerful. In addition to being able to access the authorizing accounts private assets, transactions can also read and call functions in public contracts, and access public domains in other users' accounts Transactions on Flow also feature different roles, such as defining third-party payer accounts, proposer accounts, and authorizers, which we will talk about in detail soon.

:::
```

In order for a transaction to be valid and executed it must contain signatures from accounts involved as well as some other information, let's take a look at all the required fields.

!Screenshot 2023-08-17 at 14.52.56.png

Script

The script section contains instructions for transaction execution. This is a Cadence program in source code form (human-readable), and encoded as UTF-8. The transaction program must contain a transaction declaration.

A transaction includes multiple optional phases prepare, pre, execute, and post phase. You can read more about it in the Cadence reference document on transactions. Each phase has a purpose, the two most important phases are prepare and execute.

In the prepare phase, we have access to &Account objects, which gives us the power to interact with those accounts. The accounts are called authorizers of transactions, so each account we want to interact with in the prepare phase must sign the transaction as an authorizer. The execute phase does exactly what it says, it executes the main logic of the transaction. This phase is optional, but it is a best practice to add your main transaction logic in the section, so it is explicit.

Again make sure to read Cadence documentation on transactions

This is an example of a transaction script:

```
cadence
transaction(greeting: String) {
    execute {
        log(greeting.concat(", World!"))
    }
}
```

Arguments

Transactions may declare parameters it needs during execution, these must be provided as input arguments when sending a transaction. You can think of them as function arguments. Currently, we provide arguments in the JSON-Cadence Data Interchange Format. Which is a human-readable JSON format. The sample script from above accepts a single String argument.

Reference Block

A reference to a recent block used for expiry. A transaction is considered expired if it is submitted to Flow after reference block height + N, where N is a constant defined by the network. On mainnet current setting for N is 600 which amounts to approximately 10 minutes for expiry (please note this is subject to change).

Gas Limit

When a transaction is executed each operation consumes a predefined amount of computational units (we define more about that in the Fees documentation). This defines the maximum amount of computation that is allowed to be done during this transaction. If a transaction completes execution using fewer computational units than the limit, it remains unaffected. However, if it hits this limit during execution, the transaction will fail, its changes will be reverted, but fees will still be applied. The maximum computational limit for Flow mainnet is currently at 9999, but this might change. The maximum network limit is defined to protect the network from transactions that would run forever.

Proposal Key

Each transaction must declare a proposal key, which can be an account key from any Flow account (App, User or Wallet). The account that owns the proposal key is referred to as the proposer.

Proposer is a role in a transaction that defines who is proposing the transaction, the effect of the transaction being submitted on the proposer is that it will increment the sequence number for the provided proposer key. This is done to ensure transactions are not resubmitted (replay attack) and thus sequencing actions.

A proposal key definition declares the address, key ID, and up-to-date sequence number for the account key. A single proposer can have many transactions executed in parallel only limited by the key they use to propose the transaction.

!Screenshot 2023-08-17 at 15.10.33.png

- Address identifies the account that will act as a proposer of this transaction.
- Key ID is an index number (starting at 0) that identifies the key on the account provided in the address.
- Sequence Number is a number on each key that increments by 1 with each transaction. This ensures that each transaction executes at most once and prevents many unwanted situations, such as transaction replay attacks. Each key in an account has a dedicated sequence number associated with it. Unlike Ethereum, there is no sequence number for the entire account.

Authorizers

Authorizers are accounts that authorize a transaction to read and mutate their state. A transaction can specify zero or more authorizers, depending on how many accounts the transaction needs to access.

The number of authorizers on the transaction must match the number of &Account parameters declared in the prepare statement of the Cadence script.

Example transaction with multiple authorizers:

cadence

```
transaction {
    prepare(authorizer1: auth(Capabilities) &Account, authorizer2:
auth(Storage) &Account) { }
}
```

Each account defined as an authorizer must sign the transaction with its own key, and by doing so it acknowledges the transaction it signed will have access to that account and may modify it. How it will modify it is understood from the list of account entitlements that are granted in the prepare argument list and by reading the transaction script. In an transaction, developers should only give the minimum set of account entitlements that are required for the transaction to execute properly. This ensures that users who are signing transactions can understand what parts of their account a transaction can access.

Payer

A payer is the account that pays the fees for the transaction. A transaction must specify exactly one payer. The payer is only responsible for paying the network and gas fees; the transaction is not authorized to access resources or code stored in the payer account.

By explicitly specifying a payer a transaction can be paid by third-party services such as wallet providers.

Transaction Lifecycle

Once a transaction has been submitted to the Flow network using the Access node APIs, it will begin its lifecycle and eventually reach a finality. Each submitted transaction can be identified with an ID.

Transaction ID

A transaction ID is a hash of the encoded transaction payload and can be calculated at any time. We don't submit transaction ID as part of the transaction payload as it can be derived from the data and thus would mean duplication of data.

Transaction Status

The transaction status represents the state of a transaction on the Flow blockchain. Some statuses are mutable and some are immutable, they usually follow a timeline like so:

!Screenshot 2023-08-17 at 16.08.18.png

- Unknown - The transaction has not yet been seen by the section of the network you communicate with.
- Pending - The transaction has been received by a collection node but has not yet been finalized in a block.

- Finalized - The consensus nodes have included the transaction in a block, but it has not been executed by execution nodes.
- Executed - Execution nodes have produced a result for the transaction.
- Sealed - The verification nodes have verified and agreed on the result of the transaction and the consensus node has included the seal in the latest block.
- Expired - The transaction was submitted past its expiration block height.

:::danger

It is important to differentiate the transaction status and transaction result. Transaction status will only provide you with information about the inclusion of the transaction in the blockchain, not whether the transaction was executed the way you intended. A transaction can still fail to execute the way you intended and be sealed.

:::

Transaction Result

Once a transaction is executed, its result will be available, providing details on its success or any errors encountered during execution. It also includes events the transaction may have emitted.

!Screenshot 2023-08-17 at 16.29.30.png

:::danger

From a developer perspective, a transaction is only successful if:

- It is sealed
- It didn't encounter errors

:::

Transaction Time

Understanding how transaction times work across different blockchains is crucial for developers and users to optimize their operations and expectations. Flow's multi-node architecture allows for some of the fastest transaction times and finality times across chains. Read on for more detail on how it works and what it means for developers and users.

Two Key Transaction Questions

Whenever a transaction is processed, two primary questions come to mind:

1. Inclusion: Will this transaction be included in the final chain?
2. Result: What is the outcome of the transaction?

Different blockchains tackle these questions in varied sequences. For instance, Bitcoin and Ethereum provide answers simultaneously. Layer 2 solutions (L2s) can sometimes address the outcome before confirming

inclusion. But there's a catch: you can have an answer to those questions that might be wrong. Flow, on the other hand, prioritizes the inclusion question.

Transaction Finality

Drawing a parallel to traditional finance, a vendor might instantly know if Visa approves a transaction, but the possibility of chargebacks lingers for weeks. This uncertainty introduces the concept of "finality" in blockchain transactions.

In the dominant Proof-of-Stake (PoS) environment, which includes most chains except for Bitcoin, there are three key finality stages:

- Preliminary result: It's an initial answer to the aforementioned questions. The preliminary result doesn't ensure correctness, and there are no economic penalties (like "slashing") if the informant provides false information.
- Soft economic finality: This stage provides an answer backed by cryptographic proof. If the informant is deceptive, they face economic repercussions or "slashing."
- Hard economic finality: The provided answer either holds true, or the entire blockchain requires a restart. The latter case sees at least one-third of the nodes facing economic penalties.

!finality.png

Chain Comparisons

Chain	Preliminary	Soft finality	Hard finality
Solana	100ms	n/a	30s
Ethereum	15s	n/a	15m
Flow	bypass	6s	20s

Flow

Flow bypasses preliminary results entirely. It reaches soft finality ("Executed") in about 6 seconds and hard finality ("Sealed") in around 20 seconds. If an Access Node on Flow states a transaction has occurred, it's either correct or cryptographic proof exists that can lead to the node's slashing.

!transaction-time.png

Signing a Transaction

Due to the existence of weighted keys and split signing roles, Flow transactions sometimes need to be signed multiple times by one or more parties. That is, multiple unique signatures may be needed to authorize a single transaction.

A transaction can contain two types of signatures: payload signatures and envelope signatures.

!Screenshot 2023-08-17 at 14.52.51.png

Signer Roles

- **Proposer:** the account that specifies a proposal key.
- **Payer:** the account paying for the transaction fees.
- **Authorizers:** zero or more accounts authorizing the transaction to mutate their state.

Payload

The transaction payload is the innermost portion of a transaction and contains the data that uniquely identifies the operations applied by the transaction as we have defined them above. In Flow, two transactions with the same payload will never be executed more than once.

:::warning

⚠ The transaction proposer and authorizer are only required to sign the transaction payload. These signatures are the payload signatures.

:::

Authorization Envelope

The transaction authorization envelope contains both the transaction payload and the payload signatures.

The transaction payer is required to sign the authorization envelope. These signatures are envelope signatures.

:::danger

Special case: if an account is both the payer and either a proposer or authorizer, it is required only to sign the envelope.

:::

Payment Envelope

The outermost portion of the transaction, which contains the payload and envelope signatures, is referred to as the payment envelope.

:::danger

Special case: if an account is both the payer and either a proposer or authorizer, it is required only to sign the envelope.

:::

Payer Signs Last

The payer must sign the portion of the transaction that contains the payload signatures, which means that the payer must always sign last. This ensures the payer that they are signing a valid transaction with all of the required payload signatures.

:::danger

Special case: if an account is both the payer and either a proposer or authorizer, it is required only to sign the envelope.

:::

Signature Structure

A transaction signature is a composite structure containing three fields:

- Address
- Key ID
- Signature Data

The address and key ID fields declare the account key that generated the signature, which is required in order to verify the signature against the correct public key.

Sequence Numbers

Flow uses sequence numbers to ensure that each transaction executes at most once. This prevents many unwanted situations such as transaction replay attacks.

Sequence numbers work similarly to transaction nonces in Ethereum, but with several key differences:

- Each key in an account has a dedicated sequence number associated with it. Unlike Ethereum, there is no sequence number for the entire account.
- When creating a transaction, only the proposer must specify a sequence number. Payers and authorizers are not required to.

:::tip

The transaction proposer is only required to specify a sequence number for a single account key, even if it signs with multiple keys. This key is referred to as the proposal key.

:::

Each time an account key is used as a proposal key, its sequence number is incremented by 1. The sequence number is updated after execution, even if the transaction fails (reverts) during execution.

A transaction is failed if its proposal key does not specify a sequence number equal to the sequence number stored on the account at execution time.

Common Signing Scenarios

Below are several scenarios in which different signature combinations are required to authorize a transaction.

Single party, single signature

The simplest Flow transaction declares a single account as the proposer, payer and authorizer. In this case, the account can sign the transaction with a single signature.

This scenario is only possible if the signature is generated by a key with full signing weight.

Account	Key ID	Weight
0x01	1	1000

```
json
{
  "payload": {
    "proposalKey": {
      "address": "0x01",
      "keyId": 1,
      "sequenceNumber": 42
    },
    "payer": "0x01",
    "authorizers": [ "0x01" ]
  },
  "payloadSignatures": [], // 0x01 is the payer, so only needs to sign envelope
  "envelopeSignatures": [
    {
      "address": "0x01",
      "keyId": 1,
      "sig": "0xabc123"
    }
  ]
}
```

Single party, multiple signatures

A transaction that declares a single account as the proposer, payer and authorizer may still specify multiple signatures if the account uses weighted keys to achieve multi-sig functionality.

Account	Key ID	Weight
0x01	1	500
0x01	2	500

```
json
{
```

```

"payload": {
    "proposalKey": {
        "address": "0x01",
        "keyId": 1,
        "sequenceNumber": 42
    },
    "payer": "0x01",
    "authorizers": [ "0x01" ]
},
"payloadSignatures": [], // 0x01 is the payer, so only needs to sign envelope
"envelopeSignatures": [
{
    "address": "0x01",
    "keyId": 1,
    "sig": "0xabc123"
},
{
    "address": "0x01",
    "keyId": 2,
    "sig": "0xdef456"
}
]
}

```

Multiple parties

A transaction that declares different accounts for each signing role will require at least one signature from each account.

Account	Key ID	Weight
---	---	---
0x01	1	1000
0x02	1	1000

```

json
{
    "payload": {
        "proposalKey": {
            "address": "0x01",
            "keyId": 1,
            "sequenceNumber": 42
        },
        "payer": "0x02",
        "authorizers": [ "0x01" ]
    },
    "payloadSignatures": [
    {
        "address": "0x01", // 0x01 is not payer, so only signs payload
        "keyId": 1,
        "sig": "0xabc123"
    }
    ],
}

```

```

    "envelopeSignatures": [
      {
        "address": "0x02",
        "keyId": 1,
        "sig": "0xdef456"
      },
    ],
}

```

Multiple parties, multiple signatures

A transaction that declares different accounts for each signing role may require more than one signature per account if those accounts use weighted keys to achieve multi-sig functionality.

Account	Key ID	Weight
---	---	---
0x01	1	500
0x01	2	500
0x02	1	500
0x02	2	500

```

json
{
  "payload": {
    "proposalKey": {
      "address": "0x01",
      "keyId": 1,
      "sequenceNumber": 42
    },
    "payer": "0x02",
    "authorizers": [ "0x01" ]
  },
  "payloadSignatures": [
    {
      "address": "0x01", // 0x01 is not payer, so only signs payload
      "keyId": 1,
      "sig": "0xabc123"
    },
    {
      "address": "0x01", // 0x01 is not payer, so only signs payload
      "keyId": 2,
      "sig": "0x123abc"
    }
  ],
  "envelopeSignatures": [
    {
      "address": "0x02",
      "keyId": 1,
      "sig": "0xdef456"
    },
    {
      "address": "0x02",
      "keyId": 2,
      "sig": "0x123abc"
    }
  ]
}

```

```
        "keyId": 2,  
        "sig": "0x456def"  
    },  
]  
}
```

Transaction Submission and Retrieval

You can use the Flow CLI to get an existing transaction by ID:

```
sh  
flow transactions get  
1ec90051e3bc74fc36cbd16fc83df08e463dda8f92e8e2193e061f9d41b2ad92 -n  
mainnet
```

Find more about the command in the CLI docs.

A user can define their own transactions or it can use already defined transactions by the contract authors that can be found by using the FLIX service.

Transactions can be submitted and obtained from the access node APIs, currently, there are two gRPC and REST APIs. You can find more information about them here:

[gRPC Transaction API](#)

[REST Transaction API](#)

There are multiple SDKs implementing the above APIs for different languages:

[Javascript SDK](#)

[Go SDK](#)

Find a list of all SDKs [here](#)

```
# 02-fungible-token.md:
```

```
---  
title: Fungible Token Contract  
sidebarposition: 2  
sidebarlabel: Fungible Token  
---
```

The FungibleToken contract implements the Fungible Token Standard. It is the second contract ever deployed on Flow.

- Basic Fungible Token Tutorial
- Fungible Token Guide

- Fungible Token Standard Repo

The FungibleTokenMetadataViews and FungibleTokenSwitchboard contracts are also deployed to the same account as FungibleToken.

Source: FungibleToken.cdc

Network	Contract Address
Emulator	0xee82856bf20e2aa6
Cadence Testing Framework	0x0000000000000002
Testnet	0x9a0766d93b6608b7
Mainnet	0xf233dcee88fe0abe

Transactions

All FungibleToken projects are encouraged to use the generic token transactions and scripts in the flow-ft repo. They can be used for any token that implements the fungible token standard properly without changing any code besides import addresses on different networks.

Events

Events emitted from all contracts follow a standard format:

A.{contract address}.{contract name}.{event name}

The components of the format are:

- contract address - the address of the account the contract has been deployed to
- contract name - the name of the contract in the source code
- event name - the name of the event as declared in the source code

FungibleToken Events

Contracts that implement the Fungible Token standard get access to standard events that are emitted every time a relevant action occurs, like depositing and withdrawing tokens.

This means that projects do not have to implement their own custom events unless the standard events do not satisfy requirements they have for events.

The FungibleToken events will have the following format:

A.{contract address}.FungibleToken.Deposited
A.{contract address}.FungibleToken.Withdrawn

Where the contract address is the FungibleToken address on the network being queried.

The addresses on the various networks are shown above.

FungibleToken.Deposited

```
cadence
access(all) event Deposited (
    type: String,
    amount: UFix64,
    to: Address?,
    toUUID: UInt64,
    depositedUUID: UInt64,
    balanceAfter: UFix64
)
```

Whenever deposit() is called on a resource type that implements FungibleToken.Vault, the FungibleToken.Deposited event is emitted with the following arguments:

- type: String: The type identifier of the token being deposited.
 - Example: A.4445e7ad11568276.FlowToken.Vault
- amount: UFix64: The amount of tokens that were deposited.
 - Example: 0.00017485
- to: Address?: The address of the account that owns the Vault that received
 - the tokens. If the vault is not stored in an account, to will be nil.
 - Example: 0x4445e7ad11568276
- toUUID: UInt64: The UUID of the Vault that received the tokens.
 - Example: 177021372071991
- depositedUUID: The UUID of the Vault that was deposited (and therefore destroyed).
 - Example: 177021372071991
- balanceAfter: UFix64: The balance of the Vault that received the tokens after the deposit happened.
 - Example: 1.00047545

FungibleToken.Withdrawn

```
cadence
access(all) event Withdrawn (
    type: String,
    amount: UFix64,
    from: Address?,
    fromUUID: UInt64,
    withdrawnUUID: UInt64,
    balanceAfter: UFix64
)
```

Whenever withdraw() is called on a resource type that implements FungibleToken.Vault, the FungibleToken.Withdrawn event is emitted with the following arguments:

- type: String: The type identifier of the token being withdrawn.
 - Example: A.4445e7ad11568276.FlowToken.Vault
- amount: UFix64: The amount of tokens that were withdrawn.
 - Example: 0.00017485
- from: Address?: The address of the account that owns the Vault that the tokens were withdrawn from. If the vault is not stored in an account, to will be nil.
 - Example: 0x4445e7ad11568276
- fromUUID: UInt64: The UUID of the Vault that the tokens were withdrawn from.
 - Example: 177021372071991
- withdrawnUUID: The UUID of the Vault that was withdrawn.
 - Example: 177021372071991
- balanceAfter: UFix64: The balance of the Vault that the tokens were withdrawn from after the withdrawal.
 - Example: 1.00047545

FungibleToken.Burned

```

cadence
access(all) event Burned (
    type: String,
    amount: UFix64,
    fromUUID: UInt64
)

```

Whenever a fungible token that implements FungibleToken.Vault is burned via the Burner.burn() method, this event is emitted with the following arguments:

- type: String: The type identifier of the token that was burnt.
 - Example: A.4445e7ad11568276.FlowToken.Vault
- amount: UFix64: The amount of tokens that were burnt.
 - Example: 0.00017485
- fromUUID: UInt64: The UUID of the Vault that was burnt.
 - Example: 177021372071991

```
# 03-flow-token.md:
```

```
---
title: Flow Token Contract
sidebarposition: 3
sidebarlabel: Flow Token
---
```

The FlowToken contract defines the FLOW network token.

Source: FlowToken.cdc

Network	Contract Address	
---------	------------------	--

Emulator	0xae53cb6e3f42a79
Cadence Testing Framework	0x0000000000000003
Testnet	0x7e60df042a9c0868
Mainnet	0x1654653399040a61

Transactions

Transactions and scripts for FlowToken are in the flow-core-contracts repo.

As mentioned in the FungibleToken page, developers are encouraged to use the generic token transactions in the flow-ft repo instead.

Events

Flow relies on a set of core contracts that define key portions of the Flow protocol. Those contracts are core contracts and are made to emit the events documented below. You can read about the core contracts here and view their source code and event definitions.

Events emitted from core contracts follow a standard format:

A.{contract address}.{contract name}.{event name}

The components of the format are:

- contract address - the address of the account the contract has been deployed to
- contract name - the name of the contract in the source code
- event name - the name of the event as declared in the source code

Flow Token Contract

Description of events emitted from the FLOW Token contract.
The contract defines the fungible FLOW token. Please note that events for the fungible token contracts are the same if deployed to a different account but the contract address is changed to the address of the account the contract has been deployed to.

Tokens Initialized

Event that is emitted when the contract gets created.

- Event name: TokensInitialized
- Mainnet event: A.1654653399040a61.FlowToken.TokensInitialized
- Testnet event: A.7e60df042a9c0868.FlowToken.TokensInitialized

```
cadence
access(all) event TokensInitialized(initialSupply: UFix64)
```

Field	Type	Description
initialSupply	UFix64	The initial supply of the tokens

Tokens Withdrawn

Event that is emitted when tokens get withdrawn from a Vault.

- Event name: TokensWithdrawn
- Mainnet event: A.1654653399040a61.FlowToken.TokensWithdrawn
- Testnet event: A.7e60df042a9c0868.FlowToken.TokensWithdrawn

cadence

```
access(all) event TokensWithdrawn(amount: UFix64, from: Address?)
```

Field	Type	Description
amount	UFix64	The amount of tokens withdrawn
from	Address?	Optional address of the account that owns the vault where tokens were withdrawn from. nil if the vault is not in an account's storage

Tokens Deposited

Event that is emitted when tokens get deposited to a Vault.

- Event name: TokensDeposited
- Mainnet event: A.1654653399040a61.FlowToken.TokensDeposited
- Testnet event: A.7e60df042a9c0868.FlowToken.TokensDeposited

cadence

```
access(all) event TokensDeposited(amount: UFix64, to: Address?)
```

Field	Type	Description
amount	UFix64	The amount of tokens withdrawn
to	Address?	Optional address of the account that owns the vault where tokens were deposited to. nil if the vault is not in an account's storage

Tokens Minted

Event that is emitted when new tokens gets minted.

- Event name: TokensMinted
- Mainnet event: A.1654653399040a61.FlowToken.TokensMinted
- Testnet event: A.7e60df042a9c0868.FlowToken.TokensMinted

```
cadence
access(all) event TokensMinted(amount: UFix64)
```

Field	Type	Description
-----	-----	-----
amount	UFix64	The amount of tokens to mint

Tokens Burned

Event that is emitted when tokens get destroyed.

- Event name: TokensBurned
- Mainnet event: A.1654653399040a61.FlowToken.TokensBurned
- Testnet event: A.7e60df042a9c0868.FlowToken.TokensBurned

```
cadence
access(all) event TokensBurned(amount: UFix64)
```

Field	Type	Description
-----	-----	-----
amount	UFix64	The amount of tokens to burn

Minter Created

Event that is emitted when a new minter resource gets created.

- Event name: MinterCreated
- Mainnet event: A.1654653399040a61.FlowToken.MinterCreated
- Testnet event: A.7e60df042a9c0868.FlowToken.MinterCreated

```
cadence
access(all) event MinterCreated(allowedAmount: UFix64)
```

Field	Type	Description
-----	-----	-----
allowedAmount	UFix64	The amount of tokens that the minter is allowed to mint

Burner Created

Event that is emitted when a new burner Resource gets created.

- Event name: BurnerCreated
- Mainnet event: A.1654653399040a61.FlowToken.BurnerCreated

- Testnet event: A.7e60df042a9c0868.FlowToken.BurnerCreated

```
cadence
access(all) event BurnerCreated()
```

Staking Events

To learn more about staking events, read [staking/events/](#)

04-service-account.md:

```
---
title: Service Account Contracts
sidebarposition: 4
sidebarlabel: Service Account
---
```

The service account is the account that manages the core protocol requirements of Flow.

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x8c5303eaa26202d6
Mainnet	0xe467b9dd11fa00df

Here are three important contracts deployed to the service account:

FlowServiceAccount

FlowServiceAccount tracks transaction fees, deployment permissions, and provides some convenience methods for Flow Token operations.

Source: [FlowServiceAccount.cdc](#)

Events

Important events from FlowServiceAccount are:

```
cadence
access(all) event TransactionFeeUpdated(newFee: UFix64)
access(all) event AccountCreationFeeUpdated(newFee: UFix64)
```

RandomBeaconHistory

- RandomBeaconHistory stores the history of random sources generated by the Flow network. The defined Heartbeat resource is updated by the Flow Service Account at the end of every block with that block's source of randomness.

Source: RandomBeaconHistory.cdc

Events

Important events from RandomBeaconHistory are:

```
cadence
// Event emitted when missing SoRs from past heartbeats are detected and
will be backfilled:
// - blockHeight is the height where the gap is detected
// - gapStartHeight is the height of the first missing entry detected
access(all) event RandomHistoryMissing(blockHeight: UInt64,
gapStartHeight: UInt64)

// Event emitted when missing SoRs are backfilled on the current
heartbeat:
// - blockHeight is the height where the backfill happened, it also
defines the SoR used to backfill
// - gapStartHeight is the height of the first backfilled entry
// - count is the number of backfilled entries
// Note that in very rare cases, the backfilled gap may not be
contiguous. This event does not
// fully define the backfilled entries in this case.
access(all) event RandomHistoryBackfilled(blockHeight: UInt64,
gapStartHeight: UInt64, count: UInt64)
```

NodeVersionBeacon

- NodeVersionBeacon holds the past and future protocol versions that should be used to execute/handle blocks at a given block height.

Source: NodeVersionBeacon.cdc

Events

Important events from NodeVersionBeacon are:

```
cadence
/// Event emitted when the version table is updated.
/// It contains the current version and all the upcoming versions
/// sorted by block height.
/// The sequence increases by one each time an event is emitted.
/// It can be used to verify no events were missed.
access(all) event VersionBeacon(
    versionBoundaries: [VersionBoundary],
    sequence: UInt64
)

/// Event emitted any time the version boundary freeze period is updated.
/// freeze period is measured in blocks (from the current block).
```

```
access(all) event NodeVersionBoundaryFreezePeriodChanged(freezePeriod:  
UInt64)
```

05-flow-fees.md:

```
---  
title: Flow Fees Contract  
sidebarposition: 5  
sidebarlabel: Flow Fees  
---
```

FlowFees

The FlowFees contract is where all the collected flow fees are gathered.

Source: FlowFees.cdc

Network	Contract Address
Emulator	0xe5a8b7f23e8b548f
Cadence Testing Framework	0x0000000000000004
Testnet	0x912d5440f7e3769e
Mainnet	0xf919ee77447b7497

Events

Important events for FlowFees are:

```
cadence  
// Event that is emitted when tokens are deposited to the fee vault  
access(all) event TokensDeposited(amount: UFix64)  
  
// Event that is emitted when tokens are withdrawn from the fee vault  
access(all) event TokensWithdrawn(amount: UFix64)  
  
// Event that is emitted when fees are deducted  
access(all) event FeesDeducted(amount: UFix64, inclusionEffort: UFix64,  
executionEffort: UFix64)  
  
// Event that is emitted when fee parameters change  
access(all) event FeeParametersChanged(surgeFactor: UFix64,  
inclusionEffortCost: UFix64, executionEffortCost: UFix64)
```

FlowStorageFees

The FlowStorageFees contract defines the parameters and utility methods for storage fees.

Source: FlowStorageFees.cdc

Network	Contract Address
---------	------------------

Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x000000000000000000000001
Testnet	0x8c5303eaa26202d6
Mainnet	0xe467b9dd11fa00df

Events

Important events for FlowStorageFees are:

```
cadence
// Emitted when the amount of storage capacity an account has per
reserved Flow token changes
access(all) event StorageMegaBytesPerReservedFLOWChanged(
storageMegaBytesPerReservedFLOW: UFix64)

// Emitted when the minimum amount of Flow tokens that an account needs
to have reserved for storage capacity changes.
access(all) event MinimumStorageReservationChanged(
minimumStorageReservation: UFix64)
```

06-staking-contract-reference.md:

```
---
title: Flow Staking Contract Reference
sidebarposition: 6
sidebarlabel: Staking Table
---
```

Contract

The FlowIDTableStaking contract is the central table that manages staked nodes, delegation and rewards.

Source: FlowIDTableStaking.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x000000000000000000000001
Testnet	0x9eca2b38b18b5dfe
Mainnet	0x8624b52f9ddcd04a

Transactions and Scripts

Transactions for the staking contract are in the flow-core-contracts repo.

Developers and users are advised to use the staking collection transactions

to stake tokens instead of the basic transactions that are used for tests.

Getting Staking Info with Scripts

These scripts are read-only and get info about the current state of the staking contract.

ID	Name	Source
----- ----- -----		

SC.01 Get Delegation Cut Percentage idTableStaking/getcutpercentage.cdc		
SC.02 Get Minimum Stake Requirements idTableStaking/getstakerequirements.cdc		
SC.03 Get Total Weekly Reward Payout idTableStaking/getweeklypayout.cdc		
SC.04 Get Current Staked Node Table idTableStaking/getcurrenttable.cdc		
SC.05 Get Proposed Staked Node Table idTableStaking/getproposedtable.cdc		
SC.06 Get Total Flow Staked idTableStaking/gettotalstaked.cdc		
SC.07 Get Total Flow Staked by Node Type idTableStaking/gettotalstakedbytype.cdc		
SC.08 Get All Info about a single NodeID idTableStaking/getnodeinfo.cdc		
SC.09 Get a node's total Commitment (delegators) idTableStaking/getnodetotalcommitment.cdc		
SC.10 Get All Info about a single Delegator idTableStaking/delegation/getdelegatorinfo.cdc		
SC.11 Get a node's total Commitment idTableStaking/getnodetotalcommitmentwithoutdelegators.cdc		

Delegator Transactions

Documentation for delegating with tokens is described in the staking documentation for the staking collection

Events

The FlowIDTableStaking contract emits an event whenever an important action occurs.

See the staking events Documentation for more information about each event.

```
cadence
    /// Epoch
    access(all) event NewEpoch(
        totalStaked: UFix64,
        totalRewardPayout: UFix64,
        newEpochCounter: UInt64
    )
    access(all) event EpochTotalRewardsPaid(
        total: UFix64,
        fromFees: UFix64,
        minted: UFix64,
        feesBurned: UFix64,
        epochCounterForRewards: UInt64
    )

    /// Node
    access(all) event NewNodeCreated(nodeID: String, role: UInt8,
amountCommitted: UFix64)
        access(all) event TokensCommitted(nodeID: String, amount: UFix64)
        access(all) event TokensStaked(nodeID: String, amount: UFix64)
        access(all) event NodeTokensRequestedToUnstake(nodeID: String,
amount: UFix64)
        access(all) event TokensUnstaking(nodeID: String, amount: UFix64)
        access(all) event TokensUnstaked(nodeID: String, amount: UFix64)
        access(all) event NodeRemovedAndRefunded(nodeID: String, amount:
UFix64)
        access(all) event RewardsPaid(nodeID: String, amount: UFix64,
epochCounter: UInt64)
        access(all) event UnstakedTokensWithdrawn(nodeID: String, amount:
UFix64)
        access(all) event RewardTokensWithdrawn(nodeID: String, amount:
UFix64)
        access(all) event NetworkingAddressUpdated(nodeID: String,
newAddress: String)
        access(all) event NodeWeightChanged(nodeID: String, newWeight:
UInt64)

    /// Delegator
    access(all) event NewDelegatorCreated(nodeID: String, delegatorID:
UInt32)
        access(all) event DelegatorTokensCommitted(nodeID: String,
delegatorID: UInt32, amount: UFix64)
        access(all) event DelegatorTokensStaked(nodeID: String, delegatorID:
UInt32, amount: UFix64)
        access(all) event DelegatorTokensRequestedToUnstake(nodeID: String,
delegatorID: UInt32, amount: UFix64)
        access(all) event DelegatorTokensUnstaking(nodeID: String,
delegatorID: UInt32, amount: UFix64)
        access(all) event DelegatorTokensUnstaked(nodeID: String,
delegatorID: UInt32, amount: UFix64)
```

```

    access(all) event DelegatorRewardsPaid(nodeID: String, delegatorID: UInt32, amount: UFix64, epochCounter: UInt64)
    access(all) event DelegatorUnstakedTokensWithdrawn(nodeID: String, delegatorID: UInt32, amount: UFix64)
    access(all) event DelegatorRewardTokensWithdrawn(nodeID: String, delegatorID: UInt32, amount: UFix64)

    /// Contract Fields
    access(all) event NewDelegatorCutPercentage(newCutPercentage: UFix64)
    access(all) event NewWeeklyPayout(newPayout: UFix64)
    access(all) event NewStakingMinimums(newMinimums: {UInt8: UFix64})
    access(all) event NewDelegatorStakingMinimum(newMinimum: UFix64)

```

07-epoch-contract-reference.md:

```
---
title: Flow Epoch Contracts Reference
sidebarposition: 7
sidebarlabel: Epoch Contracts
---
```

Contract

The FlowEpoch contract is the state machine that manages Epoch phases and emits service events.

The FlowClusterQC and FlowDKG contracts manage the processes that happen during the Epoch Setup phase.

These contracts are all deployed to the same account as the FlowIDTableStaking contract.

Sources:

- FlowEpoch.cdc
- FlowClusterQC.cdc
- FlowDKG.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x9eca2b38b18b5dfe
Mainnet	0x8624b52f9ddcd04a

Transactions

Getting Epoch Info

These scripts are read-only and get info about the current state of the epoch contract.

ID	Name	Source
EP.01	Get Epoch Metadata	epoch/getepochmetadata.cdc
EP.02	Get Configurable Metadata	epoch/getconfigmetadata.cdc
EP.03	Get Epoch Counter	epoch/getepochcounter.cdc
EP.04	Get Epoch Phase	epoch/getepochphase.cdc

Quorum Certificate Transactions and Scripts

ID	Name	Source
QC.01	Create QC Voter	quorumCertificate/getepochmetadata.cdc
QC.02	Submit QC Vote	quorumCertificate/getconfigmetadata.cdc
QC.03	Get Collector Cluster	quorumCertificate/scripts/getcluster.cdc
QC.04	Get QC Enabled	quorumCertificate/scripts/getqcenabled.cdc
QC.05	Get Node Has Voted	quorumCertificate/scripts/getnodehasvoted.cdc
QC.06	Get QC Voting Complete	quorumCertificate/scripts/getvotingcompleted.cdc

DKG Transactions and Scripts

ID	Name	Source
DKG.01	Create DKG Participant	dkg/createtimeparticpant.cdc
DKG.02	Get Configurable Metadata	dkg/sendwhiteboardmessage.cdc
DKG.03	Send Final Submission	dkg/sendfinalsubmission.cdc
DKG.04	Get DKG Enabled	dkg/scripts/getdkgenabled.cdc
DKG.05	Get DKG Completed	dkg/scripts/getdkgcompleted.cdc
DKG.06	Get Whiteboard Messages	dkg/scripts/getwhiteboardmessages.cdc
DKG.07	Get Final Submissions	dkg/scripts/getfinalsubmissions.cdc
DKG.08	Get Node Has Submitted	dkg/scripts/getnodehassubmitted.cdc

Events

See the epoch documentation for a list and documentation for important FlowEpoch events.

08-non-fungible-token.md:

```
---
```

title: Non-Fungible Token Contract
sidebarposition: 8
sidebarlabel: Non-Fungible Token

The NonFungibleToken contract interface implements the Fungible Token Standard.

All NFT contracts are encouraged to import and implement this standard.

- Basic Non-Fungible Token Tutorial
- Non Fungible Token Guide
- Non Fungible Token Standard Repo

Source: NonFungibleToken.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x631e88ae7f1d7c20
Mainnet	0x1d7e57aa55817448

Transactions

All NonFungibleToken projects are encouraged to use the generic token transactions and scripts in the flow-nft repo. They can be used for any token that implements the non-fungible token standard properly without changing any code besides import addresses on different networks.

Events

Events emitted from all contracts follow a standard format:

A.{contract address}.{contract name}.{event name}

The components of the format are:

- contract address - the address of the account the contract has been deployed to
- contract name - the name of the contract in the source code
- event name - the name of the event as declared in the source code

NonFungibleToken Events

Contracts that implement the Non-Fungible Token standard get access to standard events that are emitted every time a relevant action occurs, like depositing and withdrawing tokens.

This means that projects do not have to implement their own custom events unless the standard events do not satisfy requirements they have for events.

The NonFungibleToken events will have the following format:

```
A.{contract address}.NonFungibleToken.Deposited  
A.{contract address}.NonFungibleToken.Withdrawn
```

Where the contract address is the NonFungibleToken address on the network being queried.

The addresses on the various networks are shown above.

```
NonFungibleToken.Deposited
```

```
cadence  
access(all) event Deposited (  
    type: String,  
    id: UInt64,  
    uuid: UInt64,  
    to: Address?,  
    collectionUUID: UInt64  
)
```

Whenever deposit() is called on a resource type that implements NonFungibleToken.Collection, the NonFungibleToken.Deposited event is emitted with the following arguments:

- type: String: The type identifier of the token being deposited.
 - Example: A.4445e7ad11568276.TopShot.NFT
- id: UInt64: The ID of the token that was deposited. Note: This may or may not be the UUID.
 - Example: 173838
- uuid: UInt64: The UUID of the token that was deposited.
 - Example: 177021372071991
- to: Address?: The address of the account that owns the Collection that received the token. If the collection is not stored in an account, to will be nil.
 - Example: 0x4445e7ad11568276
- collectionUUID: UInt64: The UUID of the Collection that received the token.
 - Example: 177021372071991

```
NonFungibleToken.Withdrawn
```

```
cadence
access(all) event Withdrawn (
    type: String,
    id: UInt64,
    uuid: UInt64,
    from: Address?,
    providerUUID: UInt64
)
```

Whenever `withdraw()` is called on a resource type that implements `NonFungibleToken.Collection`, the `NonFungibleToken.Withdrawn` event is emitted with the following arguments:

- `type: String`: The type identifier of the token being withdrawn.
 - Example: A.4445e7ad11568276.TopShot.NFT
- `id: UInt64`: The id of the token that was withdrawn. Note: May or may not be the UUID.
 - Example: 113838
- `uuid: UInt64`: The UUID of the token that was withdrawn.
 - Example: 177021372071991
- `from: Address?`: The address of the account that owns the Collection that the token was withdrawn from. If the collection is not stored in an account, to will be nil.
 - Example: 0x4445e7ad11568276
- `providerUUID: UInt64`: The UUID of the Collection that the token was withdrawn from.
 - Example: 177021372071991

```
NonFungibleToken.Updated
```

```
cadence
access(all) event Updated(
    type: String,
    id: UInt64,
    uuid: UInt64,
    owner: Address?
)
```

Whenever a non-fungible token is updated for whatever reason, projects should call the `NonFungibleToken.emitNFTUpdated()` function to emit this event. It indicates to event listeners that they should query the NFT to update any stored information they have about the NFT in their database.

- `type: String`: The type identifier of the token that was updated.
 - Example: A.4445e7ad11568276.TopShot.NFT

```
- id: UInt64: The ID of the token that was updated. Note: This may or may  
not be the UUID.  
- Example: 173838  
- uuid: UInt64: The UUID of the token that was updated.  
- Example: 177021372071991  
- owner: Address?: The address of the account that owns the Collection  
that owns  
the token. If the collection is not stored in an account, to will be  
nil.  
- Example: 0x4445e7ad11568276
```

09-nft-metadata.md:

```
---
```

title: NFT Metadata Contract
sidebarposition: 9
sidebarlabel: NFT Metadata

The ViewResolver and MetadataViews contracts implement a standard to attach on-chain metadata to NFTs. This standard was originally proposed in FLIP-0636.

It is deployed at the same address as the NonFungibleToken contract interface.

Source: ViewResolver.cdc

Source: MetadataViews.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x631e88ae7f1d7c20
Mainnet	0x1d7e57aa55817448

There exists a tool, Flow NFT Catalog, which enables dapp developers the ability to unlock interoperability of your NFT collection across the Flow ecosystem. This will help make your NFT collection's metadata more discoverable and interoperable.

To optimize your NFT collections for this catalog, you'll need to:

1. Update your NFT contract to support ViewResolver and MetadataViews with implementation of the core NFT views.
2. Deploy the updated contract to both testnet and mainnet.
3. Afterwards, onboard your NFT to the Flow NFT catalog at <https://flow-nft-catalog.com>.

10-nft-storefront.md:

```
---
title: NFT Storefront Smart Contract
sidebarposition: 10
sidebarlabel: NFT Storefront
---
```

The NFTStorefront contracts implement a standard way to list NFTs for sale and buy them from listings. NFTStorefrontV2 is the more powerful and full-featured version, so developers and users are encouraged to use it instead of NFTStorefront or their own implementation.

Source: [NFTStorefrontV2.cdc]

Network	Contract Address
Testnet	0x2d55b98eb200daef
Mainnet	0x4eb8a10cb9f87357

Source: [NFTStorefront.cdc]

Network	Contract Address
Testnet	0x94b06cfca1d8a476
Mainnet	0x4eb8a10cb9f87357

Primer

The NFTStorefrontV2 contract lets you create a non-custodial Resource (NFT) marketplace on the FLOW blockchain.

NFTStorefrontV2 makes it simple for Sellers to list NFTs in dApp specific marketplaces. DApp developers leverage the APIs provided by the contract to manage listings being offered for sale and to transact NFT trades.

!dapps1

Developers should use the NFTStorefrontV2 to create their marketplace and to enable p2p purchases. The diagram below shows how dApps can facilitate the creation of NFT listings for different marketplaces and how marketplaces can filter their listings.

Listings made through a specific dApp storefront can be simultaneously listed on 3rd party marketplaces beyond that dApp. Well known 3rd party marketplaces listen for compatible NFT listing events enabling the automation of listings into their marketplace dashboards.

!dapps2

Using the NFTStorefrontV2, marketplaces can instantly and easily tap into the vibrant FLOW NFT ecosystem and allow NFT holders to list their NFTs and enables creator royalties.

Marketplaces then process an NFT trade by interacting directly with seller storefronts. Flow's account based model ensures that NFTs listed for sale always reside in the Seller account until traded, regardless of how many listings are posted across any number of marketplaces, for the same NFT.

`!marketplace1`

Functional Overview

A general purpose sale support contract for NFTs implementing the Flow [NonFungibleToken] standard.

Each account that wants to list NFTs for sale creates a Storefront resource to store in their account and lists individual sales within that Storefront as Listings. There is usually one Storefront per account held at the `/storage/NFTStorefrontV2`.

Each listing can define one or more sale cuts taken out of the sale price to go to one or more addresses. Listing fees, royalties, or other considerations can be paid using sale cuts. Also, the listing can include a commission as one of these sale cuts is paid to whoever facilitates the purchase.

Listings can have an optional list of marketplace [receiver capabilities] used to receive the commission for fulfilling the listing. An NFT may be listed in one or more Listings, and the validity of each listing can easily be checked.

Interested parties can globally track Listing events on-chain and filter by NFT types, IDs and other characteristics to determine which to make available for purchase within their own marketplace UIs."

Selling NFTs

NFTStorefrontV2 offers a generic process for creating the listing for an NFT. It provides all the essential APIs to manage those listings independently.

Many marketplaces create a single storefront resource to manage different individual listings. We recommend creating the listing under the user-owned storefront resource to make it trustless and platform-independent. Users should possess the Storefront resource under their account to create the listing using the storefront contract.

Creating a successful listing using the NFTStorefrontV2 contract.

As recommended above, the first step is to create and store the [Storefront resource] in the user account using the [setupaccount] transaction.

The next step is to create a listing under the newly created storefront resource. If the user (repetitive) already holds the storefront resource, then use the existing resource. The seller can come with multiple

requirements for listing their NFTs, and We try our best to cover most of them below.

Scenario 1: Selling NFTs corresponds to more than one cryptocurrency, i.e. FLOW, USDC etc.

The NFTStorefrontV2 contract doesn't support selling an NFT for multiple different currencies with a single listing. However, this can be achieved by creating multiple listings for the same NFT for each different currency.

Example - Alice wants to sell a kitty and is open to receiving FLOW and USDC

```
!scenario1
```

Putting an NFT on sell called listing, seller can create a listing using [sellitem] transaction by providing some required details to list an NFT, i.e. Receiving currency type, [Capability] from where NFT will be deducted etc. If interested look [createListing] for more details.

To receive a different currency seller has to provide a different Receiver currency type , i.e. salePaymentVaultType As depicted in the above diagram, There are two listing formations with almost the same inputs. The only differentiator is the salePaymentVaultType parameter that needs to be different when creating duplicate NFT listings with different sale currency types.

Scenario 2: Peer-to-Peer (p2p) listing of NFT: A listing anyone can fulfil.

Dapps can leverage the NFTStorefrontV2 to facilitate the creation of a listing for the seller independent of any marketplace. Dapps or marketplaces can list those listings on their platforms, or seller can settle it p2p.

The seller can use [sellitem] transaction to create a p2p listing, providing the marketplacesAddress with an empty array. The seller has a choice of providing [commission] to the facilitator of sale, which can also act as a discount if the facilitator and the purchaser are the same.

Scenario 3: The seller wants to list its NFT in different marketplaces.

NFTStorefrontV2 offers two different ways of doing it.

- The seller can create a listing and provide the marketplacesAddress that it wants to have a listing on using [sellitem] transaction.

Marketplaces can listen to ListingAvailable events and check whether their address is included in the commissionReceivers list; If yes, the marketplace would be rewarded during the successful fulfilment of the listing.

Example - Bob wants to list on marketplace 0xA, 0xB & 0xC and is willing to offer 10% commission on the sale price of the listing to the marketplaces.

```
!scenario3
```

- Another way to accomplish this is to create separate listings for each marketplace on which a user wants their listing using [sellitemwithmarketplacecut] transaction. In this case, the marketplace would be incentivized by earning one of the parts of the [saleCut] by appending marketplace saleCut in saleCuts array during the creation of the listing.

Considerations

1. Ghost listings - Ghost listings are listings which don't have an underlying NFT in the seller's account. However, the listing is still available for buyers to attempt to purchase. StorefrontV2 is not immune to ghost listings. Usually, ghost listings will cause a purchaser's transaction to fail, which is annoying but isn't a significant problem. Ghost listings become a problem for the seller when the listed NFT comes back to the seller's account after its original sale. The ghost listing will no longer be invalid when it comes back, and anyone can purchase it even if the seller doesn't want to sell it at that price anymore.

Note - We recommend that marketplaces and p2p dApps create an off-chain notification service that tells their users (i.e., sellers) to remove the listings if they don't hold the NFT anymore in the same account.

2. Expired listings - NFTStorefrontV2 introduces a safety measure to specify that a listing will expire after a certain period that can be set during the creation so no one can purchase the listing anymore. It is not a fool-proof safety measure, but it does give some safe ground to the sellers for the ghost listings & stale listings.

Note - We recommended for marketplaces and p2p dApps not to show the expired listings on their dashboards.

Purchasing NFTs

Purchasing NFTs through the NFTStorefrontV2 is simple. The buyer has to provide the payment vault and the commissionRecipient , if applicable, during the purchase. p2p dApps don't need any intermediaries to facilitate the purchase of listings. [purchase] API offered by the Listing resource gets used to facilitate the purchase of NFT.

During the listing purchase all saleCuts are paid automatically. This also includes distributing royalties for that NFT, if applicable. If the vault provided by the buyer lacks sufficient funds then the transaction will fail.

Considerations

1. Auto cleanup - NFTStorefrontV2 offers a unique ability to do auto cleanup of duplicate listings during a purchase. It comes with a drawback if one NFT has thousands of duplicate listings. It will become the bottleneck during purchasing one of the listings as it will likely trigger an out-of-gas error.

Note - We recommended NOT to have more than 50 (TBD) duplicate listings of any given NFT.

2. Unsupported receiver capability - A common pitfall during the purchase of an NFT that some saleCut receivers don't have a supported receiver capability because that entitled sale cut would transfer to first valid sale cut receiver. However, it can be partially solved by providing the generic receiver using the [FungibleTokenSwitchboard] contract and adding all the currency capabilities the beneficiary wants to receive. More on the FungibleTokenSwitchboard can be read in [Fungible Token Switchboard]

Enabling creator royalties for NFTs

The NFTStorefrontV2 contract optionally supports paying royalties to the minter account for secondary resales of that NFT after the original sale. Marketplaces decide for themselves whether to support creator royalties when validating listings for sale eligibility. We encourage all marketplaces to support creator royalties and support community creators in the FLOW ecosystem.

Providing that a seller's NFT supports the [Royalty Metadata View] standard, then marketplaces can honor royalties payments at time of purchase. NFTStorefrontV2 dynamically calculates the royalties owed at the time of listing creation and applies it as a saleCut of the listing at the time of purchase.

```
cadence
// Check whether the NFT implements the MetadataResolver or not.
if nft.getViews().contains(Type<MetadataViews.Royalties>()) {
    // Resolve the royalty view
    let royaltiesRef = nft.resolveView(Type<MetadataViews.Royalties>())
    ?? panic("Unable to retrieve the royalties view for the NFT with
type "
        .concat(nft.getType().identifier).concat(" and ID ")
        .concat(nft.id.toString()).concat(".")
    // Fetch the royalties.
    let royalties = (royaltiesRef as!
MetadataViews.Royalties).getRoyalties()

    // Append the royalties as the salecut
    for royalty in royalties {
        self.saleCuts.append(NFTStorefrontV2.SaleCut(receiver:
royalty.receiver, amount: royalty.cut effectiveSaleItemPrice))
        totalRoyaltyCut = totalRoyaltyCut + royalty.cut
effectiveSaleItemPrice
    }
}
```

Complete transaction can be viewed in [sellitem].

saleCut only supports a single token receiver type and therefore beneficiaries of a saleCut can also only receive the token type used for the purchase. To support different token types for saleCuts we recommend using the [FungibleTokenSwitchboard] contract. The contract defines a generic receiver for fungible tokens which itself handles routing of tokens to the respective vault for that token type. Learn more about this in [Fungible Token Switchboard].

Enabling marketplace commissions for NFT sales

NFTStorefrontV2 enables optional commissions on trades for marketplaces which require it as a condition to list a NFT for sale. Commission & commission receivers are set by the seller during initial listing creation. At time of purchase the commission amount is paid once only to the commission receiver matching the marketplace receiver address which facilitated the sale.

For NFT listings in marketplaces which don't require commission, commission receivers can be set as nil. Setting the buyer of the NFT and commissionRecipient to the same has the effect of applying a discount for the buyer.

!scenario2.

APIs & Events offered by NFTStorefrontV2

Resource Interface ListingPublic

```
cadence
resource interface ListingPublic {
    access(all) fun borrowNFT(): &NonFungibleToken.NFT?
    access(all) fun purchase(
        payment: @FungibleToken.Vault,
        commissionRecipient: Capability<&{FungibleToken.Receiver}>?,
        ): @NonFungibleToken.NFT
    access(all) fun getDetails(): ListingDetail
    access(all) fun getAllowedCommissionReceivers():
        [Capability<&{FungibleToken.Receiver}>]?
}
```

An interface providing a useful public interface to a Listing.

Functions

```
fun borrowNFT()

cadence
fun borrowNFT(): &NonFungibleToken.NFT?
```

This will assert in the same way as the NFT standard borrowNFT() if the NFT is absent, for example if it has been sold via another listing.

```
fun purchase()
```

cadence

```
fun purchase(payment FungibleToken.Vault, commissionRecipient  
Capability<&{FungibleToken.Receiver}>?) : NonFungibleToken.NFT
```

Facilitates the purchase of the listing by providing the payment vault and the commission recipient capability if there is a non-zero commission for the given listing.

Respective saleCuts are transferred to beneficiaries and function return underlying or listed NFT.

```
fun getDetails()
```

cadence

```
fun getDetails(): ListingDetails
```

Fetches the details of the listings

```
fun getAllowedCommissionReceivers()
```

cadence

```
fun getAllowedCommissionReceivers():  
[Capability<&{FungibleToken.Receiver}>]?
```

Fetches the allowed marketplaces capabilities or commission receivers for the underlying listing.

If it returns nil then commission is up to grab by anyone.

Resource Storefront

cadence

```
resource Storefront {  
    access(all) fun createListing(  
        nftProviderCapability:  
    Capability<&{NonFungibleToken.Provider,  
    NonFungibleToken.CollectionPublic}>,  
        nftType: Type,  
        nftID: UInt64,
```

```

        salePaymentVaultType: Type,
        saleCuts: [SaleCut],
        marketplacesCapability:
[Capability<&{FungibleToken.Receiver}>]?,
        customID: String?,
        commissionAmount: UFix64,
        expiry: UInt64
    ): UInt64
    access(all) fun removeListing(listingResourceID: UInt64)
    access(all) fun getListingIDs(): [UInt64]
    access(all) fun getDuplicateListingIDs(nftType: Type, nftID: UInt64,
listingID: UInt64): [UInt64]
    access(all) fun cleanupExpiredListings(fromIndex: UInt64, toIndex:
UInt64)
    access(all) fun borrowListing(listingResourceID: UInt64):
&Listing{ListingPublic}?
}

```

A resource that allows its owner to manage a list of Listings, and anyone to interact with them in order to query their details and purchase the NFTs that they represent.

Implemented Interfaces:

- StorefrontManager
- StorefrontPublic

Initializer

```

cadence
fun init()

```

Functions

```

fun createListing()

cadence
fun createListing(nftProviderCapability
Capability<&{NonFungibleToken.Provider,
NonFungibleToken.CollectionPublic}>, nftType Type, nftID UInt64,
salePaymentVaultType Type, saleCuts [SaleCut], marketplacesCapability
[Capability<&{FungibleToken.Receiver}>]?, customID String?,
commissionAmount UFix64, expiry UInt64): UInt64

```

insert

Create and publish a Listing for an NFT.

```

fun removeListing()

```

```
cadence
fun removeListing(listingResourceID UInt64)

removeListing
Remove a Listing that has not yet been purchased from the collection and
destroy it.

---

fun getListingIDs()

cadence
fun getListingIDs(): [UInt64]

getListingIDs
Returns an array of the Listing resource IDs that are in the collection

---

fun getDuplicateListingIDs()

cadence
fun getDuplicateListingIDs(nftType Type, nftID UInt64, listingID UInt64): [UInt64]

getDuplicateListingIDs
Returns an array of listing IDs that are duplicates of the given nftType
and nftID.

---

fun cleanupExpiredListings()

cadence
fun cleanupExpiredListings(fromIndex UInt64, toIndex UInt64)

cleanupExpiredListings
Cleanup the expired listing by iterating over the provided range of
indexes.

---

fun borrowListing()

cadence
fun borrowListing(listingResourceID: UInt64): &{ListingPublic}?

borrowListing
```

Returns a read-only view of the listing for the given listingID if it is contained by this collection.

Resource Interface StorefrontPublic

```
cadence
resource interface StorefrontPublic {
    access(all) fun getListingIDs(): [UInt64]
    access(all) fun getDuplicateListingIDs(nftType: Type, nftID: UInt64,
listingID: UInt64): [UInt64]
    access(all) fun cleanupExpiredListings(fromIndex: UInt64, toIndex:
UInt64)
    access(all) fun borrowListing(listingResourceID: UInt64):
&Listing{ListingPublic}?
    access(all) fun cleanupPurchasedListings(listingResourceID: UInt64)
    access(all) fun getExistingListingIDs(nftType: Type, nftID: UInt64):
[UInt64]
}
```

StorefrontPublic

An interface to allow listing and borrowing Listings, and purchasing items via Listings in a Storefront.

Functions

fun getListingIDs()

```
cadence
fun getListingIDs(): [UInt64]
```

getListingIDs Returns an array of the Listing resource IDs that are in the collection

fun getDuplicateListingIDs()

```
cadence
fun getDuplicateListingIDs(nftType Type, nftID UInt64, listingID UInt64):
[UInt64]
```

getDuplicateListingIDs Returns an array of listing IDs that are duplicates of the given nftType and nftID.

fun borrowListing()

```
cadence
fun borrowListing(listingResourceID UInt64): &Listing{ListingPublic}?
```

borrowListing Returns a read-only view of the listing for the given listingID if it is contained by this collection.

```
fun cleanupExpiredListings()
```

```
cadence
```

```
fun cleanupExpiredListings(fromIndex UInt64, toIndex UInt64)
```

cleanupExpiredListings Cleanup the expired listing by iterating over the provided range of indexes.

```
fun cleanupPurchasedListings()
```

```
cadence
```

```
fun cleanupPurchasedListings(listingResourceID: UInt64)
```

```
cleanupPurchasedListings
```

Allows anyone to remove already purchased listings.

```
fun getExistingListingIDs()
```

```
cadence
```

```
fun getExistingListingIDs(nftType Type, nftID UInt64): [UInt64]
```

```
getExistingListingIDs
```

Returns an array of listing IDs of the given nftType and nftID.

Events

```
event StorefrontInitialized
```

```
cadence
```

```
event StorefrontInitialized(storefrontResourceID: UInt64)
```

A Storefront resource has been created. Consumers can now expect events from this Storefront. Note that we do not specify an address: we cannot and should not. Created resources do not have an owner address, and may be moved

after creation in ways we cannot check. ListingAvailable events can be used to determine the address of the owner of the Storefront at the time of the listing but only at that precise moment in that precise transaction. If the seller moves the Storefront while the listing is valid, that is on them.

event StorefrontDestroyed

cadence

event StorefrontDestroyed(storefrontResourceID: UInt64)

A Storefront has been destroyed. Event consumers can now stop processing events from this Storefront.

Note - we do not specify an address.

event ListingAvailable

cadence

event ListingAvailable(storefrontAddress: Address, listingResourceID: UInt64, nftType: Type, nftUUID: UInt64, nftID: UInt64, salePaymentVaultType: Type, salePrice: UFix64, customID: String?, commissionAmount: UFix64, commissionReceivers: [Address]?, expiry: UInt64)

Above event gets emitted when a listing has been created and added to a Storefront resource. The Address values here are valid when the event is emitted, but the state of the accounts they refer to may change outside of the NFTStorefrontV2 workflow, so be careful to check when using them.

event ListingCompleted

cadence

event ListingCompleted(listingResourceID: UInt64, storefrontResourceID: UInt64, purchased: Bool, nftType: Type, nftUUID: UInt64, nftID: UInt64, salePaymentVaultType: Type, salePrice: UFix64, customID: String?, commissionAmount: UFix64, commissionReceiver: Address?, expiry: UInt64)

The listing has been resolved. It has either been purchased, removed or destroyed.

event UnpaidReceiver

```
cadence
event UnpaidReceiver(receiver: Address, entitledSaleCut: UFix64)
```

A entitled receiver has not been paid during the sale of the NFT.

Holistic process flow diagram of NFTStorefrontV2 -

!NFT Storefront Process flow

<!-- Relative-style links. Does not render on the page -->

```
[NFTStorefrontV2.cdc]: https://github.com/onflow/nft-
storefront/blob/main/contracts/NFTStorefrontV2.cdc
[NFTStorefront.cdc]: https://github.com/onflow/nft-
storefront/blob/main/contracts/NFTStorefront.cdc
[NonFungibleToken]: https://github.com/onflow/flow-
nft/blob/master/contracts/NonFungibleToken.cdc
[receiver capabilities]: https://cadence-
lang.org/docs/language/capabilities
[Storefront resource]: #resource-storefront
[setupaccount]: https://github.com/onflow/nft-
storefront/blob/main/transactions/setupaccount.cdc
[sellitem]: https://github.com/onflow/nft-
storefront/blob/main/transactions/sellitem.cdc
[Capability]: https://cadence-lang.org/docs/language/capabilities
[createListing]: #fun-createlisting
[Fungible Token Switchboard]: https://github.com/onflow/flow-ft#fungible-
token-switchboard
[commission]: #enabling-marketplace-commissions-for-nft-sales
[sellitemwithmarketplacecut]: https://github.com/onflow/nft-
storefront/blob/main/transactions/sellitemwithmarketplacecut.cdc
[saleCut]: https://github.com/onflow/nft-
storefront/blob/160e97aa802405ad26a3164bc0fde7ee52ad2/contracts/NFTSto
refrontV2.cdc#L104
[purchase]: #fun-purchase
[FungibleTokenSwitchboard]: https://github.com/onflow/flow-
ft/blob/master/contracts/FungibleTokenSwitchboard.cdc
[Royalty Metadata View]: https://github.com/onflow/flow-
nft/blob/21c254438910c8a4b5843beda3df20e4e2559625/contracts/MetadataViews
.cdc#L335
```

11-staking-collection.md:

```
title: Flow Staking Collection Contract Reference
sidebarposition: 11
sidebarlabel: Staking Collection
---
```

Contract

The FlowStakingCollection contract is a contract that manages a resource containing a user's stake and delegation objects.

The FlowStakingCollection allows a user to manage multiple active nodes or delegators and interact with node or delegator objects stored in either their optional locked account or in the StakingCollection itself (stored in the main account). If a user has locked tokens, StakingCollection allows a user to interact with their locked tokens to perform staking actions for any of their nodes or delegators.

The staking collection also manages creating a node's machine accounts if they have any collector or consensus nodes. It also allows them to deposit and withdraw tokens from any of their machine accounts through the staking collection.

See the Staking Collection Docs for more information on the design of the staking collection contract.

Source: FlowStakingCollection.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x95e019a17d0e23d7
Mainnet	0x8d0e87b65159ae63

Transactions

Use the following transactions to interact with the StakingCollection.

\Note: The StakingCollection differentiates between stake and delegation requests through passing an optional DelegatorID argument. For example, if you wish to Stake New Tokens for an active node, pass nil as the optional DelegatorID argument to the Stake New Tokens transaction. The same applies for all the other staking operation transactions.

ID	Name	Source
SCO.01	Setup Staking Collection	stakingCollection/setupstakingcollection.cdc
SCO.02	Register Delegator	stakingCollection/registerdelegator.cdc
SCO.03	Register Node	stakingCollection/registernode.cdc

SCO.04 Create Machine Account	
stakingCollection/createmachineaccount.cdc	
SCO.05 Request Unstaking	
stakingCollection/requestunstaking.cdc	
SCO.06 Stake New Tokens	
stakingCollection/stakenewtokens.cdc	
SCO.07 Stake Rewarded Tokens	
stakingCollection/stakerewardedtokens.cdc	
SCO.08 Stake Unstaked Tokens	
stakingCollection/stakeunstakedtokens.cdc	
SCO.09 Unstake All	
stakingCollection/unstakeall.cdc	
SCO.10 Withdraw Rewarded Tokens	
stakingCollection/withdrawrewardedtokens.cdc	
SCO.11 Withdraw Unstaked Tokens	
stakingCollection/withdrawunstakedtokens.cdc	
SCO.12 Close Stake	
stakingCollection/closestake.cdc	
SCO.13 Transfer Node	
stakingCollection/transfernode.cdc	
SCO.14 Transfer Delegator	
stakingCollection/transferdelegator.cdc	
SCO.15 Withdraw From Machine Account	
stakingCollection/withdrawfrommachineaccount.cdc	
SCO.22 Update Networking Address	
stakingCollection/updatenetworkingaddress.cdc	

Scripts

ID	Name	Source
----- ----- -----		

SCO.16 Get All Delegator Info		
stakingCollection/scripts/getalldelegatorinfo.cdc		
SCO.15 Get All Node Info		
stakingCollection/scripts/getallnodeinfo.cdc		
SCO.22 Get Delegator Ids		
stakingCollection/scripts/getdelegatorids.cdc		
SCO.17 Get Node Ids		
stakingCollection/scripts/getnodeids.cdc		
SCO.18 Get Does Stake Exist		
stakingCollection/scripts/getdoesstakeexist.cdc		
SCO.19 Get Locked Tokens Used		
stakingCollection/scripts/getlockedtokensused.cdc		
SCO.20 Get Unlocked Tokens Used		
stakingCollection/scripts/getunlockedtokensused.cdc		
SCO.21 Get Machine Accounts		
stakingCollection/scripts/getmachineaccounts.cdc		

Setup Transaction

To setup the Staking Collection for an account, use the SC.01 transaction.

The setup process finds any node or delegator records already stored in the main account's storage, as well as any in the associated locked account if an associated locked account exists. It connects these node and delegator records with the new Staking Collection, allowing them to be interacted with using the Staking Collection API.

Events

The StakingCollection contract emits an event whenever an important action occurs.

```
cadence
    access(all) event NodeAddedToStakingCollection(nodeID: String, role: UInt8, amountCommitted: UFix64, address: Address?)
    access(all) event DelegatorAddedToStakingCollection(nodeID: String, delegatorID: UInt32, amountCommitted: UFix64, address: Address?)

    access(all) event NodeRemovedFromStakingCollection(nodeID: String, role: UInt8, address: Address?)
    access(all) event DelegatorRemovedFromStakingCollection(nodeID: String, delegatorID: UInt32, address: Address?)

    access(all) event MachineAccountCreated(nodeID: String, role: UInt8, address: Address)
```

12-hybrid-custody.md:

```
---
title: Flow Account Linking Contract Address
sidebarposition: 12
sidebarlabel: Account Linking
---
```

Contract

The Account Linking contracts manage ChildAccounts to permit hybrid custody in scenarios where apps only want to share a subset of resources on their accounts with various parents. In many cases, this will be a user's primary wallet outside of the application a child account came from.

You can see the docs for account linking here

Network Contract Address
----- -----

```
| Testnet | 0x294e44e1ec6993c6 |
| Mainnet | 0xd8a7e05a7ac670c0 |
```

13-evm.md:

```
---
title: Flow EVM
sidebarposition: 13
sidebarlabel: EVM
---
```

Contract

The EVM contract is the entrypoint from Cadence to EVM on Flow. While many developers may choose to interact with EVM via EVM-equivalent tooling paths, all access to Flow EVM ultimately interfaces via Cadence at some level.

If you would like to interact with EVM directly from Cadence, you can use the EVM contract and its constructs. Read more about the EVM contract and its role in Flow's EVM equivalence in FLIP #223.

Mainnet/Testnet Source: EVM.cdc

Network	Contract Address
Emulator	0xf8d6e0586b0a20c7
Cadence Testing Framework	0x0000000000000001
Testnet	0x8c5303eaa26202d6
Mainnet	0xe467b9dd11fa00df

14-burner.md:

```
---
title: Flow Burner Contract Address
sidebarposition: 14
sidebarlabel: Burner
---
```

Contract

The Burner contract provides a way for resources to define custom logic that is executed when the resource is destroyed. Resources that want to utilize this functionality should implement the Burner.Burnable interface which requires that they include

a `burnCallback()` function that includes the custom logic.
It is recommended that regardless of the resource, all users and
developers
should use `Burner.burn()` when destroying a resource instead of `destroy`.

```
| Network | Contract Address
|
| ----- | -----
----- |
| Cadence Testing Framework | 0x0000000000000001 |
| Emulator | 0xee82856bf20e2aa6 |
| Testnet | 0x294e44e1ec6993c6 |
| Mainnet | 0xd8a7e05a7ac670c0 |

# index.md:

---
title: Flow Core Contracts
description: The smart contracts that power the Flow protocol
sidebarlabel: Core Smart Contracts
sidebarposition: 9
sidebarcustomprops:
  icon: 🛡️
  description: Explore the foundational contracts driving the Flow
  blockchain and learn how to utilize these vital building blocks for your
  own smart contract development.
---

Flow relies on a set of core contracts that define key portions of the
Flow protocol.
```

These contracts control the following:

- Standard fungible token behavior. (`FungibleToken`,
`FungibleTokenMetadataViews`, `FungibleTokenSwitchboard`, `Burner`)
- Flow Protocol Token. (`FlowToken`)
- Flow Service Account. (`ServiceAccount`, `NodeVersionBeacon`,
`RandomBeaconHistory`)
- Account, transaction and storage fee payments. (`FlowFees` and
`FlowStorageFees`)
- Staking and delegation (`FlowIDTableStaking`)
- Epochs (`FlowEpoch`, `FlowClusterQC`, `FlowDKG`)

There are other important contracts that aren't part of the core protocol
but are nevertheless important to developers on Flow:

- Standard Non-Fungible Token Behavior. (`NonFungibleToken`)
- NFT Metadata Standard. (`MetadataViews`, `ViewResolver`)
- Staking Collection. (`StakingCollection`)
- NFT Storefronts. (`NFTStorefront`)
- Account linking and Hybrid Custody. (`AccountLinking`)
- EVM interfacing contract. (`EVM`)

```
# index.md:

---
sidebarposition: 2
title: Differences vs. EVM
sidebarcustomprops:
  icon: ↵
---

Flow [Cadence] is designed with many improvements over prior blockchain networks. As a result, you'll notice many differences between Flow vs. other blockchains, especially Ethereum. This document will be most useful to developers who are already familiar with building on the EVM, but contains details useful to all developers. Check out [Why Flow] for a more general overview of the Flow blockchain.
```

:::tip

Remember, Flow also supports full [EVM] equivalence! You can start by moving over your existing contracts, then start building new features that take advantage of the power of Cadence.

:::

The Flow Cadence Account Model

Key pairs establish ownership on blockchains. In other blockchains (e.g. Bitcoin and Ethereum), the user's address is also calculated based on their public key, making a unique one-to-one relationship between accounts (addresses) and public keys. This also means there is no concrete "account creation" process other than generating a valid key pair.

With the advent of smart contracts, Ethereum introduced a new account type for deploying contracts that can use storage space (i.e., to store contract bytecode). You can learn more about the distinction between EOA and Contract [accounts on Ethereum].

The [Flow account model] combines the concepts of EOAs and Contract Accounts into a single account model and decouples accounts and public keys. Flow accounts are associated with one or more public keys of varying weights that specify interested parties that need to produce valid cryptographic signatures for each transaction authorized by that account.

!Screenshot 2023-08-16 at 16.43.07.png

This natively enables interesting use cases, like key revocation, rotation, and multi-signature transactions. All Flow accounts can use network storage (e.g., for deploying contracts and storing resources like NFTs) based on the number of FLOW tokens they hold.

:::warning

You must run an explicit account creation transaction on Flow to create a new account. [Flow CLI] can create an account on any network with a given public key. Doing so requires a [very small fee] to be paid in FLOW.

:::

Another key difference is that [storage] for data and assets related to an account are stored in the account, not in the contract as with the EVM.

Check out the [Accounts] concept document to learn more about Flow accounts.

Smart Contracts

On Flow, smart contracts can be written in [Cadence], or Solidity. Cadence syntax is user-friendly and inspired by modern languages like Swift. Notable features of Cadence that make it unique and the key power of the Flow blockchain are:

- Resource-oriented: Cadence introduces a new type called Resources. Resources enable onchain representation of digital assets natively and securely. Resources can only exist in one location at a time and are strictly controlled by the execution environment to avoid common mishandling mistakes. Each resource has a unique uuid associated with it on the blockchain. Examples of usage are fungible tokens, NFTs, or any custom data structure representing a real-world asset. Check out [Resources] to learn more.
- Capability-based: Cadence offers a [Capability-based Security] model. This also enables the use of Resources as structures to build access control. Capabilities and [Entitlements] can provide fine-grained access to the underlying objects for better security. For example, when users list an NFT on a Flow marketplace, they create a new Capability to the stored NFT in their account so the buyer can withdraw the asset when they provide the tokens. Check out [Capability-based Access Control] to learn more about Capabilities on Cadence.

:::warning

Cadence is not compiled. All contracts are public and unobfuscated on Flow. This isn't that different from the EVM, where it's trivial to decompile a contract back into Solidity.

:::

Check out the [Cadence] website to learn the details of the Cadence programming language.

If you are a Solidity developer, we recommend you start with Cadence's [Guide for Solidity Developers] to dive deeper into the differences between the two languages.

Here are some additional resources that can help you get started with Cadence:

- [The Cadence tutorial]
- ERC-20 equivalent on Flow is the Flow Fungible Token Standard
 - Repository
 - Tutorial
- ERC-721 equivalent on Flow is the Flow Non-Fungible Token Standard
 - Repository
 - Tutorial
- Asset marketplaces with Cadence
 - Tutorial
 - NFT Storefront is an example marketplace standard

Transactions and Scripts

You can interact with the state on most other blockchains by cryptographically authorizing smart contract function calls. On Flow, transactions offer rich functionality through Cadence code. This allows you to seamlessly combine multiple contracts and function calls into a single transaction that updates the blockchain state - all executing together as one unified operation.

Here is a sample transaction that mints an NFT from ExampleNFT contract on Testnet:

```
cadence
import NonFungibleToken from 0x631e88ae7f1d7c20
import ExampleNFT from 0x2bd9d8989a3352a1

/// Mints a new ExampleNFT into recipient's account

transaction(recipient: Address) {

    /// Reference to the receiver's collection
    let recipientCollectionRef: &{NonFungibleToken.Collection}

    /// Previous NFT ID before the transaction executes
    let mintingIDBefore: UInt64

    prepare(signer: &Account) {

        self.mintingIDBefore = ExampleNFT.totalSupply

        // Borrow the recipient's public NFT collection reference
        self.recipientCollectionRef = getAccount(recipient)

        .capabilities.get<&{NonFungibleToken.Collection}>(ExampleNFT.CollectionPublicPath)
            .borrow()
            ?? panic("The recipient does not have a NonFungibleToken
Receiver at "
                    .concat(ExampleNFT.CollectionPublicPath.toString())
                    .concat(" that is capable of receiving an NFT."))

    }
}
```

```

        .concat("The recipient must initialize their account
with this collection and receiver first!"))

    }

execute {

    let currentIDString = self.mintingIDBefore.toString()

    // Mint the NFT and deposit it to the recipient's collection
    ExampleNFT.mintNFT(
        recipient: self.recipientCollectionRef,
        name: "Example NFT #".concat(currentIDString),
        description: "Example description for
#".concat(currentIDString),
        thumbnail: "https://robohash.org/".concat(currentIDString),
        royalties: []
    )
}

post {

    self.recipientCollectionRef.getIDs().contains(self.mintingIDBefore): "The
next NFT ID should have been minted and delivered"
    ExampleNFT.totalSupply == self.mintingIDBefore + 1: "The total
supply should have been increased by 1"
}
}
```

Authorizing Transactions

The process to authorize a transaction on Flow Cadence is more complex, but also much more powerful than an EVM transaction:

- [Accounts] can have multiple keys with varying weights
- Multiple accounts can sign a single transaction (prepare takes any number of arguments)
- Transaction computation fees can be paid by a different account, called the Payer account.
- The [transaction nonce] is provided by the Proposer account. This enables rate control and order to be dictated by a different party if needed.
- All of the above roles can be the same account.

The same powerful concept also exists for querying the blockchain state using Scripts. Here is a sample script that fetches the ExampleNFT IDs owned by a given account on Testnet:

```

cadence
/// Script to get NFT IDs in an account's collection

import NonFungibleToken from 0x631e88ae7f1d7c20
import ExampleNFT from 0x2bd9d8989a3352a1
```

```

access(all) fun main(address: Address, collectionPublicPath: PublicPath):
[UInt64] {

    let account = getAccount(address)

    let collectionRef = account

    .capabilities.get<&{NonFungibleToken.Collection}>(collectionPublicPath)
    .borrow()
        ?? panic("The account with address "
            .concat(address.toString())
            .concat("does not have a NonFungibleToken Collection
at "
            .concat(ExampleNFT.CollectionPublicPath.toString())
            .concat("."). The account must initialize their account
with this collection first!"))

    return collectionRef.getIDs()

}

```

Check out [Transactions] and [Scripts] to learn more about the concepts. You can also read the Cadence language reference on [Transactions] to dive deeper.

Flow Nodes

Developers need a blockchain node to send transactions and fetch state. Flow is based on a multi-node architecture that separates tasks like consensus and computation into separate nodes. You can learn more about the Flow architecture in the [Flow Primer].

Access Nodes are the node type that are most useful for developers, as they provide access to the Flow network [via an API].

SDKs and Tools

If you're already familiar with blockchain development, here's a comparison between popular software packages and Flow's tooling:

- hardhat / Truffle / Foundry
 - Flow CLI provides local development tools and the Flow Emulator
- OpenZeppelin
 - Emerald OZ
- go-ethereum
 - Flow Go SDK
 - FCL also provides Backend API for Flow in JS
- web3.js
 - FCL
 - flow-cadut provides more utilities for using Flow on Web
- Remix
 - Flow Playground provides basic experimentation on the web

- Cadence VSCode Extension is strongly suggested to install for local development
- Testing Smart Contracts
 - Cadence testing framework enables native tests in Cadence.
 - overflow for testing in Go.

<!-- Relative-style links. Does not render on the page -->

```
[Why Flow]: ../flow.md
[EVM]: ../../evm/about.md
[accounts on Ethereum]: https://ethereum.org/en/developers/docs/accounts
[Flow CLI]: ../../tools/flow-cli/accounts/create-accounts.md
[very small fee]: ../basics/fees.md#fee-structure
[Flow account model]: ../basics/accounts.md
[Accounts]: ../basics/accounts.md
[storage]: ../basics/accounts.md#storage
[Cadence]: https://cadence-lang.org/
[Resources]: https://cadence-lang.org/docs/language/resources
[Capability-based Security]: https://en.wikipedia.org/wiki/Capability-based_security
[Entitlements]: https://cadence-lang.org/docs/language/access-control#entitlements
[Capability-based Access Control]: https://cadence-lang.org/docs/language/capabilities
[Guide for Solidity Developers]: https://cadence-lang.org/docs/solidity-to-cadence
[The Cadence tutorial]: https://cadence-lang.org/docs/tutorial/first-steps
[transaction nonce]:
https://ethereum.org/en/developers/docs/accounts/#an-account-examined
[Transactions]: ../basics/transactions.md
[Scripts]: ../basics/scripts.md
[Transactions]: https://cadence-lang.org/docs/language/transactions
[Flow Primer]: https://flow.com/primer#primer-how-flow-works
[via an API]: ../../networks/flow-networks/index.md
```

fcl-quickstart.md:

```
---
sidebarposition: 3
sidebarlabel: Simple Frontend
---
```

Simple Frontend

Building upon the Counter contract you interacted with in Step 1: Contract Interaction and deployed locally in Step 2: Local Development, this tutorial will guide you through creating a simple frontend application using [Next.js] to interact with the Counter smart contract on the local Flow emulator. Using the [Flow Client Library] (FCL), you'll learn how to read and modify the contract's state from a React web application, set up wallet authentication using FCL's Discovery UI

connected to the local emulator, and query the chain to read data from smart contracts.

Objectives

After completing this guide, you'll be able to:

- Display data from a [Cadence] smart contract (Counter) on a Next.js frontend using the [Flow Client Library].
- Query the chain to read data from smart contracts on the local emulator.
- Mutate the state of a smart contract by sending transactions using FCL and a wallet connected to the local emulator.
- Set up the Discovery UI to use a wallet for authentication with the local emulator.

Prerequisites

- Completion of Step 1: Contract Interaction and Step 2: Local Development.
- Flow CLI installed.
- Node.js and npm installed.

Setting Up the Next.js App

Assuming you're in your project directory from Steps 1 and 2, we'll create a Next.js frontend application to interact with your smart contract deployed on the local Flow emulator.

Step 1: Create a New Next.js App

First, we'll create a new Next.js application using `npx create-next-app`. We'll create it inside your existing project directory and then move it up to the root directory.

Assumption: You are already in your project directory.

Run the following command:

```
bash
npx create-next-app@latest fcl-app-quickstart
```

During the setup process, you'll be prompted with several options. Choose the following:

- TypeScript: No
- Use src directory: Yes
- Use App Router: Yes

This command will create a new Next.js project named `fcl-app-quickstart` inside your current directory.

Step 2: Move the Next.js App Up a Directory

Now, we'll move the contents of the fcl-app-quickstart directory up to your project root directory.

Note: Moving the Next.js app into your existing project may overwrite existing files such as package.json, package-lock.json, .gitignore, etc. Make sure to back up any important files before proceeding. You may need to merge configurations manually.

Remove the README File

Before moving the files, let's remove the README.md file from the fcl-app-quickstart directory to avoid conflicts:

```
bash  
rm fcl-app-quickstart/README.md
```

Merge .gitignore Files and Move Contents

To merge the .gitignore files, you can use the cat command to concatenate them and then remove duplicates:

```
bash  
cat .gitignore fcl-app-quickstart/.gitignore | sort | uniq >  
tempgitignore  
mv tempgitignore .gitignore
```

Now, move the contents of the fcl-app-quickstart directory to your project root:

On macOS/Linux:

```
bash  
mv fcl-app-quickstart/ .  
mv fcl-app-quickstart/. . This moves hidden files like .env.local if any  
rm -r fcl-app-quickstart
```

On Windows (PowerShell):

```
powershell  
Move-Item -Path .\fcl-app-quickstart\ -Destination . -Force  
Move-Item -Path .\fcl-app-quickstart\. -Destination . -Force  
Remove-Item -Recurse -Force .\fcl-app-quickstart
```

Note: When moving hidden files (those starting with a dot, like .gitignore), ensure you don't overwrite important files in your root directory.

Step 3: Install FCL

Now, install the Flow Client Library (FCL) in your project. FCL is a JavaScript library that simplifies interaction with the Flow blockchain:

```
bash
npm install @onflow/fcl
```

Setting Up the Local Flow Emulator and Dev Wallet

Before proceeding, ensure that both the Flow emulator and the Dev Wallet are running.

Step 1: Start the Flow Emulator

In a new terminal window, navigate to your project directory and run:

```
bash
flow emulator start
```

This starts the Flow emulator on `http://localhost:8888`.

Step 2: Start the FCL Dev Wallet

In another terminal window, run:

```
bash
flow dev-wallet
```

This starts the Dev Wallet, which listens on `http://localhost:8701`. The Dev Wallet is a local wallet that allows you to authenticate with the Flow blockchain and sign transactions on the local emulator. This is the wallet we'll select in Discovery UI when authenticating.

Querying the Chain

Now, let's read data from the Counter smart contract deployed on the local Flow emulator.

Since you've already deployed the Counter contract in Step 2: Local Development, we can proceed to query it.

Step 1: Update the Home Page

Open `src/app/page.js` in your editor.

Adding the FCL Configuration Before the Rest

At the top of your `page.js` file, before the rest of the code, we'll add the FCL configuration. This ensures that FCL is properly configured before we use it.

Add the following code:

```

jsx
import  as fcl from "@onflow/fcl";

// FCL Configuration
fcl.config({
  "flow.network": "local",
  "accessNode.api": "http://localhost:8888", // Flow Emulator
  "discovery.wallet": "http://localhost:8701/fcl/authn", // Local Wallet
Discovery
});

```

This configuration code sets up FCL to work with the local Flow emulator and Dev Wallet. The flow.network and accessNode.api properties point to the local emulator, while discovery.wallet points to the local Dev Wallet for authentication.

For more information on Discovery configurations, refer to the [Wallet Discovery Guide](#).

Implementing the Component

Now, we'll implement the component to query the count from the Counter contract.

Update your page.js file to the following:

```

jsx
// src/app/page.js

"use client"; // This directive is necessary when using useState and
useEffect in Next.js App Router

import { useState, useEffect } from "react";
import  as fcl from "@onflow/fcl";

// FCL Configuration
fcl.config({
  "flow.network": "local",
  "accessNode.api": "http://localhost:8888",
  "discovery.wallet": "http://localhost:8701/fcl/authn", // Local Dev
Wallet
});

export default function Home() {
  const [count, setCount] = useState(0);

  const queryCount = async () => {
    try {
      const res = await fcl.query({
        cadence:
          import Counter from 0xf8d6e0586b0a20c7
          import NumberFormatter from 0xf8d6e0586b0a20c7
    }
  }
}

```

```

access(all)
fun main(): String {
    // Retrieve the count from the Counter contract
    let count: Int = Counter.getCount()

    // Format the count using NumberFormatter
    let formattedCount =
        NumberFormatter.formatWithCommas(number: count)

    // Return the formatted count
    return formattedCount
}

),
setCount(res);
} catch (error) {
    console.error("Error querying count:", error);
}
};

useEffect(() => {
    queryCount();
}, []);

return (
<div>
    <h1>FCL App Quickstart</h1>
    <div>Count: {count}</div>
</div>
);
}

```

In the above code:

- We import the necessary React hooks (`useState` and `useEffect`) and the FCL library.
- We define the `Home` component, which is the main page of our app.
- We set up a state variable `count` using the `useState` hook to store the count value.
- We define an async function `queryCount` to query the count from the Counter contract.
- We use the `useEffect` hook to call `queryCount` when the component mounts.
- We return a simple JSX structure that displays the count value on the page.
- If an error occurs during the query, we log it to the console.
- We use the script from Step 2 to query the count from the Counter contract and format it using the `NumberFormatter` contract.

Step 2: Run the App

Start your development server:

```
bash
npm run dev
```

Visit `http://localhost:3000` in your browser. You should see the current count displayed on the page, formatted according to the `NumberFormatter` contract.

Mutating the Chain State

Now that we've successfully read data from the Flow blockchain emulator, let's modify the state by incrementing the count in the Counter contract. We'll set up wallet authentication and send a transaction to the blockchain emulator.

Adding Authentication and Transaction Functionality

Step 1: Manage Authentication State

In `src/app/page.js`, add new state variables to manage the user's authentication state:

```
jsx
const [user, setUser] = useState({ loggedIn: false });
```

Step 2: Subscribe to Authentication Changes

Update the `useEffect` hook to subscribe to the current user's authentication state:

```
jsx
useEffect(() => {
  fcl.currentUser.subscribe(setUser);
  queryCount();
}, []);
```

The `currentUser.subscribe` method listens for changes to the current user's authentication state and updates the user state accordingly.

Step 3: Define Log In and Log Out Functions

Define the `logIn` and `logOut` functions:

```
jsx
const logIn = () => {
  fcl.authenticate();
};

const logOut = () => {
  fcl.unauthenticate();
};
```

The authenticate method opens the Discovery UI for the user to log in, while unauthenticate logs the user out.

Step 4: Define the incrementCount Function

Add the incrementCount function:

```
jsx
const incrementCount = async () => {
  try {
    const transactionId = await fcl.mutate({
      cadence:
        import Counter from 0xf8d6e0586b0a20c7

      transaction {

        prepare(acct: &Account) {
          // Authorizes the transaction
        }

        execute {
          // Increment the counter
          Counter.increment()

          // Retrieve the new count and log it
          let newCount = Counter.getCount()
          log("New count after incrementing:
".concat(newCount.toString()))
        }
      }
    ,
    proposer: fcl.currentUser,
    payer: fcl.currentUser,
    authorizations: [fcl.currentUser.authorization],
    limit: 50,
  });

  console.log("Transaction Id", transactionId);

  await fcl.tx(transactionId).onceSealed();
  console.log("Transaction Sealed");

  queryCount();
} catch (error) {
  console.error("Transaction Failed", error);
}
};
```

In the above code:

- We define an async function incrementCount to send a transaction to increment the count in the Counter contract.

- We use the `mutate` method to send a transaction to the blockchain emulator.
- The transaction increments the count in the Counter contract and logs the new count.
- We use the `proposer`, `payer`, and `authorizations` properties to set the transaction's `proposer`, `payer`, and `authorizations` to the current user.
- The `limit` property sets the gas limit for the transaction.
- We log the transaction ID and wait for the transaction to be sealed before querying the updated count.
- If an error occurs during the transaction, we log it to the console.
- After the transaction is sealed, we call `queryCount` to fetch and display the updated count.
- We use the transaction from Step 2 to increment the count in the Counter contract.

Step 5: Update the Return Statement

Update the return statement to include authentication buttons and display the user's address when they're logged in:

```
jsx
return (
  <div>
    <h1>FCL App Quickstart</h1>
    <div>Count: {count}</div>
    {user.loggedIn ? (
      <div>
        <p>Address: {user.addr}</p>
        <button onClick={logOut}>Log Out</button>
        <div>
          <button onClick={incrementCount}>Increment Count</button>
        </div>
      </div>
    ) : (
      <button onClick={logIn}>Log In</button>
    )
  </div>
);
```

Full page.js Code

Your `src/app/page.js` should now look like this:

```
jsx
// src/app/page.js

"use client";

import { useState, useEffect } from "react";
import as fcl from "@onflow/fcl";

// FCL Configuration
fcl.config({
```

```

"flow.network": "local",
"accessNode.api": "http://localhost:8888",
"discovery.wallet": "http://localhost:8701/fcl/authn", // Local Dev
Wallet
});

export default function Home() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState({ loggedIn: false });

  const queryCount = async () => {
    try {
      const res = await fcl.query({
        cadence:
          import Counter from 0xf8d6e0586b0a20c7
          import NumberFormatter from 0xf8d6e0586b0a20c7

        access(all)
        fun main(): String {
          // Retrieve the count from the Counter contract
          let count: Int = Counter.getCount()

          // Format the count using NumberFormatter
          let formattedCount =
            NumberFormatter.formatWithCommas(number: count)

          // Return the formatted count
          return formattedCount
        }
      });
      setCount(res);
    } catch (error) {
      console.error("Error querying count:", error);
    }
  };

  useEffect(() => {
    fcl.currentUser.subscribe(setUser);
    queryCount();
  }, []);

  const logIn = () => {
    fcl.authenticate();
  };

  const logOut = () => {
    fcl.unauthenticate();
  };

  const incrementCount = async () => {
    try {
      const transactionId = await fcl.mutate({
        cadence:

```

```

import Counter from 0xf8d6e0586b0a20c7

transaction {

    prepare(acct: &Account) {
        // Authorizes the transaction
    }

    execute {
        // Increment the counter
        Counter.increment()

        // Retrieve the new count and log it
        let newCount = Counter.getCount()
        log("New count after incrementing:
".concat(newCount.toString()))
    }
}

,
proposer: fcl.currentUser,
payer: fcl.currentUser,
authorizations: [fcl.currentUser.authorization],
limit: 50,
});

console.log("Transaction Id", transactionId);

await fcl.tx(transactionId).onceSealed();
console.log("Transaction Sealed");

queryCount();
} catch (error) {
    console.error("Transaction Failed", error);
}
};

return (
<div>
<h1>FCL App Quickstart</h1>
<div>Count: {count}</div>
{user.loggedIn ? (
<div>
<p>Address: {user.addr}</p>
<button onClick={logOut}>Log Out</button>
<div>
    <button onClick={incrementCount}>Increment Count</button>
</div>
</div>
) : (
<button onClick={logIn}>Log In</button>
)
}
</div>
);
}
}

```

Visit <http://localhost:3000> in your browser.

- Log In:
 - Click the "Log In" button.
 - The Discovery UI will appear, showing the available wallets. Select the "Dev Wallet" option.
 - Select the account to log in with.
 - If prompted, create a new account or use an existing one.
- Increment Count:
 - After logging in, you'll see your account address displayed.
 - Click the "Increment Count" button.
 - Your wallet will prompt you to approve the transaction.
 - Approve the transaction to send it to the Flow emulator.
- View Updated Count:
 - Once the transaction is sealed, the app will automatically fetch and display the updated count.
 - You should see the count incremented on the page, formatted using the NumberFormatter contract.

Conclusion

By following these steps, you've successfully created a simple frontend application using Next.js that interacts with the Counter smart contract on the Flow blockchain emulator. You've learned how to:

- Add the FCL configuration before the rest of your code within the page.js file.
- Configure FCL to work with the local Flow emulator and Dev Wallet.
- Start the Dev Wallet using flow dev-wallet to enable local authentication.
- Read data from the local blockchain emulator, utilizing multiple contracts (Counter and NumberFormatter).
- Authenticate users using the local Dev Wallet.
- Send transactions to mutate the state of a smart contract on the local emulator.

Additional Resources

[Flow Client Library]: <https://github.com/onflow/fcl-js>
[Cadence]: <https://developers.flow.com/cadence>
[Next.js]: <https://nextjs.org/docs/getting-started>
[Flow Emulator]: <https://developers.flow.com/tools/emulator>
[Flow Dev Wallet]: <https://github.com/onflow/fcl-dev-wallet>

```
# flow-cli.md:  
  
---  
sidebarposition: 2  
sidebarlabel: Local Development  
---
```

Local Development

The [Flow Command Line Interface] (CLI) is a set of tools that developers can use to interact with the Flow blockchain by managing accounts, sending transactions, deploying smart contracts, running the emulator, and more. This quickstart will get you familiar with its main concepts and functionality.

Objectives

After completing this guide, you'll be able to:

- Create a Flow project using the [Flow Command Line Interface]
- Run tests for a smart contract
- Add an already-deployed contract to your project with the [Dependency Manager]
- Deploy a smart contract locally to the Flow Emulator
- Write and execute scripts to interact with a deployed smart contract

Installation

The first thing you'll need to do is install the Flow CLI. If you have [homebrew] installed you can run:

```
zsh
brew install flow-cli
```

For other ways of installing, please refer to the [installation guide].

Creating a New Project

To create a new project, navigate to the directory where you want to create your project and run:

```
zsh
flow init
```

Upon running this command, you'll be prompted to enter a project name. Enter a name and press Enter.

You'll also be asked if you'd like to install any core contracts (such as FungibleToken, NonFungibleToken, etc.) using the Dependency Manager. For this tutorial, you can select No.

The init command will create a new directory with the project name and the following files:

- flow.json: This file contains the configuration for your project.
- emulator-account.pkey: This file contains the private key for the default emulator account.
- flow.json: This file contains the configuration for your project.

- cadence/: This directory contains your Cadence code. Inside there are subdirectories for contracts, scripts, transactions, and tests.

Inside the cadence/contracts directory, you'll find a Counter.cdc file. This is the same as the Counter contract in the previous step.

Next, cd into your new project directory.

Running the Tests

To run the example test for the Counter contract located in cadence/tests, you can run:

```
zsh  
flow test
```

```
:::info
```

For additional details on how flow.json is configured, review the [configuration docs].

```
:::
```

Deploying the Contract to Emulator

The emulator is a local version of the Flow blockchain that you can use to test your contracts and scripts. It's a great way to develop and test your contracts locally - before you try them on the testnet or mainnet.

Before we deploy, let's open a new terminal window and run the emulator. From the root of your project directory, where your emulator-account.pkey and flow.json files are located, run:

```
zsh  
flow emulator start
```

Your emulator should now be running.

Deploying a Contract

Creating an Account

When you created a project you'll see that a Counter contract was added to your flow.json configuration file, but it's not set up for deployment yet. We could deploy it to the emulator-account, but for this example lets also create a new account on the emulator to deploy it to.

With your emulator running, run the following command:

```
zsh  
flow accounts create
```

When prompted, give your account the name test-account and select Emulator as the network. You'll now see this account in your flow.json.

> Note: We won't use this much in this example, but it's good to know how to create an account.

Configuring the Deployment

To deploy the Counter contract to the emulator, you'll need to add it to your project configuration. You can do this by running:

```
zsh
flow config add deployment
```

You'll be prompted to select the contract you want to deploy. Select Counter and then select the account you want to deploy it to. For this example, select emulator-account.

Deploying the Contract

To deploy the Counter contract to the emulator, run:

```
zsh
flow project deploy
```

That's it! You've just deployed your first contract to the Flow Emulator.

Running Scripts

Scripts are used to read data from the Flow blockchain. There is no state modification. In our case, we are going to read a greeting from the HelloWorld contract.

If we wanted to generate a new script, we could run:

```
zsh
flow generate script ScriptName
```

But the default project already has a GetCounter script for reading the count of the Counter contract. Open cadence/scripts/GetCounter.cdc in your editor to see the script.

To run the script, you can run:

```
zsh
flow scripts execute cadence/scripts/GetCounter.cdc
```

You should see zero as the result since the Counter contract initializes the count to zero and we haven't run any transactions to increment it.

:::tip

If you'll like to learn more about writing scripts, please check out the docs for [basic scripts].

:::

Executing Transactions

Transactions are used to modify the state of the blockchain. In our case, we want to increment the count of the Counter contract. Luckily, we already have a transaction for that in the project that was generated for us. Open cadence/transactions/IncrementCounter.cdc in your editor to see the transaction.

To run the transaction, you can run:

```
zsh
flow transactions send cadence/transactions/IncrementCounter.cdc
```

By default, this uses the emulator-account to sign the transaction and the emulator network. If you want to use your test-account account, you can specify the --signer flag with the account name.

:::tip

If you want to learn more about writing transactions, please read the docs for [basic transactions].

:::

Installing & Interacting With External Dependencies

In addition to creating your own contracts, you can also install contracts that have already been deployed to the network by using the [Dependency Manager]. This is useful for interacting with contracts that are part of the Flow ecosystem or that have been deployed by other developers.

For example, let's say we want to format the result of our GetCounter script so that we display the number with commas if it's greater than 999. To do that we can install a contract called NumberFormatter from testnet that has a function to format numbers.

To grab it, run:

```
zsh
flow dependencies add testnet://8a4dce54554b225d.NumberFormatter
```

When prompted for the account to deploy the contract to, select any account and ignore the prompt for an alias. This is if you wanted to configure a mainnet address for the contract.

This will add the NumberFormatter contract and any of its dependencies to an imports directory in your project. It will also add any dependencies to your flow.json file. In addition, the prompt will configure the deployment of the contract to the account you selected. Make sure to select the emulator-account account to deploy the contract to the emulator.

You should then see the NumberFormatter in your deployments for emulator in your flow.json. If you messed this up, you can always run flow config add deployment to add the contract to your deployments.

Now we can deploy the NumberFormatter contract to the emulator by running:

```
zsh
flow project deploy
```

Now that we have the NumberFormatter contract deployed, we can update our GetCounter script to format the result. Open cadence/scripts/GetCounter.cdc and update it to use the following code:

```
cadence
import "Counter"
import "NumberFormatter"

access(all)
fun main(): String {
    // Retrieve the count from the Counter contract
    let count: Int = Counter.getCount()

    // Format the count using NumberFormatter
    let formattedCount = NumberFormatter.formatWithCommas(number: count)

    // Return the formatted count
    return formattedCount
}
```

The things to note here are:

- We import the NumberFormatter contract.
- We call the formatWithCommas function from the NumberFormatter contract to format the count.
- We return the formatted count as a String.

Now, to run the updated script, you can run:

```
zsh
flow scripts execute cadence/scripts/GetCounter.cdc
```

You should now see the result. You won't see the commas unless the number is greater than 999.

More

If you want to continue on generating your own contracts, you can also use the generate subcommand to create a new contract file. See more in the [generate documentation].

After that, it's easy to add your contract to your project configuration using the Flow CLI [config commands].

<!-- Relative-style links. Does not render on the page -->

```
[Flow Command Line Interface]: ../../tools/flow-cli/index.md
[Cadence]: https://cadence-lang.org/
[configuration docs]: ../../tools/flow-cli/flow.json/configuration.md
[homebrew]: https://brew.sh/
[installation guide]: ../../tools/flow-cli/install
[0xa1296b1e2e90ca5b]:
https://contractbrowser.com/A.9dca641e9a4b691b.HelloWorld
[Dependency Manager]: ../../tools/flow-cli/dependency-manager
[basic scripts]: ./basics/scripts.md
[basic transactions]: ./basics/transactions.md
[generate documentation]: ../../tools/flow-cli/boilerplate.md
[config commands]: ../../tools/flow-cli/flow.json/manage-configuration.md
```

hello-world.md:

```
---
sidebarposition: 1
sidebarlabel: Contract Interaction
---
```

```
import VerticalSplit from "./vertical-split.svg"
```

Contract Interaction

In this quickstart guide, you'll interact with your first smart contract on the Flow Testnet. Testnet is a public instance of the Flow blockchain designed for experimentation, where you can deploy and invoke smart contracts without incurring any real-world costs.

Smart contracts on Flow are permanent pieces of code that live on the blockchain. They allow you to encode business logic, define digital assets, and much more. By leveraging smart contracts, you can create decentralized applications (dApps) that are transparent, secure, and open to anyone.

Flow supports modern smart contracts written in [Cadence], a resource-oriented programming language designed specifically for smart contracts. Cadence focuses on safety and security, making it easier to write robust

contracts. Flow also supports traditional [EVM]-compatible smart contracts written in Solidity, allowing developers to port their existing Ethereum contracts to Flow. In this guide, we'll focus on interacting with Cadence smart contracts.

Objectives

After completing this guide, you'll be able to:

- Read data from a [Cadence] smart contract deployed on Flow.
- Understand how to interact with contracts on Flow's testnet.
- Retrieve and display data from a deployed smart contract via scripts.

In later steps, you'll learn how to:

- Create a Flow project using the Flow CLI.
- Add an already-deployed contract to your project with the Dependency Manager.
- Deploy a smart contract locally to the Flow Emulator.
- Write and execute transactions to interact with a deployed smart contract.
- Display data from a Cadence smart contract on a React frontend using the Flow Client Library.

Calling a Contract With a Script

The Counter contract exposes a public function named `getCount()` that returns the current value of the counter. We can retrieve its value using a simple script written in the [Cadence] programming language. Scripts in Cadence are read-only operations that allow you to query data from the blockchain without changing any state.

Here's the script:

```
cadence
import Counter from 0x8a4dce54554b225d

access(all)
fun main(): Int {
    return Counter.getCount()
}
```

Let's break down what this script does:

- Import Statement: `import Counter from 0x8a4dce54554b225d` tells the script to use the Counter contract deployed at the address `0x8a4dce54554b225d` on the testnet.
- Main Function: `access(all) fun main(): Int` defines the entry point of the script, which returns an Int.
- Return Statement: `return Counter.getCount()` calls the `getCount()` function from the Counter contract and returns its value.

Steps to Execute the Script

- Run the Script: Click the Run button to execute the script.
- View the Output: Observe the output returned by the script. You should see the current value of the count variable, which is 0 unless it has been modified.

```
<iframe sandbox className="flow-runner-iframe"
src="https://run.dnz.dev/snippet/a7a18e74d27f691a?colormode=dark&output=h
orizontal&outputSize=400" width="100%" height="400px"></iframe>
```

Understanding the Counter Contract

To fully grasp how the script works, it's important to understand the structure of the Counter contract. Below is the source code for the contract:

```
cadence
access(all) contract Counter {

    access(all) var count: Int

    // Event to be emitted when the counter is incremented
    access(all) event CounterIncremented(newCount: Int)

    // Event to be emitted when the counter is decremented
    access(all) event CounterDecrementedException(newCount: Int)

    init() {
        self.count = 0
    }

    // Public function to increment the counter
    access(all) fun increment() {
        self.count = self.count + 1
        emit CounterIncremented(newCount: self.count)
    }

    // Public function to decrement the counter
    access(all) fun decrement() {
        self.count = self.count - 1
        emit CounterDecrementedException(newCount: self.count)
    }

    // Public function to get the current count
    view access(all) fun getCount(): Int {
        return self.count
    }
}
```

Breakdown of the Contract

- Contract Declaration: `access(all) contract Counter` declares a new contract named Counter that is accessible to everyone.

- State Variable: `access(all) var count: Int` declares a public variable `count` of type `Int`. The `access(all)` modifier means that this variable can be read by anyone.
- Events: Two events are declared:
 - `CounterIncremented(newCount: Int)`: Emitted when the counter is incremented.
 - `CounterDecrementedException(newCount: Int)`: Emitted when the counter is decremented.
- Initializer: The `init()` function initializes the `count` variable to 0 when the contract is deployed.
- Public Functions:
 - `increment()`: Increases the count by 1 and emits the `CounterIncremented` event.
 - `decrement()`: Decreases the count by 1 and emits the `CounterDecrementedException` event.
 - `getCount()`: Returns the current value of `count`. The `view` modifier indicates that this function does not modify the contract's state.

Key Points

- Public Access: The `count` variable and the functions `increment()`, `decrement()`, and `getCount()` are all public, allowing anyone to interact with them.
- State Modification: The `increment()` and `decrement()` functions modify the state of the contract by changing the value of `count` and emitting events.
- Read Costs: Reading data from the blockchain is free on Flow. Executing scripts like the one you ran does not incur any costs. However, transactions that modify state, such as calling `increment()` or `decrement()`, will incur costs and require proper authorization.

What's Next?

In the upcoming tutorials, you'll learn how to:

- Modify the Counter: Invoke the `increment()` and `decrement()` functions to update the `count` value.
- Deploy Contracts: Use the Flow CLI to deploy your own smart contracts.
- Interact with Contracts Locally: Use the Flow Emulator to test contracts in a local development environment.
- Build Frontend Applications: Display data from smart contracts in a React application using the Flow Client Library.

By understanding the Counter contract and how to interact with it, you're building a solid foundation for developing more complex applications on the Flow blockchain.

Proceed to the next tutorial to learn how to create your own contracts and deploy them live using the Flow CLI.

<!-- Relative-style links. Does not render on the page -->

[Cadence]: <https://cadence-lang.org/>
 [EVM]: <https://flow.com/upgrade/crescendo/evm>

```
# account-linking-with-dapper.md:  
---  
title: Account Linking With NBA Top Shot  
description: Use Account Linking between the Dapper Wallet and Flow  
Wallet to effortlessly use NBA Top Shot Moments in your app.  
sidebarposition: 5  
sidebarcustomprops:  
  icon: 🏀  
---
```

Account Linking With NBA Top Shot

[Account Linking] is a powerful Flow feature that allows users to connect their wallets, enabling linked wallets to view and manage assets in one wallet with another. This feature helps reduce or even eliminate the challenges posed by other account abstraction solutions, which often lead to multiple isolated wallets and fragmented assets.

In this tutorial, you'll build a [simple onchain app] that allows users to sign into your app with their Flow wallet and view [NBA Top Shot] Moments that reside in their [Dapper Wallet] - without those users needing to sign in with Dapper.

Objectives

After completing this guide, you'll be able to:

- Pull your users' NBA Top Shot Moments into your Flow app without needing to transfer them out of their Dapper wallet

- Retrieve and list all NFT collections in any child wallet linked to a given Flow address

- Write a [Cadence] script to iterate through the storage of a Flow wallet to find NFT collections

- Run Cadence Scripts from the frontend

Prerequisites

Next.js and Modern Frontend Development

This tutorial uses [Next.js]. You don't need to be an expert, but it's helpful to be comfortable with development using a current React framework. You'll be on your own to select and use a package manager, manage Node versions, and other frontend environment tasks. If you don't have your own preference, you can just follow along with us and use [Yarn].

Flow Wallet

You'll need a [Flow Wallet], but you don't need to deposit any funds.

Moments NFTs

You'll need a [Dapper Wallet] containing some Moments NFTs, such as [NBA Top Shot] Moments.

Getting Started

This tutorial will use a [Next.js] project as the foundation of the frontend. Create a new project with:

```
zsh
npx create-next-app@latest
```

We will be using TypeScript and the App Router, in this tutorial.

Open your new project in the editor of your choice, install dependencies, and run the project.

```
zsh
yarn install
yarn run dev
```

If everything is working properly, you'll be able to navigate to localhost:3000 and see the default [Next.js] page.

Flow Cadence Setup

You'll need a few more dependencies to efficiently work with Cadence inside of your app.

Flow CLI and Types

The [Flow CLI] contains a number of command-line tools for interacting with the Flow ecosystem. If you don't already have it installed, you can add it with Brew (or using [other installation methods]):

```
zsh
brew install flow-cli
```

Once it's installed, you'll need to initialize Flow in your Next.js project. From the root, run:

```
zsh
flow init --config-only
```

The --config-only flag [initializes a project] with the just the config file. This allows the Flow CLI to interact with your project without adding any unnecessary files.

Next, you'll need to do a little bit of config work so that your project knows how to read Cadence files. Install the Flow Cadence Plugin:

```
zsh
yarn add flow-cadence-plugin --dev
```

Finally, open `next.config.ts` and update it to use the plugin with Raw Loader:

```
tsx
// next.config.ts
import type { NextConfig } from "next";
import FlowCadencePlugin from "flow-cadence-plugin";

const nextConfig: NextConfig = {
  webpack: (config) => {
    config.plugins.push(new FlowCadencePlugin())
  },
};

export default nextConfig;
```

Frontend Setup

We'll use the Flow Client Library [FCL] to manage blockchain interaction from the frontend. It's similar to viem, ethers, or web3.js, but works with the Flow blockchain and transactions and scripts written in Cadence.

```
zsh
yarn add @onflow/fcl
```

Go ahead and install dotenv as well:

```
yarn add dotenv
```

Provider Setup

A fair amount of boilerplate code is needed to set up your provider. We'll provide it, but since it's not the purpose of this tutorial, we'll be brief on explanations. For more details, check out the [App Quickstart Guide].

Add `app/providers/AuthProvider.tsx`:

```
tsx
'use client';
/* eslint-disable @typescript-eslint/no-explicit-any */

import { createContext, useContext, ReactNode } from 'react';
import useCurrentUser from '../hooks/use-current-user.hook';
```

```

interface State {
  user: any;
  loggedIn: any;
  logIn: any;
  logOut: any;
}

const AuthContext = createContext<State | undefined>(undefined);

interface AuthProviderProps {
  children: ReactNode;
}

const AuthProvider: React.FC<AuthProviderProps> = ({ children }) => {
  const [user, loggedIn, logIn, logOut] = useCurrentUser();

  return (
    <AuthContext.Provider
      value={ {
        user,
        loggedIn,
        logIn,
        logOut,
      } }
    >
      {children}
    </AuthContext.Provider>
  );
};

export default AuthProvider;

export const useAuth = (): State => {
  const context = useContext(AuthContext);

  if (context === undefined) {
    throw new Error('useAuth must be used within a AuthProvider');
  }

  return context;
};

```

Then, add app/hooks/use-current-user-hook.tsx:

```

tsx
import { useEffect, useState } from 'react';
import as fcl from '@onflow/fcl';

export default function useCurrentUser() {
  const [user, setUser] = useState({ addr: null });

  const logIn = () => {

```

```
fcl.authenticate();
};

const logOut = () => {
  fcl.unauthenticate();
};

useEffect(() => {
  fcl.currentUser().subscribe(setUser);
}, []);

return {user, loggedIn: user?.addr != null, logIn, logOut};
}
```

.env

Add a .env to the root and fill it with:

```
text
NEXTPUBLICACCESSNODEAPI="https://rest-mainnet.onflow.org"
NEXTPUBLICFLOWNETWORK="mainnet"
NEXTPUBLICWALLETCONNECTID=<YOUR ID HERE>
```

:::warning

Don't forget to replace <YOUR ID HERE> with your own [Wallet Connect] app id!

:::

Implement the Provider and Flow Config

Finally, open layout.tsx. Start by importing Flow dependencies and the AuthProvider:

```
tsx
import flowJSON from '../flow.json'
import  as fcl from "@onflow/fcl";

import AuthProvider from "./providers/AuthProvider";
```

Then add your Flow config:

```
tsx
fcl.config({
  "discovery.wallet": "https://fcl-discovery.onflow.org/authn",
  'accessNode.api': process.env.NEXTPUBLICACCESSNODEAPI,
  'flow.network': process.env.NEXTPUBLICFLOWNETWORK,
  'walletconnect.projectId': process.env.NEXTPUBLICWALLETCONNECTID
}).load({ flowJSON });
```

```
:::warning
```

We're going to force some things client side to get this proof-of-concept working quickly. Use Next.js best practices for a production app.

```
:::
```

Add a 'use client'; directive to the top of the file and delete the import for Metadata and fonts, as well as the code related to them.

Finally, update the <body> to remove the font references and suppress hydration warnings:

```
tsx
<body suppressHydrationWarning={true}>
```

Your code should be:

```
tsx
// layout.tsx
'use client';
import './globals.css';
import flowJSON from '../flow.json'
import { fcl } from '@onflow/fcl';

import AuthProvider from './providers/AuthProvider';

fcl.config({
  'discovery.wallet': "https://fcl-discovery.onflow.org/authn",
  'accessNode.api': process.env.NEXTPUBLICACCESSNODEAPI,
  'flow.network': process.env.NEXTPUBLICFLOWNETWORK,
  'walletconnect.projectId': process.env.NEXTPUBLICWALLETCONNECTID
}).load({ flowJSON });

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body suppressHydrationWarning={true}>
        <AuthProvider>
          {children}
        </AuthProvider>
      </body>
    </html>
  );
}
```

Add the Connect Button

Open page.tsx and clean up the demo code leaving only the <main> block:

```
tsx
import Image from "next/image";

export default function Home() {
  return (
    <div className="grid grid-rows-[20px1fr20px] items-center justify-items-center min-h-screen p-8 pb-20 gap-16 sm:p-20 font-[family-name:var(--font-geist-sans)]">
      <main className="flex flex-col gap-8 row-start-2 items-center sm:items-start">
        <div>TODO</div>
      </main>
    </div>
  );
}
```

Add a 'use client'; directive, import the useAuth hook and instantiate it in the Home function:

```
tsx
'use client';
import { useAuth } from "./providers/AuthProvider";
```

```
tsx
const { user, loggedIn, logIn, logOut } = useAuth();
```

Then add a button in the <main> to handle logging in or out:

```
tsx
<main className="flex flex-col gap-8 row-start-2 items-center sm:items-start">
  <div>Welcome</div>
  <button
    onClick={loggedIn ? logOut : logIn}
    className="px-6 py-2 text-white bg-green-600 hover:bg-green-700 rounded-lg shadow-md transition duration-200 ease-in-out focus:outline-none focus:ring-2 focus:ring-green-500 sm:ml-auto"
  >
    {loggedIn ? "Log Out" : "Log In"}
  </button>
</main>
```

Testing Pass

Run the app:

```
zsh
```

```
yarn dev
```

You'll see your Log In button in the middle of the window.

!Welcome

Click the button and log in with your Flow wallet.

!Flow Wallet

Account Linking

Now that your app is set up, you can make use of [Account Linking] to pull your NFTs from your Dapper Wallet, through your Flow Wallet, and into the app.

Setting Up Account Linking

If you haven't yet, you'll need to [link your Dapper Wallet] to your Flow Wallet.

:::warning

The Dapper Wallet requires that you complete KYC before you can use Account Linking. While this may frustrate some members of the community, it makes it much easier for app developers to design onboarding rewards and bonuses that are less farmable.

:::

Discovering the NFTs with a Script

With your accounts linked, your Flow Wallet now has a set of capabilities related to your Dapper Wallet and it's permitted to use those to view and even manipulate those NFTs and assets.

Before you can add a script that can handle this, you'll need to import the HybridCustody contract using the [Flow Dependency Manager]:

```
zsh
flow dependencies add mainnet://d8a7e05a7ac670c0.HybridCustody
```

Choose none to skip deploying on the emulator and skip adding testnet aliases. There's no point, these NFTs are on mainnet!

You'll get a complete summary from the Dependency Manager:

```
zsh
⚡ Dependency Manager Actions Summary
```

⌚ File System Actions:

- ✓ Contract HybridCustody from d8a7e05a7ac670c0 on mainnet installed
- ✓ Contract MetadataViews from 1d7e57aa55817448 on mainnet installed
- ✓ Contract FungibleToken from f233dcee88fe0abe on mainnet installed
- ✓ Contract ViewResolver from 1d7e57aa55817448 on mainnet installed
- ✓ Contract Burner from f233dcee88fe0abe on mainnet installed
- ✓ Contract NonFungibleToken from 1d7e57aa55817448 on mainnet installed
- ✓ Contract CapabilityFactory from d8a7e05a7ac670c0 on mainnet installed
- ✓ Contract CapabilityDelegator from d8a7e05a7ac670c0 on mainnet installed
- ✓ Contract CapabilityFilter from d8a7e05a7ac670c0 on mainnet installed

▣ State Updates:

- ✓ HybridCustody added to emulator deployments
- ✓ Alias added for HybridCustody on mainnet
- ✓ HybridCustody added to flow.json
- ✓ MetadataViews added to flow.json
- ✓ FungibleToken added to flow.json
- ✓ ViewResolver added to flow.json
- ✓ Burner added to flow.json
- ✓ NonFungibleToken added to flow.json
- ✓ CapabilityFactory added to emulator deployments
- ✓ Alias added for CapabilityFactory on mainnet
- ✓ CapabilityFactory added to flow.json
- ✓ CapabilityDelegator added to emulator deployments
- ✓ Alias added for CapabilityDelegator on mainnet
- ✓ CapabilityDelegator added to flow.json
- ✓ CapabilityFilter added to emulator deployments
- ✓ Alias added for CapabilityFilter on mainnet
- ✓ CapabilityFilter added to flow.json

Add app/cadence/scripts/FetchNFTsFromLinkedAccts.cdc. In it, add this script. Review the inline comments to see what each step is doing:

```

cadence
import "HybridCustody"
import "NonFungibleToken"
import "MetadataViews"

// This script iterates through a parent's child accounts,
// identifies private paths with an accessible NonFungibleToken.Provider,
and returns the corresponding typeIds

access(all) fun main(addr: Address): AnyStruct {
    let manager = getAuthAccount<auth(Storage)
&Account>(addr).storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(from: HybridCustody.ManagerStoragePath)
    ?? panic ("manager does not exist")
}

```

```

var typeIdsWithProvider: {Address: [String]} = {}
var nftViews: {Address: {UInt64: MetadataViews.Display}} = {}

let providerType = Type<auth(NonFungibleToken.Withdraw)
&{NonFungibleToken.Provider}>()
let collectionType: Type = Type<@{NonFungibleToken.CollectionPublic}>()

for address in manager.getChildAddresses() {
    let acct = getAuthAccount<auth(Storage, Capabilities)
&Account>(address)
    let foundTypes: [String] = []
    let views: {UInt64: MetadataViews.Display} = {}
    let childAcct = manager.borrowAccount(addr: address) ?? panic("child
account not found")

        // Iterate through storage paths to find NFTs that are controlled by
the parent account
        // To just find NFTs, check if thing stored is nft collection and
borrow it as NFT collection and get IDs
        for s in acct.storage.storagePaths {
            // Iterate through capabilities
            for c in acct.capabilities.storage.getControllers(forPath: s) {
                if !c.borrowType.isSubtype(of: providerType) {
                    // If this doesn't have providerType, it's not an NFT collection
                    continue
                }

                // We're dealing with a Collection but we need to check if
accessible from the parent account
                if let cap: Capability = childAcct.getCapability(controllerID:
c.capabilityID, type: providerType) { // Part 1
                    let providerCap = cap as!
Capability<&{NonFungibleToken.Provider}>

                    if !providerCap.check() {
                        // If I don't have access to control the account, skip it.
                        // Disable this check to do something else.
                        //
                        continue
                    }
                }

                foundTypes.append(cap.borrow<&AnyResource>()!.getType().identifier)
                typeIdsWithProvider[address] = foundTypes
                // Don't need to keep looking at capabilities, we can control
NFT from parent account
                break
            }
        }
}

// Iterate storage, check if typeIdsWithProvider contains the typeId,
if so, add to views

```

```

acct.storage.forEachStored(fun (path: StoragePath, type: Type): Bool
{
    if typeIdsWithProvider[address] == nil {
        return true
    }

    for key in typeIdsWithProvider.keys {
        for idx, value in typeIdsWithProvider[key]! {
            let value = typeIdsWithProvider[key]!

            if value[idx] != type.identifier {
                continue
            } else {
                if type.isInstance(collectionType) {
                    continue
                }
                if let collection =
acct.storage.borrow<&{NonFungibleToken.CollectionPublic}>(from: path) {
                    // Iterate over IDs & resolve the Display view
                    for id in collection.getIDs() {
                        let nft = collection.borrowNFT(id)!
                        if let display =
nft.resolveView(Type<MetadataViews.Display>())! as? MetadataViews.Display
{
                            views.insert(key: id, display)
                        }
                    }
                }
                continue
            }
        }
    }
    return true
})
nftViews[address] = views
}
return nftViews
}

```

:::warning

The above script is a relatively naive implementation. For production, you'll want to filter for only the collections you care about, and you will eventually need to add handling for very large collections in a wallet.

:::

Running the Script and Displaying the NFTs

Add a component in app/components called `DisplayLinkedNFTs.cdc`.

In it, import dependencies from React and FCL, as well as the script you just added:

```
tsx
import React, { useState, useEffect } from 'react';
import as fcl from "@onflow/fcl";
import as t from '@onflow/types';

import FetchNFTs from '../cadence/scripts/FetchNFTsFromLinkedAccts.cdc';
```

As we're using TypeScript, you should add some types as well to manage the data from the NFTs nicely. For now, just add them to this file:

```
typescript
type Thumbnail = {
  url: string;
};

type Moment = {
  name: string;
  description: string;
  thumbnail: Thumbnail;
};

type MomentsData = {
  [momentId: string]: Moment;
};

type ApiResponse = {
  [address: string]: MomentsData;
};

interface AddressDisplayProps {
  address: string;
}
```

Then, add the function for the component:

```
tsx
const DisplayLinkedNFTs: React.FC<AddressDisplayProps> = ({ address }) =>
{
  // TODO...

  return (
    <div>Nothing here yet</div>
  )
}

export default DisplayLinkedNFTs;
```

In the function, add a state variable to store the data retrieved by the script:

```
typescript
const [responseData, setresponseData] = useState<ApiResponse | null>(null);
```

Then, use useEffect to fetch the NFTs with the script and fcl.query:

```
tsx
useEffect(() => {
  const fetchLinkedAddresses = async () => {
    if (!address) return;

    try {
      const cadenceScript = FetchNFTs;

      // Fetch the linked addresses
      const response: ApiResponse = await fcl.query({
        cadence: cadenceScript,
        args: () => [fcl.arg(address, t.Address)],
      });

      console.log(JSON.stringify(response, null, 2));

      setresponseData(response);
    } catch (error) {
      console.error("Error fetching linked addresses:", error);
    }
  };

  fetchLinkedAddresses();
}, [address]);
```

Return to page.tsx, import your new component, and add an instance of <DisplayLinkedNFTs> that passes in the user's address and is only displayed while loggedIn.

```
tsx
{loggedIn && <DisplayLinkedNFTs address={user.addr} />}
```

Testing

Run the app again. If you have linked your account and have NFTs in that account, you'll see them in the console!

Displaying the Moments

Now that they're here, all to do is display them nicely! Return to DisplayLinkedNFTs.tsx. Add a helper function to confirm each returned

NFT matches the Moments format. You can update this to handle other NFTs you'd like to show as well.

```
:::warning
```

Remember, you'll also need to update the script in a production app to filter for only the collections you want, and handle large collections.

```
:::
```

```
tsx
// Type-checking function to validate moment structure
// eslint-disable-next-line @typescript-eslint/no-explicit-any
const isValidMoment = (moment: any): moment is Moment => {
  const isValid =
    typeof moment.name === 'string' &&
    typeof moment.description === 'string' &&
    moment.thumbnail &&
    typeof moment.thumbnail.url === 'string';

  if (!isValid) {
    console.warn('Invalid moment data:', moment);
  }

  return isValid;
};
```

Next, add a rendering function with some basic styling:

```
tsx
// Function to render moments with validation
const renderMoments = (data: ApiResponse) => {
  return Object.entries(data).map(([addr, moments]) => (
    <div key={addr} className="border border-gray-300 rounded-lg shadow-sm p-4 mb-6 bg-white">
      <h4 className="text-lg font-semibold mb-4 text-gray-800">Linked
      Wallet: {addr}</h4>
      <div className="grid grid-cols-1 gap-4 md:grid-cols-2 lg:grid-cols-3">
        {Object.entries(moments).map(([momentId, moment]) => (
          isValidMoment(moment) ?
            <div key={momentId} className="border border-gray-200 rounded-lg p-4 shadow hover:shadow-lg transition-shadow duration-200 bg-gray-50">
              <h5 className="text-md font-bold text-blue-600 mb-2">{moment.name}</h5>
              <p className="text-sm text-gray-600 mb-4">{moment.description}</p>
              <img src={moment.thumbnail.url} alt={moment.name}
                className="w-full h-32 object-cover rounded" />
            </div>
          ) : null
        )));
  
```

```

        </div>
    </div>
));
};

}

```

Finally, update the return with some more styling and the rendered NFT data:

```

tsx
return (
<div className="p-6 bg-gray-100 min-h-screen">
{address ? (
    <div className="max-w-4xl mx-auto">
        <h3 className="text-2xl font-bold text-gray-800 mb-4">Moments
Data:</h3>
        <div>
            {responseData ? renderMoments(responseData) : (
                <p className="text-gray-500">No Moments Data Available</p>
            )
        </div>
    </div>
) : (
    <div className="text-center text-gray-500 mt-8">No Address
Provided</div>
)
</div>
);

```

Further Polish

Finally, you can polish up your page.tsx to look a little nicer, and guide your users to the Account Linking process in the Dapper Wallet:

```

tsx
'use client';
import DisplayLinkedNFTs from './components/DisplayLinkedNFTs';
import { useAuth } from './providers/AuthProvider';

export default function Home() {
    const { user, loggedIn, logIn, logOut } = useAuth();

    return (
        <div className="grid grid-rows-[auto1frauto] items-center justify-
items-center min-h-screen p-8 sm:p-20 bg-gray-100 font-sans">
            <main className="flex flex-col gap-8 row-start-2 items-center w-
full max-w-5xl px-12 py-12 bg-white rounded-lg shadow-lg border border-
gray-200">
                {/ Message visible for all users /}
                <p className="text-center text-gray-700 mb-4">
                    Please link your Dapper wallet to view your NFTs. For more
information, check the{" "}
                <a

```

```

        href="https://support.meetdapper.com/hc/en-
us/articles/20744347884819-Account-Linking-and-FAQ"
        target="blank"
        rel="noopener noreferrer"
        className="text-blue-600 hover:text-blue-800 underline"
      >
      Account Linking and FAQ
    </a>.
  </p>

  <div className="flex flex-col sm:flex-row sm:items-center
sm:justify-between w-full gap-6">
    {/ Display user address or linked NFTs if logged in /}
    {loggedIn ? (
      <div className="text-lg font-semibold text-gray-800">
        Address: {user.addr}
      </div>
    ) : (
      <div className="text-lg font-semibold text-gray-800">
        Please log in to view your linked NFTs.
      </div>
    )}
  </div>

  {/ Login/Logout Button /}
  <button
    onClick={loggedIn ? logOut : logIn}
    className="px-6 py-2 text-white bg-blue-600 hover:bg-blue-700
rounded-lg shadow-md transition duration-200 ease-in-out focus:outline-
none focus:ring-2 focus:ring-blue-500 sm:ml-auto"
  >
    {loggedIn ? "Log Out" : "Log In"}
  </button>
</div>

  {/ Display NFTs if logged in /}
  {loggedIn && <DisplayLinkedNFTs address={user.addr} />}
</main>
</div>
);
}

```

Your app will now look like the [simple onchain app] demo!

Conclusion

In this tutorial, you took your first steps towards building powerful new experiences that meet your customers where they are. They can keep their assets in the wallet associate with one app, but also give your app the ability to use them - seamlessly, safely, and beautifully!

[Account Linking]: ./account-linking/index.md

[NBA Top Shot]: <https://nbatopshot.com>

[simple onchain app]: <https://nextjs-topshot-account-linking.vercel.app>

```
[Dapper Wallet]: https://meetdapper.com
[Cadence]: https://cadence-lang.org/docs
[Next.js]: https://nextjs.org/docs/app/getting-started/installation
[Yarn]: https://yarnpkg.com
[Flow CLI]: ../../tools/flow-cli/index.md
[other installation methods]: ../../tools/flow-cli/install.md
[initializes a project]: ../../tools/flow-cli/super-commands.md#init
[Flow Dependency Manager]: ../../tools/flow-cli/dependency-manager.md
[FCL]: ../../tools/clients/fcl-js/index.md
[App Quickstart Guide]: ./getting-started/fcl-quickstart.md
[Wallet Connect]: https://cloud.walletconnect.com/sign-in
[Flow Wallet]: https://wallet.flow.com
[link your Dapper Wallet]: https://support.meetdapper.com/hc/en-us/articles/20744347884819-Account-Linking-and-FAQ
```

```
# fungible-token.md:
```

```
---
```

```
title: How to Create a Fungible Token on Flow
sidebarlabel: Create a Fungible Token
description: Guide to creating a fungible token on Flow with the Flow CLI and Cadence.
sidebarposition: 2
---
```

```
:::info
```

This guide is an in-depth tutorial on launching a Fungible Token contract from scratch. To launch in 2 minutes using a tool check out Toucans

```
:::
```

What are Fungible Tokens?

Fungible tokens are digital assets that are interchangeable and indistinguishable with other tokens of the same type. This means that each token is identical in specification to every other token in circulation. Think of them like traditional money; every dollar bill has the same value as every other dollar bill. Fungible tokens play a crucial role in web3 ecosystems, serving as both a means of payment and an incentive for network participation. They can take on various roles including currencies, structured financial instruments, shares of index funds, and even voting rights in decentralized autonomous organizations.

Vaults on Flow

On the Flow blockchain and in the Cadence programming language, fungible tokens are stored in structures called resources. Resources are objects in Cadence that store data, but have special restrictions about how they can be stored and transferred, making them perfect for representing digital objects with real value.

You can learn more about resources in the Cadence documentation

and tutorials.

For fungible tokens specifically, tokens are represented by a resource type called a Vault:

```
cadence
access(all) resource interface Vault {

    /// Field that tracks the balance of a vault
    access(all) var balance: UFix64

}
```

Think of a Vault as a digital piggy bank.

Users who own fungible tokens store vault objects that track their balances

directly in their account storage. This is opposed to languages that track user balances in a central ledger smart contract.

When you transfer tokens from one vault to another:

1. The transferor's vault creates a temporary vault holding the transfer amount.
2. The original vault's balance decreases by the transfer amount.
3. The recipient's vault receives the tokens from the temporary vault and adds the temporary vault's balance to its own balance.
4. The temporary vault is then destroyed.

This process ensures secure and accurate token transfers on the Flow blockchain.

Fungible Token Standard

The Fungible Token Standard defines how a fungible token should behave on Flow.

Wallets and other platforms need to recognize these tokens, so they adhere to a specific interface, which defines fields like `balance`, `totalSupply`, `withdraw` functionality, and more. This interface ensures that all fungible tokens on Flow have a consistent structure and behavior. Click the link to the fungible token standard to see the full standard and learn about specific features and requirements.

Learn more about interfaces here.

Setting Up a Project

To start creating a Fungible Token on the Flow blockchain, you'll first need some tools and configurations in place.

Installing Flow CLI

The Flow CLI (Command Line Interface) provides a suite of tools that allow developers to interact seamlessly with the Flow blockchain.

If you haven't installed the Flow CLI yet and have Homebrew installed, you can run brew install flow-cli. If you don't have Homebrew, please follow the installation guide [here](#).

Initializing a New Project

>  Note: Here is a link to the completed code if you want to skip ahead or reference as you follow along. [View Completed Code](#)

Once you have the Flow CLI installed, you can set up a new project using the flow init command. This command initializes the necessary directory structure and a flow.json configuration file (a way to configure your project for contract sources, deployments, accounts, and more):

```
bash
flow init FooToken
```

> Note: Select "No" when it asks you to install core contracts for the purposes of this tutorial.

Upon execution, the command will generate the following directory structure:

```
/cadence
  /contracts
  /scripts
  /transactions
  /tests
flow.json
```

Now, navigate into the project directory:

```
bash
cd FooToken
```

In our configuration file, called flow.json, for the network we want to use, we are going to state the address the FungibleToken contract is deployed to via aliases in a new contracts section. Since it is a standard contract, it has already been deployed to the emulator, a tool that runs and emulates a local development version of the Flow Blockchain, for us. You can find addresses for other networks, like Testnet and Mainnet, on the Fungible Token Standard repo.

We'll also need to add the addresses for ViewResolver, MetadataViews,

and `FungibleTokenMetadataViews`, which are other important contracts to use.
These contracts are deployed to the Flow emulator by default, so there is not need to copy their code into your repo.
The addresses below are the addresses in the emulator that your contract will import them from.

```
json
"contracts": {
    "FungibleToken": {
        "aliases": {
            "emulator": "0xee82856bf20e2aa6"
        }
    },
    "FungibleTokenMetadataViews": {
        "aliases": {
            "emulator": "0xee82856bf20e2aa6"
        }
    },
    "ViewResolver": {
        "aliases": {
            "emulator": "0xf8d6e0586b0a20c7"
        }
    },
    "MetadataViews": {
        "aliases": {
            "emulator": "0xf8d6e0586b0a20c7"
        }
    }
}
```

Writing Our Token Contract

Next let's create a `FooToken` contract at `cadence/contract/FooToken.cdc` using the boilerplate generate command from the Flow CLI:

```
bash
flow generate contract FooToken
```

This will create a new file called `FooToken.cdc` in the `contracts` directory. Let's open it up and add some code.

In this contract file, we want to import our `FungibleToken` contract that we've defined in `flow.json`.

```
cadence
import "FungibleToken"
```

In this same file, let's create our contract which implements the `FungibleToken` contract interface (it does so by setting it following the `FooToken:`).

We'll also include fields for standard storage and public paths for our resource definitions.

In our init – which runs on the contract's first deployment and is used to set initial values – let's set an starting total supply of 1,000 tokens for this example.

```
cadence
// ...previous code

access(all) contract FooToken: FungibleToken {
    access(all) var totalSupply: UFix64

    access(all) let VaultStoragePath: StoragePath
    access(all) let VaultPublicPath: PublicPath
    access(all) let MinterStoragePath: StoragePath

    init() {
        self.totalSupply = 1000.0
        self.VaultStoragePath = /storage/fooTokenVault
        self.VaultPublicPath = /public/fooTokenVault
        self.MinterStoragePath = /storage/fooTokenMinter
    }
}
```

Creating a Vault

Inside of this contract, we'll need to create a resource for a Vault. The FungibleToken standard requires that your vault implements the FungibleToken.Vault interface.

This interface inherits from many other interfaces which enforce different functionality that you can learn about in the standard.

```
cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {
    // ...totalSupply and path code

    access(all) resource Vault: FungibleToken.Vault {

        access(all) var balance: UFix64

        init(balance: UFix64) {
            self.balance = balance
        }
    }
    // ...init code
}
```

In order to give an account a vault, we need to create a function

that creates a vault of our FooToken type and returns it to the account. This function takes a vaultType: Type argument that allows the caller to specify which type of Vault they want to create.

Contracts that implement multiple Vault types can use this argument, but since your contract is only implementing one Vault type, it can ignore the argument.

A simpler version of this function with no parameter should also be added to your Vault implementation.

```
cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {
    // ...other code

    access(all) resource Vault: FungibleToken.Vault {

        // ...other vault code

        access(all) fun createEmptyVault(): @FooToken.Vault {
            return <-create Vault(balance: 0.0)
        }

        // ...vault init code
    }

    // ...other code

    access(all) fun createEmptyVault(vaultType: Type): @FooToken.Vault {
        return <- create Vault(balance: 0.0)
    }

    // ...FooToken.init() code
}
```

Inside our Vault resource, we also need a way to withdraw balances. To do that, we need to add a withdraw() function that returns a new vault with the transfer amount and decrements the existing balance.

```
cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {

    // ...previous code

    access(all) resource Vault: FungibleToken.Vault {

        // ...other vault code

        access(FungibleToken.Withdraw) fun withdraw(amount: UFix64):
            @FooToken.Vault {
```

```

        self.balance = self.balance - amount
        return <-create Vault(balance: amount)
    }

    // ...vault init code
}

// ...additional code
}

```

As you can see, this function has an access(FungibleToken.Withdraw) access modifier.

This is an example of entitlements in Cadence.

Entitlements

are a way for developers to restrict access to privileged fields and functions

in a composite type like a resource when a reference is created for it.

In this example, the withdraw() function is always accessible to code that

controls the full Vault object, but if a reference is created for it, the withdraw() function can only be called if the reference is authorized by the owner with FungibleToken.Withdraw,

which is a standard entitlement

defined by the FungibleToken contract:

```

cadence
// Example of an authorized entitled reference to a FungibleToken.Vault
<auth(FungibleToken.Withdraw) &{FungibleToken.Vault}>

```

Entitlements are important to understand because they are what protects privileged functionality in your resource objects from being accessed by third-parties.

It is recommended to read the entitlements documentation to understand how to use the feature properly.

References can be freely up-casted and down-casted in Cadence, so it is important for privileged functionality to be protected by an entitlement so that it can only be accessed if it is authorized.

In addition to withdrawing, the vault also needs a way to deposit.

We'll typecast to make sure we are dealing with the correct token, update the vault balance, and destroy the vault. Add this code to your resource:

```

cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {

```

```

// ...previous code

access(all) resource Vault: FungibleToken.Vault {

    // ...other vault code

    access(all) fun deposit(from: @{FungibleToken.Vault}) {
        let vault <- from as! @FooToken.Vault
        self.balance = self.balance + vault.balance
        destroy vault
    }

    // ...vault init

}

// ...additional code
}

```

Many projects rely on events the signal when withdrawals, deposits, or burns happen.

Luckily, the FungibleToken standard handles the definition and emission of events for projects, so there is no need for you to add any events to your implementation for withdraw, deposit, and burn.

Here are the FungibleToken event definitions:

```

cadence
/// The event that is emitted when tokens are withdrawn from a Vault
access(all) event Withdrawn(type: String, amount: UFix64, from: Address?,  

fromUUID: UInt64, withdrawnUUID: UInt64, balanceAfter: UFix64)

/// The event that is emitted when tokens are deposited to a Vault
access(all) event Deposited(type: String, amount: UFix64, to: Address?,  

toUUID: UInt64, depositedUUID: UInt64, balanceAfter: UFix64)

/// Event that is emitted when the global burn method is called with a  

non-zero balance
access(all) event Burned(type: String, amount: UFix64, fromUUID: UInt64)

```

These events are emitted by the Vault interface in the FungibleToken contract whenever the relevant function is called on any implementation.

One important piece to understand about the Burned event in particular is that in order for it to be emitted when a Vault is burned, it needs to be burnt via the Burner contract's burn() method.

This will call the resource's burnCallback() function, which emits the event.

You'll need to also add this function to your token contract now:

```

cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {

    // ...previous code

    access(all) resource Vault: FungibleToken.Vault {

        // ...other vault code

        /// Called when a fungible token is burned via the Burner.burn()
method
        access(contract) fun burnCallback() {
            if self.balance > 0.0 {
                FooToken.totalSupply = FooToken.totalSupply -
self.balance
            }
            self.balance = 0.0
        }

        // ...vault init
    }

    // ...additional code
}

```

If you ever need to destroy a Vault with a non-zero balance, you should destroy it via the Burner.burn method so this important function can be called.

There are three other utility methods that need to be added to your Vault to get various information:

```

cadence
import "FungibleToken"

access(all) contract FooToken: FungibleToken {

    // ...previous code

    access(all) resource Vault: FungibleToken.Vault {

        // ...other vault code

        /// getSupportedVaultTypes optionally returns a list of vault
types that this receiver accepts
        access(all) view fun getSupportedVaultTypes(): {Type: Bool} {
            let supportedTypes: {Type: Bool} = {}
            supportedTypes[self.getType()] = true
            return supportedTypes
        }
    }
}

```

```

    /// Says if the Vault can receive the provided type in the
deposit method
    access(all) view fun isSupportedVaultType(type: Type): Bool {
        return self.getSupportedVaultTypes() [type] ?? false
    }

    /// Asks if the amount can be withdrawn from this vault
    access(all) view fun isAvailableToWithdraw(amount: UFix64): Bool
{
    return amount <= self.balance
}

// ...vault init

}

// ...additional code
}

```

Adding Support for Metadata Views

The Fungible Token standard also enforces that implementations provide functionality to return a set of standard views about the tokens via the ViewResolver and FungibleTokenMetadataViews definitions.

(You will need to add these imports to your contract now)

These provide developers with standard ways of representing metadata about a given token such as supply, token symbols, website links, and standard account paths and types that third-parties can access in a standard way.

You can see the metadata views documentation for a more thorough guide using a NFT contract as an example.

For now, you can add this code to your contract to support the important metadata views:

```

cadence
import "FungibleToken"

// Add these imports
import "MetadataViews"
import "FungibleTokenMetadataViews"

access(all) contract FooToken: FungibleToken {
    // ...other code

    access(all) view fun getContractViews(resourceType: Type?): [Type] {
        return [
            Type<FungibleTokenMetadataViews.FTView>(),
            Type<FungibleTokenMetadataViews.FTDisplay>(),
            Type<FungibleTokenMetadataViews.FTVaultData>(),
            Type<FungibleTokenMetadataViews.TotalSupply>()
        ]
    }
}

```

```

        ]
    }

    access(all) fun resolveContractView(resourceType: Type?, viewType: Type): AnyStruct? {
        switch viewType {
            case Type<FungibleTokenMetadataViews.FTView>():
                return FungibleTokenMetadataViews.FTView(
                    ftDisplay: self.resolveContractView(resourceType: nil, viewType: Type<FungibleTokenMetadataViews.FTDisplay>()) as! FungibleTokenMetadataViews.FTDisplay?,
                    ftVaultData: self.resolveContractView(resourceType: nil, viewType: Type<FungibleTokenMetadataViews.FTVaultData>()) as! FungibleTokenMetadataViews.FTVaultData?
                )
            case Type<FungibleTokenMetadataViews.FTDisplay>():
                let media = MetadataViews.Media(
                    file: MetadataViews.HTTPFile(
                        // Change this to your own SVG image
                        url: "https://assets.website-
files.com/5f6294c0c7a8cdd643b1c820/5f6294c0c7a8cda55cb1c936FlowWordmark.s
vg"
                    ),
                    mediaType: "image/svg+xml"
                )
                let medias = MetadataViews.Medias([media])
                return FungibleTokenMetadataViews.FTDisplay(
                    // Change these to represent your own token
                    name: "Example Foo Token",
                    symbol: "EFT",
                    description: "This fungible token is used as an
example to help you develop your next FT #onFlow.",
                    externalURL:
                    MetadataViews.ExternalURL("https://developers.flow.com/build/guides/fungi
ble-token"),
                    logos: medias,
                    socials: {
                        "twitter":
                    MetadataViews.ExternalURL("https://twitter.com/flowblockchain")
                })
            case Type<FungibleTokenMetadataViews.FTVaultData>():
                return FungibleTokenMetadataViews.FTVaultData(
                    storagePath: self.VaultStoragePath,
                    receiverPath: self.VaultPublicPath,
                    metadataPath: self.VaultPublicPath,
                    receiverLinkedType: Type<&FooToken.Vault>(),
                    metadataLinkedType: Type<&FooToken.Vault>(),
                    createEmptyVaultFunction: (fun():
                    @{FungibleToken.Vault} {
                        return <-FooToken.createEmptyVault(vaultType:
                    Type<@FooToken.Vault>())
                    })
                )
        }
    }
}

```

```

        case Type<FungibleTokenMetadataViews.TotalSupply>():
            return FungibleTokenMetadataViews.TotalSupply(
                totalSupply: FooToken.totalSupply
            )
        }
        return nil
    }

    // ...other code

    access(all) resource Vault: FungibleToken.Vault {

        // ...other vault code

        access(all) view fun getViews(): [Type] {
            return FooToken.getContractViews(resourceType: nil)
        }

        access(all) fun resolveView( view: Type): AnyStruct? {
            return FooToken.resolveContractView(resourceType: nil,
viewType: view)
        }

        // ...other vault code
    }

    // ...other FooToken code
}

```

Creating a Minter

Let's create a minter resource which is used to mint vaults that have tokens in them. We can keep track of tokens we are minting with totalSupply

If we want the ability to create new tokens, we'll need a way to mint them. To do that, let's create another resource on the FooToken contract. This will have a mintTokenfunction which can increase the total supply of the token.

```

cadence
import "FungibleToken"
import "MetadataViews"
import "FungibleTokenMetadataViews"

access(all) contract FooToken: FungibleToken {

    // ...additional contract code

    // Add this event
    access(all) event TokensMinted(amount: UFix64, type: String)

    /// Minter
}

```

```

    /**
     * Resource object that token admin accounts can hold to mint new
     * tokens.
    /**
     access(all) resource Minter {
        /**
         * mintTokens
        /**
         * Function that mints new tokens, adds them to the total
         * supply,
        /**
         * and returns them to the calling context.
        /**
         access(all) fun mintTokens(amount: UFix64): @FooToken.Vault {
            FooToken.totalSupply = FooToken.totalSupply + amount
            let vault <-create Vault(balance: amount)
            emit TokensMinted(amount: amount, type:
                vault.getType().identifier)
            return <-vault
        }
    }

    // ...additional contract code
}

```

We also want to decide which account/s we want to give this ability to. In our example, we'll give it to the account where the contract is deployed.

We can set this in the contract init function below the setting of total supply so that when the contract is created the minter is stored on the same account.

```

cadence
import "FungibleToken"
import "MetadataViews"
import "FungibleTokenMetadataViews"

access(all) contract FooToken: FungibleToken {

    // ...additional contract code

    init() {
        self.totalSupply = 1000.0 // existed before
        self.account.save(<- create Minter(), to: self.MinterStoragePath)
    }
}

```

After each of these steps, your `FooToken.cdc` contract file should now look like this:

```

cadence
import "FungibleToken"
import "MetadataViews"

```

```

import "FungibleTokenMetadataViews"

access(all) contract FooToken: FungibleToken {

    /// The event that is emitted when new tokens are minted
    access(all) event TokensMinted(amount: UFix64, type: String)

    /// Total supply of FooTokens in existence
    access(all) var totalSupply: UFix64

    /// Storage and Public Paths
    access(all) let VaultStoragePath: StoragePath
    access(all) let VaultPublicPath: PublicPath
    access(all) let ReceiverPublicPath: PublicPath
    access(all) let MinterStoragePath: StoragePath

    access(all) view fun getContractViews(resourceType: Type?): [Type] {
        return [
            Type<FungibleTokenMetadataViews.FTView>(),
            Type<FungibleTokenMetadataViews.FTDisplay>(),
            Type<FungibleTokenMetadataViews.FTVaultData>(),
            Type<FungibleTokenMetadataViews.TotalSupply>()
        ]
    }

    access(all) fun resolveContractView(resourceType: Type?, viewType: Type): AnyStruct? {
        switch viewType {
            case Type<FungibleTokenMetadataViews.FTView>():
                return FungibleTokenMetadataViews.FTView(
                    ftDisplay: self.resolveContractView(resourceType: nil, viewType: Type<FungibleTokenMetadataViews.FTDisplay>()) as!
                    FungibleTokenMetadataViews.FTDisplay?,
                    ftVaultData: self.resolveContractView(resourceType: nil, viewType: Type<FungibleTokenMetadataViews.FTVaultData>()) as!
                    FungibleTokenMetadataViews.FTVaultData?
                )
            case Type<FungibleTokenMetadataViews.FTDisplay>():
                let media = MetadataViews.Media(
                    file: MetadataViews.HTTPFile(
                        // Change this to your own SVG image
                        url: "https://assets.website-
files.com/5f6294c0c7a8cdd643b1c820/5f6294c0c7a8cda55cb1c936FlowWordmark.s
vg"
                    ),
                    mediaType: "image/svg+xml"
                )
                let medias = MetadataViews.Medias([media])
                return FungibleTokenMetadataViews.FTDisplay(
                    // Change these to represent your own token
                    name: "Example Foo Token",
                    symbol: "EFT",
                    description: "This fungible token is used as an
example to help you develop your next FT #onFlow."
                )
        }
    }
}

```

```

        externalURL:
    MetadataViews.ExternalURL("https://developers.flow.com/build/guides/fungible-token"),
        logos: medias,
        socials: {
            "twitter":
    MetadataViews.ExternalURL("https://twitter.com/flowblockchain")
        }
    )
case Type<FungibleTokenMetadataViews.FTVaultData>():
    return FungibleTokenMetadataViews.FTVaultData(
        storagePath: self.VaultStoragePath,
        receiverPath: self.VaultPublicPath,
        metadataPath: self.VaultPublicPath,
        receiverLinkedType: Type<&FooToken.Vault>(),
        metadataLinkedType: Type<&FooToken.Vault>(),
        createEmptyVaultFunction: (fun():
@{FungibleToken.Vault} {
            return <-FooToken.createEmptyVault(vaultType:
Type<@FooToken.Vault>())
        })
    )
case Type<FungibleTokenMetadataViews.TotalSupply>():
    return FungibleTokenMetadataViews.TotalSupply(
        totalSupply: FooToken.totalSupply
    )
}
return nil
}

access(all) resource Vault: FungibleToken.Vault {

    /// The total balance of this vault
    access(all) var balance: UFix64

    // initialize the balance at resource creation time
    init(balance: UFix64) {
        self.balance = balance
    }

    /// Called when a fungible token is burned via the Burner.burn()
method
    access(contract) fun burnCallback() {
        if self.balance > 0.0 {
            FooToken.totalSupply = FooToken.totalSupply -
self.balance
        }
        self.balance = 0.0
    }

    access(all) view fun getViews(): [Type] {
        return FooToken.getContractViews(resourceType: nil)
    }
}

```

```

        access(all) fun resolveView( view: Type): AnyStruct? {
            return FooToken.resolveContractView(resourceType: nil,
viewType: view)
        }

        access(all) view fun getSupportedVaultTypes(): {Type: Bool} {
            let supportedTypes: {Type: Bool} = {}
            supportedTypes[self.getType()] = true
            return supportedTypes
        }

        access(all) view fun isSupportedVaultType(type: Type): Bool {
            return self.getSupportedVaultTypes()[type] ?? false
        }

        access(all) view fun isAvailableToWithdraw(amount: UFix64): Bool
{
            return amount <= self.balance
}

        access(FungibleToken.Withdraw) fun withdraw(amount: UFix64):
@FooToken.Vault {
            self.balance = self.balance - amount
            return <-create Vault(balance: amount)
}

        access(all) fun deposit(from: @{FungibleToken.Vault}) {
            let vault <- from as! @FooToken.Vault
            self.balance = self.balance + vault.balance
            vault.balance = 0.0
            destroy vault
}

        access(all) fun createEmptyVault(): @FooToken.Vault {
            return <-create Vault(balance: 0.0)
}
}

access(all) resource Minter {
    /// mintTokens
    ///
    /// Function that mints new tokens, adds them to the total
supply,
    /// and returns them to the calling context.
    ///
    access(all) fun mintTokens(amount: UFix64): @FooToken.Vault {
        FooToken.totalSupply = FooToken.totalSupply + amount
        let vault <-create Vault(balance: amount)
        emit TokensMinted(amount: amount, type:
vault.getType().identifier)
        return <-vault
    }
}

```

```

access(all) fun createEmptyVault(vaultType: Type): @FooToken.Vault {
    return <- create Vault(balance: 0.0)
}

init() {
    self.totalSupply = 1000.0

    self.VaultStoragePath = /storage/fooTokenVault
    self.VaultPublicPath = /public/fooTokenVault
    self.MinterStoragePath = /storage/fooTokenMinter

    // Create the Vault with the total supply of tokens and save it
    in storage
    //
    let vault <- create Vault(balance: self.totalSupply)
    emit TokensMinted(amount: vault.balance, type:
    vault.getType().identifier)
    self.account.storage.save(<-vault, to: self.VaultStoragePath)

    // Create a public capability to the stored Vault that exposes
    // the deposit method and getAcceptedTypes method through the
    Receiver interface
    // and the balance method through the Balance interface
    //
    let fooTokenCap =
    self.account.capabilities.storage.issue<&FooToken.Vault>(self.VaultStoragePath)
        self.account.capabilities.publish(fooTokenCap, at:
    self.VaultPublicPath)

    let minter <- create Minter()
    self.account.storage.save(<-minter, to: self.MinterStoragePath)
}
}

```

Deploying the Contract

In order to use the contract, we need to deploy it to the network we want to use it on.

In our case we are going to deploy it to emulator while developing.

Back in our flow.json, let's add our FooToken to the contracts after FungibleToken with the path of the source code:

```

json
"FooToken": "cadence/contracts/FooToken.cdc"

```

Let's also add a new deployments section to flow.json with the network we want to deploy it to, emulator, the account we want it deployed to emulator-account,

and the list of contracts we want deployed in the array.

```
json
"deployments": {
  "emulator": {
    "emulator-account": ["FooToken"]
  }
}
```

Next, using the Flow CLI, we will start the emulator. As mentioned, this will give us a local development environment for the Flow Blockchain.

```
bash
flow emulator start
```

Open a new terminal and run the following to deploy your project:

```
bash
flow project deploy
```

Congrats, you've deployed your contract to the Flow Blockchain emulator. To read more about deploying your project to other environments, see the CLI docs.

Reading the Token's Total Supply

Let's now check that our total supply was initialized with 1,000 FooTokens. Go ahead and create a script called `gettotsupply.cdc` using the `generate` command.

```
bash
flow generate script gettotalsupply
```

In `cadence/scripts/gettotalsupply.cdc` (which was just created), let's add this code which will log the `totalSupply` value from the `FooToken` contract:

```
cadence
import "FooToken"

access(all) fun main(): UFix64 {
  return FooToken.totalSupply
}
```

To run this using the CLI, enter this in your terminal:

```
bash
flow scripts execute cadence/scripts/gettotalsupply.cdc
```

In the terminal where you started the emulator, you should see Result:
1000.0

To learn more about running scripts using Flow CLI, see the docs.

Giving Accounts the Ability to Receive Tokens

On Flow, newly created accounts cannot receive arbitrary assets. They need to be initialized to receive resources. In our case, we want to give accounts tokens and we'll need to create a Vault (which acts as a receiver) on each account that we want to have the ability to receive tokens. To do this, we'll need to run a transaction which will create the vault and set it in their storage using the `createEmptyVault()` function we created earlier on the contract.

Let's first create the file at `cadence/transactions/setupftaccount.cdc` using the `generate` command:

```
bash
flow generate transaction setupftaccount
```

Then add this code to it.

This will call the `createEmptyVault` function, save it in storage, and create a capability for the vault which will later allow us to read from it
(To learn more about capabilities, see the Cadence docs here).

```
cadence
import "FungibleToken"
import "FooToken"

transaction () {

    prepare(signer: auth(BorrowValue, IssueStorageCapabilityController,
    PublishCapability, SaveValue) &Account) {

        // Return early if the account already stores a FooToken Vault
        if signer.storage.borrow<&FooToken.Vault>(from:
        FooToken.VaultStoragePath) != nil {
            return
        }

        let vault <- FooToken.createEmptyVault(vaultType:
        Type<@FooToken.Vault>())

        // Create a new FooToken Vault and put it in storage
        signer.storage.save(<-vault, to: FooToken.VaultStoragePath)

        // Create a public capability to the Vault that exposes the Vault
        interfaces
        let vaultCap =
        signer.capabilities.storage.issue<&FooToken.Vault>()
```

```
        FooToken.VaultStoragePath
    )
    signer.capabilities.publish(vaultCap, at:
FooToken.VaultPublicPath)
}
}
```

There are also examples of generic transactions that you can use to setup an account for ANY fungible token using metadata views! You should check those out and try to use generic transactions whenever it is possible.

Next let's create a new emulator account using the CLI. We'll use this account to create a new vault and mint tokens into it. Run:

```
bash
flow accounts create
```

Let's call it test-acct and select "Emulator" for the network:

```
bash
test-acct
```

This will have added a new account, called test-acct to your flow.json.

To call our setup account transaction from the CLI, we'll run the following:

```
bash
flow transactions send ./cadence/transactions/setupftaccount.cdc --signer
test-acct --network emulator
```

To learn more about running transactions using CLI, see the docs.

Reading a Vault's Balance

Let's now read the balance of the newly created account (test-acct) to check it's zero.

Create this new script file cadence/scripts/getfootokenbalance.cdc:

```
bash
flow generate script getfootokenbalance
```

Add this code which attempts to borrow the capability from the account requested and logs the vault balance if permitted:

```
cadence
```

```

import "FungibleToken"
import "FooToken"
import "FungibleTokenMetadataViews"

access(all) fun main(address: Address): UFix64 {
    let vaultData = FooToken.resolveContractView(resourceType: nil,
viewType: Type<FungibleTokenMetadataViews.FTVaultData>()) as!
FungibleTokenMetadataViews.FTVaultData?
    ?? panic("Could not get FTVaultData view for the FooToken
contract")

    return
getAccount(address).capabilities.borrow<&{FungibleToken.Balance}>(
    vaultData.metadataPath
)?.balance
    ?? panic("Could not borrow a reference to the FooToken Vault in
account "
        .concat(address.toString()).concat(" at path
") .concat(vaultData.metadataPath.toString())
        .concat(". Make sure you are querying an address that has an
FooToken Vault set up properly."))
}

```

To run this script using the CLI, enter the following in your terminal.
Note: you'll need to replace 123 with the address created by CLI
in your flow.json for the test-acct address.

```

bash
flow scripts execute cadence/scripts/getfootokenbalance.cdc 123 // change
"123" to test-acct address

```

You should see a balance of zero logged.

Minting More Tokens

Now that we have an account with a vault, let's mint some tokens into it
using the Minter we created on the contract account.

To do this, let's create a new transaction file
cadence/transactions/mintfootoken.cdc:

```

bash
flow generate transaction mintfootoken

```

Next, let's add the following code to the mintfootoken.cdc file.
This code will attempt to borrow the minting capability
and mint 20 new tokens into the receivers account.

```

cadence
import "FungibleToken"
import "FooToken"

```

```

transaction(recipient: Address, amount: UFix64) {

    /// Reference to the Example Token Minter Resource object
    let tokenMinter: &FooToken.Minter

    /// Reference to the Fungible Token Receiver of the recipient
    let tokenReceiver: &{FungibleToken.Receiver}

    prepare(signer: auth(BorrowValue) &Account) {

        // Borrow a reference to the admin object
        self.tokenMinter = signer.storage.borrow<&FooToken.Minter>(from:
        FooToken.MinterStoragePath)
            ?? panic("Cannot mint: Signer does not store the FooToken
        Minter in their account!")

        self.tokenReceiver =
        getAccount(recipient).capabilities.borrow<&{FungibleToken.Receiver}>(FooT
        oken.VaultPublicPath)
            ?? panic("Could not borrow a Receiver reference to the
        FungibleToken Vault in account "
                .concat(recipient.toString()).concat(" at path
        ").concat(FooToken.VaultPublicPath.toString())
                .concat(". Make sure you are sending to an address that
        has ")
                .concat("a FungibleToken Vault set up properly at the
        specified path."))
    }

    execute {

        // Create mint tokens
        let mintedVault <- self.tokenMinter.mintTokens(amount: amount)

        // Deposit them to the receiver
        self.tokenReceiver.deposit(from: <-mintedVault)
    }
}

```

To run this transaction, enter this in your terminal.
Note: 123 should be replaced with address of test-acct found in your flow.json.
This command also states to sign with our emulator-account on the Emulator network.

```

bash
flow transactions send ./cadence/transactions/mintfootoken.cdc 123 20.0 -
--signer emulator-account --network emulator

```

Let's go ahead and read the vault again. Remember to replace 123 with the correct address.

```
bash
flow scripts execute cadence/scripts/getfootokenbalance.cdc 123
```

It should now say 20 tokens are in the vault.

Transferring Tokens Between Accounts

The final functionality we'll add is the ability to transfer tokens from one account to another.

To do that, create a new cadence/transactions/transferfootoken.cdc transaction file:

```
bash
flow generate transaction transferfootoken
```

Let's add the code which states that the signer of the transaction will withdraw from their vault and put it into the receiver's vault which will be passed as a transaction argument.

```
cadence
import "FungibleToken"
import "FooToken"

transaction(to: Address, amount: UFix64) {
    // The Vault resource that holds the tokens that are being transferred
    let sentVault: @{}{FungibleToken.Vault}

    prepare(signer: auth(BorrowValue) &Account) {
        // Get a reference to the signer's stored vault
        let vaultRef = signer.storage.borrow<auth(FungibleToken.Withdraw) &FooToken.Vault>(from: FooToken.VaultStoragePath)
            ?? panic("The signer does not store an FooToken.Vault object at the path ")
                .concat(FooToken.VaultStoragePath.toString())
                .concat("."). The signer must initialize their account with this vault first!")

        // Withdraw tokens from the signer's stored vault
        self.sentVault <- vaultRef.withdraw(amount: amount)
    }

    execute {
        // Get the recipient's public account object
        let recipient = getAccount(to)

        // Get a reference to the recipient's Receiver
    }
}
```

```

        let receiverRef =
    recipient.capabilities.borrow<&{FungibleToken.Receiver}>(FooToken.VaultPu
blicPath)
        ?? panic("Could not borrow a Receiver reference to the
FooToken Vault in account "
            .concat(recipient.toString()).concat(" at path
") .concat(FooToken.VaultPublicPath.toString())
            .concat(". Make sure you are sending to an address that
has ")
            .concat("a FooToken Vault set up properly at the
specified path."))
    // Deposit the withdrawn tokens in the recipient's receiver
    receiverRef.deposit(from: <-self.sentVault)
}
}

```

To send our tokens, we'll need to create a new account to send them to. Let's make one more account on emulator. Run:

```
bash
flow accounts create
```

And pick the name:

```
bash
test-acct-2
```

Make sure to select Emulator as the network.

Don't forget the new account will need a vault added, so let's run the following transaction to add one:

```
bash
flow transactions send ./cadence/transactions/setupftaccount.cdc --signer
test-acct-2 --network emulator
```

Now, let's send 1 token from our earlier account to the new account. Remember to replace 123 with account address of test-acct-2.

```
bash
flow transactions send ./cadence/transactions/transferfootoken.cdc 123
1.0 --signer test-acct --network emulator
```

After that, read the balance of test-acct-2 (replace the address 123).

```
bash
flow scripts execute cadence/scripts/getfootokenbalance.cdc 123
```

You should now see 1 token in test-acct-2 account!

The transfer transaction also has a generic version that developers are encouraged to use!

More

- View a repo of this example code
- Review an ExampleToken contract implementing all of the remaining FungibleToken interface
- View the Flow Token Standard

nft.md:

```
---
```

title: How to Create an NFT Project on Flow
sidebarlabel: Create an NFT Project
description: Guide to creating an NFT Project with the Flow CLI and Cadence.
sidebarposition: 3

:::info

This guide is an in-depth tutorial on launching NFT contracts from scratch.

To launch in 2 minutes using a tool check out Touchstone

:::

What are NFTs

NFTs, or Non-Fungible Tokens, represent a unique digital asset verified using blockchain technology. Unlike cryptocurrencies such as Bitcoin, which are fungible and can be exchanged on a one-for-one basis, NFTs are distinct and cannot be exchanged on a like-for-like basis. This uniqueness and indivisibility make them ideal for representing rare and valuable items like art, collectibles, tickets and even real estate. Their blockchain-backed nature ensures the authenticity and ownership of these digital assets.

Setting Up a Project

To start creating an NFT on the Flow blockchain, you'll first need some tools and configurations in place.

Installing Flow CLI

The Flow CLI (Command Line Interface) provides a suite of tools that allow developers to interact seamlessly with the Flow blockchain.

If you haven't installed the Flow CLI yet and have Homebrew installed, you can run brew install flow-cli. If you don't have Homebrew, please follow the installation guide [here](#).

Initializing a New Project

> **!** Note: Here is a link to the completed code if you want to skip ahead or reference as you follow along. [View Completed Code](#)

Once you have the Flow CLI installed, you can set up a new project using the flow init command. This command initializes the necessary directory structure and a flow.json configuration file (a way to configure your project for contract sources, deployments, accounts, and more):

```
bash
flow init foobar-nft
```

> Note: Select "No" when it asks you to install core contracts for the purposes of this tutorial.

Upon execution, the command will generate the following directory structure:

```
/cadence
  /contracts
  /scripts
  /transactions
  /tests
flow.json
```

Now, navigate into the project directory:

```
bash
cd foobar-nft
```

To begin, let's create a contract file named FooBar for the FooBar token, which will be the focus of this tutorial. To do this, we can use the boilerplate generate command from the Flow CLI:

```
bash
flow generate contract FooBar
```

This will create a new file at cadence/contracts/FooBar.cdc with the following contents:

```
cadence
access(all) contract FooBar {
    init() {}
}
```

Now, add these contracts to your flow.json. These are important contracts that your contract will import that are pre-deployed to the emulator.

```
json
"contracts": {
    "NonFungibleToken": {
        "aliases": {
            "emulator": "f8d6e0586b0a20c7"
        }
    },
    "ViewResolver": {
        "aliases": {
            "emulator": "0xf8d6e0586b0a20c7"
        }
    },
    "MetadataViews": {
        "aliases": {
            "emulator": "0xf8d6e0586b0a20c7"
        }
    }
}
```

Setting Up Our NFT on the Contract

Understanding Resources

On the Flow blockchain, "Resources" are a key feature of the Cadence programming language. They represent unique, non-duplicable assets, ensuring that they can only exist in one place at a time. This concept is crucial for representing NFTs on Flow, as it guarantees their uniqueness.

To begin, let's define a basic NFT resource. This resource requires an init method, which is invoked when the resource is instantiated:

```
cadence
access(all) contract FooBar {

    access(all) resource NFT {
        init() {}
    }

    init() {}
}
```

Every resource in Cadence has a unique identifier assigned to it.

We can use it to set an ID for our NFT. Here's how you can do that:

```
cadence
access(all) contract FooBar {

    access(all) resource NFT {
        access(all) let id: UInt64

        init() {
            self.id = self.uuid
        }
    }

    init() {}
}
```

To control the creation of NFTs, it's essential to have a mechanism that restricts their minting. This ensures that not just anyone can create an NFT and inflate its supply.

To achieve this, you can introduce an NFTMinter resource that contains a `createNFT` function:

```
cadence
access(all) contract FooBar {

    // ...[previous code]...

    access(all) resource NFTMinter {
        access(all) fun createNFT(): @NFT {
            return <-create NFT()
        }
    }

    init() {}
}

init() {}
}
```

In this example, the NFTMinter resource will be stored on the contract account's storage.

This means that only the contract account will have the ability to mint new NFTs.

To set this up, add the following line to the contract's `init` function:

```
cadence
access(all) contract FooBar {

    // ...[previous code]...

    init() {
        self.account.storage.save(<- create NFTMinter(), to:
/storage/fooBarNFTMinter)
```

```
    }
}
```

Setting Up an NFT Collection

Storing individual NFTs directly in an account's storage can cause issues, especially if you want to store multiple NFTs. Instead, it's required to create a collection that can hold multiple NFTs. This collection can then be stored in the account's storage.

Start by creating a new resource named `Collection`. This resource will act as a container for your NFTs, storing them in a dictionary indexed by their IDs.

```
cadence
access(all) contract FooBar {

    // ...[NFT resource code]...

    access(all) resource Collection {

        access(all) var ownedNFTs: @{UInt64: NFT}

        init() {
            self.ownedNFTs <- {}
        }

    }

    // ...[NFTMinter code]...
}
```

Fitting the Flow NFT Standard

To ensure compatibility and interoperability within the Flow ecosystem, it's crucial that your NFT contract adheres to the Flow NFT standard. This standard defines the events, functions, resources, metadata and other elements that a contract should have. By following this standard, your NFTs will be compatible with various marketplaces, apps, and other services within the Flow ecosystem.

Applying the Standard

To start, you need to inform the Flow blockchain that your contract will implement the `NonFungibleToken` standard. Since it's a standard, there's no need for deployment. It's already available on the Emulator, Testnet, and Mainnet for the community's benefit.

Begin by importing the token standard into your contract

and adding the correct interface conformances to FooBar, NFT, and Collection:

```
cadence
import "NonFungibleToken"

access(all) contract FooBar: NonFungibleToken {

    /// Standard Paths
    access(all) let CollectionStoragePath: StoragePath
    access(all) let CollectionPublicPath: PublicPath

    /// Path where the minter should be stored
    /// The standard paths for the collection are stored in the
    collection resource type
    access(all) let MinterStoragePath: StoragePath

    // ...contract code

    access(all) resource NFT: NonFungibleToken.NFT {
        // ...NFT code
    }

    access(all) resource Collection: NonFungibleToken.Collection {

        // Make sure to update this field!
        access(all) var ownedNFTs: @{UInt64: {NonFungibleToken.NFT}}}

        // ...Collection Code
    }

    // ...rest of the contract code

    init() {
        // Set the named paths
        self.CollectionStoragePath = /storage/fooBarNFTCollection
        self.CollectionPublicPath = /public/fooBarNFTCollection
        self.MinterStoragePath = /storage/fooBarNFTMinter
        self.account.storage.save(<- create NFTMinter(), to:
self.MinterStoragePath)
    }
}
```

As you can see, we also added standard paths for the Collection and Minter

These interface conformances for NFT and Collection inherit from other interfaces that provide important functionality and restrictions for your NFT and Collection types.

To allow accounts to create their own collections, add a function

in the main contract that creates a new Collection and returns it. This function takes a nftType: Type argument that allows the caller to specify which type of Collection they want to create. Contracts that implement multiple NFT and/or Collection types can use this argument, but since your contract is only implementing one NFT and Collection type, it can ignore the argument. You'll also want to add a simpler one directly to the NFT and Collection definitions so users can directly create a collection from an existing collection:

```

cadence
access(all) contract FooBar: NonFungibleToken {

    // ...other FooBar contract code

    access(all) resource NFT: NonFungibleToken.NFT {
        // ...NFT code

        access(all) fun createEmptyCollection():
            @{NonFungibleToken.Collection} {
                return <-FooBar.createEmptyCollection(nftType:
Type<@FooBar.NFT>())
            }
    }

    access(all) resource Collection: NonFungibleToken.Collection {
        // ...other Collection code

        /// createEmptyCollection creates an empty Collection of the same
        type
        /// and returns it to the caller
        /// @return A an empty collection of the same type
        access(all) fun createEmptyCollection():
            @{NonFungibleToken.Collection} {
                return <-FooBar.createEmptyCollection(nftType:
Type<@FooBar.NFT>())
            }
    }

    // ...other FooBar contract code

    /// createEmptyCollection creates an empty Collection for the
    specified NFT type
    /// and returns it to the caller so that they can own NFTs
    access(all) fun createEmptyCollection(nftType: Type):
        @{NonFungibleToken.Collection} {
            return <- create Collection()
        }
    // ...FooBar minter and init code
}

```

To manage the NFTs within a collection, you'll need functions to deposit and withdraw NFTs. Here's how you can add a deposit function:

```
cadence
access(all) resource Collection: NonFungibleToken.Collection {

    access(all) var ownedNFTs: @{UInt64: {NonFungibleToken.NFT}} 

    /// deposit takes a NFT and adds it to the collections dictionary
    /// and adds the ID to the id array
    access(all) fun deposit(token: @{NonFungibleToken.NFT}) {
        let token <- token as! @FooBar.NFT
        let id = token.id

        // add the new token to the dictionary which removes the old one
        let oldToken <- self.ownedNFTs[token.id] <- token

        destroy oldToken
    }

    // ...[following code]...
}
```

Similarly, you can add a withdraw function to remove an NFT from the collection:

```
cadence
access(all) resource Collection: NonFungibleToken.Collection {
    // ...[deposit code]...

    /// withdraw removes an NFT from the collection and moves it to the
    caller
    access(NonFungibleToken.Withdraw) fun withdraw(withdrawID: UInt64):
    @{NonFungibleToken.NFT} {
        let token <- self.ownedNFTs.remove(key: withdrawID)
        ?? panic("FooBar.Collection.withdraw: Could not withdraw
an NFT with ID "
            .concat(withdrawID.toString())
            .concat(". Check the submitted ID to make sure it
is one that this collection owns."))
        return <-token
    }

    // ...[createEmptyCollection code]...
}
```

As you can see, this function has an `access(NonFungibleToken.Withdraw)` access modifier.

This is an example of entitlements in Cadence.

Entitlements are a way for developers to restrict access to privileged fields and functions in a composite type like a resource when a reference is created for it. In this example, the withdraw() function is always accessible to code that controls the full Collection object, but if a reference is created for it, the withdraw() function can only be called if the reference is authorized by the owner with NonFungibleToken.Withdraw, which is a standard entitlement defined by the NonFungibleToken contract:

```
cadence
// Example of an authorized entitled reference to a
NonFungibleToken.Collection
<auth(NonFungibleToken.Withdraw) &{NonFungibleToken.Collection}>
```

Entitlements are important to understand because they are what protects privileged functionality in your resource objects from being accessed by third-parties.

It is recommended to read the entitlements documentation to understand how to use the feature properly.

References can be freely up-casted and down-casted in Cadence, so it is important for privileged functionality to be protected by an entitlement so that it can only be accessed if it is authorized.

Standard NFT Events

Many projects rely on events the signal when withdrawals or deposits happen. Luckily, the NonFungibleToken standard handles the definition and emission of events for projects, so there is no need for you to add any events to your implementation for withdraw and deposit.

Here are the FungibleToken event definitions:

```
cadence
    /// Event that is emitted when a token is withdrawn,
    /// indicating the type, id, uuid, the owner of the collection that
    it was withdrawn from,
    /// and the UUID of the resource it was withdrawn from, usually a
    collection.
    ///
    /// If the collection is not in an account's storage, from will be
    nil.
    ///
    access(all) event Withdrawn(type: String, id: UInt64, uuid: UInt64,
from: Address?, providerUUID: UInt64)
```

```

    /// Event that emitted when a token is deposited to a collection.
    /// Indicates the type, id, uuid, the owner of the collection that it
was deposited to,
    /// and the UUID of the collection it was deposited to
    ///
    /// If the collection is not in an account's storage, from, will be
nil.
    ///
access(all) event Deposited(type: String, id: UInt64, uuid: UInt64,
to: Address?, collectionUUID: UInt64)

```

These events are emitted by the Collection interface in the NonFungibleToken contract whenever the relevant function is called on any implementation.

There is also a standard NonFungibleToken.Updated event that your contract can emit if the NFT is updated in any way. This is optional though, so no need to include support for it in your implementation.

To facilitate querying, you'll also want a function to retrieve important information from the collection, like what types it supports and all the NFT IDs within a collection:

```

cadence
access(all) resource Collection: NonFungibleToken.Collection {
    // ...[withdraw code]...

    /// getIDs returns an array of the IDs that are in the collection
    access(all) view fun getIDs(): [UInt64] {
        return self.ownedNFTs.keys
    }

    /// getSupportedNFTTypes returns a list of NFT types that this
receiver accepts
    access(all) view fun getSupportedNFTTypes(): {Type: Bool} {
        let supportedTypes: {Type: Bool} = {}
        supportedTypes[Type<@FooBar.NFT>()] = true
        return supportedTypes
    }

    /// Returns whether or not the given type is accepted by the
collection
    /// A collection that can accept any type should just return true by
default
    access(all) view fun isSupportedNFTType(type: Type): Bool {
        return type == Type<@FooBar.NFT>()
    }

    // ...[createEmptyCollection code]...
}

```

Supporting NFT Metadata

The Non-Fungible Token standard also enforces that implementations provide functionality to return a set of standard views about the tokens via the ViewResolver and MetadataViews definitions.

(You will need to add these imports to your contract)

These provide developers with standard ways of representing metadata about a given token such as token symbols, images, royalties, editions, website links, and standard account paths and types that third-parties can access in a standard way.

You can see the metadata views documentation for a more thorough guide using a NFT contract as an example.

For now, you can add this code to your contract to support the important metadata:

```
cadence
// Add this import!
import "MetadataViews"

access(all) contract FooBar: NonFungibleToken {
    // ...other FooBar contract code

    access(all) resource NFT: NonFungibleToken.NFT {
        // ...other NFT code

        /// Gets a list of views specific to the individual NFT
        access(all) view fun getViews(): [Type] {
            return [
                Type<MetadataViews.Display>(),
                Type<MetadataViews.Editions>(),
                Type<MetadataViews.NFTCollectionData>(),
                Type<MetadataViews.NFTCollectionDisplay>(),
                Type<MetadataViews.Serial>()
            ]
        }

        /// Resolves a view for this specific NFT
        access(all) fun resolveView( view: Type): AnyStruct? {
            switch view {
                case Type<MetadataViews.Display>():
                    return MetadataViews.Display(
                        name: "FooBar Example Token",
                        description: "An Example NFT Contract from the
Flow NFT Guide",
                        thumbnail: MetadataViews.HTTPFile(
                            url: "Fill this in with a URL to a thumbnail
of the NFT"
                    )
            }
        }
    }
}
```

```

        case Type<MetadataViews.Editions>():
            // There is no max number of NFTs that can be minted
from this contract
            // so the max edition field value is set to nil
            let editionInfo = MetadataViews.Edition(name: "FooBar
Edition", number: self.id, max: nil)
            let editionList: [MetadataViews.Edition] =
[editionInfo]
            return MetadataViews.Editions(
                editionList
            )
        case Type<MetadataViews.Serial>():
            return MetadataViews.Serial(
                self.id
            )
        case Type<MetadataViews.NFTCollectionData>():
            return FooBar.resolveContractView(resourceType:
Type<@FooBar.NFT>(), viewType: Type<MetadataViews.NFTCollectionData>())
        case Type<MetadataViews.NFTCollectionDisplay>():
            return FooBar.resolveContractView(resourceType:
Type<@FooBar.NFT>(), viewType:
Type<MetadataViews.NFTCollectionDisplay>())
        }
    }

access(all) resource Collection: NonFungibleToken.Vault {

    // ...[getIDs code]...

    /// Allows a caller to borrow a reference to a specific NFT
    /// so that they can get the metadata views for the specific NFT
    access(all) view fun borrowNFT( id: UInt64):
&{NonFungibleToken.NFT}? {
        return &self.ownedNFTs[id]
    }

    // ...[rest of code]...
}

/// Gets a list of views for all the NFTs defined by this contract
access(all) view fun getContractViews(resourceType: Type?): [Type] {
    return [
        Type<MetadataViews.NFTCollectionData>(),
        Type<MetadataViews.NFTCollectionDisplay>()
    ]
}

/// Resolves a view that applies to all the NFTs defined by this
contract
access(all) fun resolveContractView(resourceType: Type?, viewType:
Type): AnyStruct? {
    switch viewType {

```

```

        case Type<MetadataViews.NFTCollectionData>():
            let collectionData = MetadataViews.NFTCollectionData(
                storagePath: self.CollectionStoragePath,
                publicPath: self.CollectionPublicPath,
                publicCollection: Type<&FooBar.Collection>(),
                publicLinkedType: Type<&FooBar.Collection>(),
                createEmptyCollectionFunction: (fun():
                    @NonFungibleToken.Collection {
                        return <-FooBar.createEmptyCollection(nftType:
                            Type<@FooBar.NFT>())
                    }
                )
            )
            return collectionData
        case Type<MetadataViews.NFTCollectionDisplay>():
            let media = MetadataViews.Media(
                file: MetadataViews.HTTPFile(
                    url: "Add your own SVG+XML link here"
                ),
                mediaType: "image/svg+xml"
            )
            return MetadataViews.NFTCollectionDisplay(
                name: "The FooBar Example Collection",
                description: "This collection is used as an example
to help you develop your next Flow NFT.",
                externalURL: MetadataViews.ExternalURL("Add your own
link here"),
                squareImage: media,
                bannerImage: media,
                socials: {
                    "twitter": MetadataViews.ExternalURL("Add a link
to your project's twitter")
                }
            )
        }
        return nil
    }
}

```

If you ever plan on making your NFTs more complex, you should look into adding views for Edition, EVMBridgedMetadata, Traits, and Royalties. These views make it much easier for third-party sites like marketplaces and NFT information aggregators to clearly display information about your projects on their apps and websites and are critical for every project to include if we want to have a vibrant and interoperable ecosystem.

Deploying the Contract

With your contract ready, it's time to deploy it. First, add the FooBar contract to the flow.json configuration file:

```
bash
```

```
flow config add contract
```

When prompted, enter the following name and location (press Enter to skip alias questions):

```
Enter name: FooBar
```

```
Enter contract file location: cadence/contracts/FooBar.cdc
```

Next, configure the deployment settings by running the following command:

```
bash  
flow config add deployment
```

Choose the emulator for the network and emulator-account for the account to deploy to.

Then, select the FooBar contract (you may need to scroll down).

This will update your flow.json configuration.

After that, you can select No when asked to deploy another contract.

To start the Flow emulator, run (you may need to approve a prompt to allow connection the first time):

```
bash  
flow emulator start
```

In a separate terminal or command prompt, deploy the contract:

```
bash  
flow project deploy
```

You'll then see a message that says All contracts deployed successfully.

Creating an NFTCollection

To manage multiple NFTs, you'll need an NFT collection.

Start by creating a transaction file for this purpose (we can use the generate command again):

```
bash  
flow generate transaction setupfoobarcollection
```

This creates a transaction file at cadence/transactions/setupfoobarcollection.cdc.

Transactions, on the other hand, are pieces of Cadence code that can mutate the state of the blockchain.

Transactions need to be signed by one or more accounts,

and they can have multiple phases, represented by different blocks of code.

In this file, import the necessary contracts and define a transaction to create a new collection, storing it in the account's storage. Additionally, the transaction creates a capability that allows others to get a public reference to the collection to read from its methods.

This capability ensures secure, restricted access to specific functionalities or information within a resource.

```
cadence
import "FooBar"
import "NonFungibleToken"

transaction {

    prepare(signer: auth(BorrowValue, IssueStorageCapabilityController,
PublishCapability, SaveValue, UnpublishCapability) &Account) {

        // Return early if the account already has a collection
        if signer.storage.borrow<&FooBar.Collection>(from:
FooBar.CollectionStoragePath) != nil {
            return
        }

        // Create a new empty collection
        let collection <- FooBar.createEmptyCollection(nftType:
Type<@FooBar.NFT>())

        // save it to the account
        signer.storage.save(<-collection, to:
FooBar.CollectionStoragePath)

        let collectionCap =
signer.capabilities.storage.issue<&FooBar.Collection>(FooBar.CollectionSt
oragePath)
            signer.capabilities.publish(collectionCap, at:
FooBar.CollectionPublicPath)
        }
    }
}
```

There are also examples of generic transactions that you can use to setup an account for ANY non-fungible token using metadata views! You should check those out and try to use generic transactions whenever it is possible.

To store this new NFT collection, create a new account:

```
bash
flow accounts create
```

Name it `test-acct` and select `emulator` as the network. Then, using the Flow CLI, run the transaction:

```
bash
flow transactions send cadence/transactions/setupfoobarcollection.cdc --
signer test-acct --network emulator
```

Congratulations! You've successfully created an NFT collection for the `test-acct`.

Get an Account's NFTs

To retrieve the NFTs associated with an account, you'll need a script. Scripts are read-only operations that allow you to query the blockchain. They don't modify the blockchain's state, and therefore, they don't require gas fees or signatures (read more about scripts here).

Start by creating a script file using the `generate` command again:

```
bash
flow generate script getfoobarids
```

In this script, import the necessary contracts and define a function that retrieves the NFT IDs associated with a given account:

```
cadence
import "NonFungibleToken"
import "FooBar"

access(all) fun main(address: Address): [UInt64] {
    let account = getAccount(address)

    let collectionRef =
        account.capabilities.borrow<&{NonFungibleToken.Collection}>(
            FooBar.CollectionPublicPath
        ) ?? panic("The account ".concat(address.toString()).concat(" does
not have a NonFungibleToken Collection at ")
            .concat(FooBar.CollectionPublicPath.toString())
            .concat(".").concat(" The account must initialize their account with
this collection first!"))

    return collectionRef.getIDs()
}
```

To check the NFTs associated with the `test-acct`, run the script (note: replace `0x123` with the address for `test-acct` from `flow.json`):

```
bash
flow scripts execute cadence/scripts/getfoobarids.cdc 0x123
```

Since you haven't added any NFTs to the collection yet, the result will be an empty array.

Minting and Depositing an NFT to a Collection

To mint and deposit an NFT into a collection, create a new transaction file:

```
bash
flow generate transaction mintfoobarnft
```

In this file, define a transaction that takes a recipient's address as an argument.

This transaction will borrow the minting capability from the contract account, borrow the recipient's collection capability, create a new NFT using the minter, and deposit it into the recipient's collection:

```
cadence
import "NonFungibleToken"
import "FooBar"

transaction(
    recipient: Address
) {

    /// local variable for storing the minter reference
    let minter: &FooBar.NFTMinter

    /// Reference to the receiver's collection
    let recipientCollectionRef: &{NonFungibleToken.Receiver}

    prepare(signer: auth(BorrowValue) &Account) {

        // borrow a reference to the NFTMinter resource in storage
        self.minter = signer.storage.borrow<&FooBar.NFTMinter>(from:
        FooBar.MinterStoragePath)
        ?? panic("The signer does not store a FooBar Collection
object at the path "
            .concat(FooBar.CollectionStoragePath.toString())
            .concat("The signer must initialize their account
with this collection first!"))

        // Borrow the recipient's public NFT collection reference
        self.recipientCollectionRef =
getAccount(recipient).capabilities.borrow<&{NonFungibleToken.Receiver}>(
            FooBar.CollectionPublicPath
        ) ?? panic("The account ".concat(recipient.toString()).concat("
does not have a NonFungibleToken Receiver at ")
            .concat(FooBar.CollectionPublicPath.toString()))
```

```

        .concat(". The account must initialize their account with
this collection first!"))
    }

    execute {
        // Mint the NFT and deposit it to the recipient's collection
        let mintedNFT <- self.minter.createNFT()
        self.recipientCollectionRef.deposit(token: <-mintedNFT)
    }
}

```

To run this transaction, use the Flow CLI. Remember, the contract account (which has the minting resource) should be the one signing the transaction.

Pass the test account's address (from the flow.json file) as the recipient argument

(note: replace 0x123 with the address for test-acct from flow.json):

```

bash
flow transactions send cadence/transactions/mintfoobarnft.cdc 0x123 --
signer emulator-account --network emulator

```

After executing the transaction, you can run the earlier script to verify that the NFT was added to the test-acct's collection (remember to replace 0x123):

```

bash
flow scripts execute cadence/scripts/getfoobarids.cdc 0x123

```

You should now see a value in the test-acct's collection array!

Transferring an NFT to Another Account

To transfer an NFT to another account, create a new transaction file using generate:

```

bash
flow generate transaction transferfoobarnft

```

In this file, define a transaction that takes a recipient's address and the ID of the NFT you want to transfer as arguments. This transaction will borrow the sender's collection, get the recipient's capability, withdraw the NFT from the sender's collection, and deposit it into the recipient's collection:

```

cadence
import "FooBar"
import "NonFungibleToken"

```

```

transaction(recipient: Address, withdrawID: UInt64) {

    /// Reference to the withdrawer's collection
    let withdrawRef: auth(NonFungibleToken.Withdraw)
    &{NonFungibleToken.Collection}

    /// Reference of the collection to deposit the NFT to
    let receiverRef: &{NonFungibleToken.Receiver}

    prepare(signer: auth(BorrowValue) &Account) {

        // borrow a reference to the signer's NFT collection
        self.withdrawRef =
signer.storage.borrow<auth(NonFungibleToken.Withdraw)
&{NonFungibleToken.Collection}>(
            from: FooBar.CollectionStoragePath
        ) ?? panic("The signer does not store a FooBar Collection
object at the path "
            .concat(FooBar.CollectionStoragePath.toString())
            .concat("The signer must initialize their account
with this collection first!"))

        // get the recipients public account object
        let recipient = getAccount(recipient)

        // borrow a public reference to the receivers collection
        let receiverCap =
recipient.capabilities.get<&{NonFungibleToken.Receiver}>(FooBar.Collectio
nPublicPath)

        self.receiverRef = receiverCap.borrow()
        ?? panic("The account ".concat(recipient.toString()).concat(")
does not have a NonFungibleToken Receiver at ")
            .concat(FooBar.CollectionPublicPath.toString())
            .concat(". The account must initialize their account with
this collection first!"))
    }

    execute {
        let nft <- self.withdrawRef.withdraw(withdrawID: withdrawID)
        self.receiverRef.deposit(token: <-nft)
    }
}

```

To transfer the NFT, first create a new account:

```

bash
flow accounts create

```

Name it test-acct-2 and select Emulator as the network. Next, create a collection for this new account:

```
bash
flow transactions send cadence/transactions/setupfoobarcollection.cdc --
signer test-acct-2 --network emulator
```

Now, run the transaction to transfer the NFT from test-acct to test-acct-2 using the addresses from the flow.json file (replace 0x124 with test-acct-2's address). Also note that 0 is the id of the NFT we'll be transferring):

```
bash
flow transactions send cadence/transactions/transferfoobarNFT.cdc 0x124 0
--signer test-acct --network emulator
```

To verify the transfer, you can run the earlier script for test-acct-2 (replace 0x124):

```
bash
flow scripts execute cadence/scripts/getfoobarids.cdc 0x123
```

The transfer transaction also has a generic version that developers are encouraged to use!

Congrats, you did it! You're now ready to launch the next fun NFT project on Flow.

More

- Explore an example NFT repository
- Dive into the details of the NFT Standard
- For a deeper dive into MetadataViews, consult the introduction guide or the FLIP that introduced this feature.
- Use a no code tool for creating NFT projects on Flow

```
# child-accounts.md:
```

```
---
title: Building Walletless Applications Using Child Accounts
sidebarposition: 1
---
```

In this doc, we'll dive into a progressive onboarding flow, including the Cadence scripts & transactions that go into its implementation in your app. These components will enable any implementing app to create a custodial account, mediate the user's onchain actions on their behalf, and later delegate access of that app-created account to the user's wallet. We'll refer to this custodial pattern as the Hybrid Custody Model and the process of delegating control of the app

account as Account Linking.

Objectives

- Create a walletless onboarding transaction
- Link an existing app account as a child to a newly authenticated parent account
- Get your app to recognize "parent" accounts along with any associated "child" accounts
- Put it all together to create a blockchain-native onboarding transaction
- View fungible and non-fungible Token metadata relating to assets across all of a user's associated accounts - their wallet-mediated "parent" account and any "child" accounts
- Facilitate transactions acting on assets in child accounts

Point of Clarity

Before diving in, let's make a distinction between "account linking" and "linking accounts".

Account Linking

:::info

Note that since account linking is a sensitive action, transactions where an account may be linked are designated by a topline pragma `#allowAccountLinking`. This lets wallet providers inform users that their account may be linked in the signed transaction.

:::

Very simply, account linking is a feature in Cadence that let's an Account create a Capability on itself.

Below is an example demonstrating how to issue an &Account Capability from a signing account

transaction:

```
cadence linkaccount.cdc
#allowAccountLinking

transaction(linkPathSuffix: String) {
    prepare(signer: auth(IssueAccountCapabilityController) &Account) {
        // Issues a fully-entitled Account Capability
        let accountCapability = signer.capabilities
            .account
            .issue<auth(Storage, Contracts, Keys, Inbox, Capabilities)
&Account>()
    }
}
```

}

From there, the signing account can retrieve the privately linked &Account Capability and delegate it to another account, revoking the Capability if they wish to revoke delegated access.

Note that in order to link an account, a transaction must state the #allowAccountLinking pragma in the top line of the transaction. This is an interim safety measure so that wallet providers can notify users they're about to sign a transaction that may create a Capability on their Account.

Linking Accounts

Linking accounts leverages this account link, otherwise known as an &Account Capability, and encapsulates it. The components and actions involved in this process - what the Capability is encapsulated in, the collection that holds those encapsulations, etc. is what we'll dive into in this doc.

Terminology

Parent-Child accounts - For the moment, we'll call the account created by the app the "child" account and the account receiving its &Account Capability the "parent" account. Existing methods of account access & delegation (i.e. keys) still imply ownership over the account, but insofar as linked accounts are concerned, the account to which both the user and the app share access via &Account Capability will be considered the "child" account.

Walletless onboarding - An onboarding flow whereby an app creates a custodial account for a user, onboarding them to the app, obviating the need for user wallet authentication.

Blockchain-native onboarding - Similar to the already familiar Web3 onboarding flow where a user authenticates with their existing wallet, an app onboards a user via wallet authentication while additionally creating a custodial app account and linking it with the authenticated account, resulting in a "hybrid custody" model.

Hybrid Custody Model - A custodial pattern in which an app and a user maintain access to an app created account and user access to that account has been mediated via account linking.

Account Linking - Technically speaking, account linking in our context consists of giving some other account an &Account Capability from the granting account. This Capability is maintained in standardized resource called a HybridCustody.Manager, providing its owning user access to any and all of their linked accounts.

Progressive Onboarding - An onboarding flow that walks a user up to self-custodial ownership, starting with walletless onboarding and later linking the app account with the user's authenticated wallet once the user chooses to do so.

Restricted Child Account - An account delegation where the access on the delegating account is restricted according to rules set by the linking child account. The distinctions between this and the subsequent term ("owned" account) will be expanding on later.

Owned Account - An account delegation where the delegatee has unrestricted access on the delegating child account, thereby giving the delegatee presiding authority superseding any other "restricted" parent accounts.

Account Linking

Linking an account is the process of delegating account access via &Account Capability. Of course, we want to do this in a way that allows the receiving account to maintain that Capability and allows easy identification of the accounts on either end of the linkage - the user's main "parent" account and the linked "child" account. This is accomplished in the HybridCustody contract which we'll continue to use in this guidance.

Prerequisites

Since account delegation is mediated by developer-defined rules, you should make sure to first configure the resources that contain those rules. Contracts involved in defining and enforcing this ruleset are CapabilityFilter and CapabilityFactory. The former enumerates those types that are and are not accessible from a child account while the latter enables the access of those allowable Capabilities such that the returned values can be properly typed - e.g. retrieving a Capability that can be cast to Capability<&NonFungibleToken.Collection> for example.

Here's how you would configure an AllowlistFilter and add allowed types to it:

```
cadence setupallowallfilter.cdc
import "CapabilityFilter"

transaction(identifiers: [String]) {
    prepare(acct: auth(BorrowValue, SaveValue, StorageCapabilities,
    PublishCapability, UnpublishCapability) &Account) {
        // Setup the AllowlistFilter
        if acct.storage.borrow<&AnyResource>(from:
    CapabilityFilter.StoragePath) == nil {
            acct.storage.save(
```

```

        <-
CapabilityFilter.createFilter(Type<@CapabilityFilter.AllowlistFilter>()),
    to: CapabilityFilter.StoragePath)
}

// Ensure the AllowlistFilter is linked to the expected
PublicPath
    acct.capabilities.unpublish(CapabilityFilter.PublicPath)
    acct.capabilities.publish()

acct.capabilities.storage.issue<&{CapabilityFilter.Filter}>(CapabilityFil
ter.StoragePath),
    at: CapabilityFilter.PublicPath
)

// Get a reference to the filter
let filter = acct.storage.borrow<auth(CapabilityFilter.Add)
&CapabilityFilter.AllowlistFilter>(
    from: CapabilityFilter.StoragePath
) ?? panic("filter does not exist")

// Add the given type identifiers to the AllowlistFilter
// Note: the whole transaction fails if any of the given
identifiers are malformed
for identifier in identifiers {
    let c = CompositeType(identifier) !
    filter.addType(c)
}
}
}
}

```

And the following transaction configures a CapabilityFactory.Manager, adding NFT-related Factory objects:

:::info

Note that the Manager configured here enables retrieval of castable Capabilities. It's recommended that you implement Factory resource definitions to support any NFT Collections related with the use of your application so that users can retrieve Typed Capabilities from accounts linked from your app.

:::

```

cadence setupfactory.cdc
import "NonFungibleToken"

import "CapabilityFactory"
import "NFTCollectionPublicFactory"
import "NFTProviderAndCollectionFactory"
import "NFTProviderFactory"
import "NFTCollectionFactory"

```

```

transaction {
    prepare(acct: auth(BorrowValue, SaveValue, StorageCapabilities,
PublishCapability, UnpublishCapability) &Account) {
        // Check for a stored Manager, saving if not found
        if acct.storage.borrow<&AnyResource>(from:
CapabilityFactory.StoragePath) == nil {
            let f <- CapabilityFactory.createFactoryManager()
            acct.storage.save(<-f, to: CapabilityFactory.StoragePath)
        }

        // Check for Capabilities where expected, linking if not found
        acct.capabilities.unpublish(CapabilityFactory.PublicPath)
        acct.capabilities.publish()

        acct.capabilities.storage.issue<&CapabilityFactory.Manager>(CapabilityFac-
tory.StoragePath),
            at: CapabilityFactory.PublicPath
        )

        assert(
            acct.capabilities.get<&CapabilityFactory.Manager>(CapabilityFactory.Publi-
cPath).check(),
                message: "CapabilityFactory is not setup properly"
        )

        let manager = acct.storage.borrow<auth(CapabilityFactory.Add)>
&CapabilityFactory.Manager>(from: CapabilityFactory.StoragePath)
            ?? panic("manager not found")

        /// Add generic NFT-related Factory implementations to enable
        castable Capabilities from this Manager

        manager.updateFactory(Type<&{NonFungibleToken.CollectionPublic}>(),
NFTCollectionPublicFactory.Factory())
            manager.updateFactory(Type<auth(NonFungibleToken.Withdraw)>
&{NonFungibleToken.Provider, NonFungibleToken.CollectionPublic}>(),
NFTProviderAndCollectionFactory.Factory())
            manager.updateFactory(Type<auth(NonFungibleToken.Withdraw)>
&{NonFungibleToken.Provider}>(), NFTProviderFactory.Factory())
            manager.updateFactory(Type<auth(NonFungibleToken.Withdraw)>
&{NonFungibleToken.Collection}>(),
NFTCollectionFactory.WithdrawFactory())
            manager.updateFactory(Type<&{NonFungibleToken.Collection}>(),
NFTCollectionFactory.Factory())
        }
    }
}

!resources/hybridcustodyhighlevel

```

In this scenario, a user custodies a key for their main account which maintains access to a wrapped Account

Capability, providing the user restricted access on the app account. The app maintains custodial access to the account and regulates the access restrictions to delegatee "parent" accounts.

Linking accounts can be done in one of two ways. Put simply, the child account needs to get the parent an Account Capability, and the parent needs to save that Capability so they can retain access. This delegation must be done manner that represents each side of the link while safeguarding the integrity of any access restrictions an application puts in place on delegated access.

We can achieve issuance from the child account and claim from the parent account pattern by either:

1. Leveraging Cadence's Account.Inbox to publish the Capability from the child account & have the parent claim the Capability in a subsequent transaction.
2. Executing a multi-party signed transaction, signed by both the child and parent accounts.

Let's take a look at both.

:::info

You'll want to consider whether you would like the parent account to be configured with some app-specific resources or Capabilities and compose your multisig or claim transactions to include such configurations.

For example, if your app deals with specific NFTs, you may want to configure the parent account with Collections for those NFTs so the user can easily transfer them between their linked accounts.

:::

Publish & Claim

Publish

Here, the account delegating access to itself links its &Account Capability, and publishes it to be claimed by the designated parent account.

```
cadence publishtoparent.cdc
import "HybridCustody"
import "CapabilityFactory"
import "CapabilityFilter"
import "CapabilityDelegator"

transaction(parent: Address, factoryAddress: Address, filterAddress:
Address) {
    prepare(acct: auth(BorrowValue) &Account) {
```

```

    // NOTE: The resources and Capabilities needed for this
    transaction are assumed to have be pre-configured

    // Borrow the OwnedAccount resource
    let owned = acct.storage.borrow<auth(HybridCustody.Owner)>
&HybridCustody.OwnedAccount>(
        from: HybridCustody.OwnedAccountStoragePath
    ) ?? panic("owned account not found")

    // Get a CapabilityFactory.Manager Capability
    let factory = getAccount(factoryAddress).capabilities
        .get<&CapabilityFactory.Manager>(
            CapabilityFactory.PublicPath
        )
    assert(factory.check(), message: "factory address is not
configured properly")

    // Get a CapabilityFilter.Filter Capability
    let filter = getAccount(filterAddress).capabilities
        .get<&{CapabilityFilter.Filter}>(
            CapabilityFilter.PublicPath
        )
    assert(filter.check(), message: "capability filter is not
configured properly")

    // Publish the OwnedAccount to the designated parent account
    owned.publishToParent(parentAddress: parent, factory: factory,
filter: filter)
}
}

```

Claim

On the other side, the receiving account claims the published ChildAccount Capability, adding it to the signer's HybridCustody.Manager.childAccounts indexed on the child account's Address.

```

cadence redeemaccount.cdc
import "MetadataViews"
import "ViewResolver"

import "HybridCustody"
import "CapabilityFilter"

transaction(childAddress: Address, filterAddress: Address?, filterPath:
PublicPath?) {
    prepare(acct: auth(Storage, Capabilities, Inbox) &Account) {
        // Get a Manager filter if a path is provided
        var filter: Capability<&{CapabilityFilter.Filter}>? = nil
        if filterAddress != nil && filterPath != nil {
            filter = getAccount(filterAddress!).capabilities
                .get<&{CapabilityFilter.Filter}>(

```

```

        filterPath!
    )
}

// Configure a Manager if not already configured
if acct.storage.borrow<&HybridCustody.Manager>(from:
HybridCustody.ManagerStoragePath) == nil {
    let m <- HybridCustody.createManager(filter: filter)
    acct.storage.save(<- m, to: HybridCustody.ManagerStoragePath)

    for c in acct.capabilities.storage.getControllers(forPath:
HybridCustody.ManagerStoragePath) {
        c.delete()
    }

    acct.capabilities.unpublish(HybridCustody.ManagerPublicPath)

    acct.capabilities.publish(
        acct.capabilities.storage.issue<&{HybridCustody.ManagerPublic}>(
            HybridCustody.ManagerStoragePath
        ),
        at: HybridCustody.ManagerPublicPath
    )

    acct.capabilities
        .storage
        .issue<auth(HybridCustody.Manage)
&{HybridCustody.ManagerPrivate, HybridCustody.ManagerPublic}>(
            HybridCustody.ManagerStoragePath
        )
    }
}

// Claim the published ChildAccount Capability
let inboxName =
HybridCustody.getChildAccountIdentifier(acct.address)
let cap = acct.inbox.claim<auth(HybridCustody.Child)
&{HybridCustody.AccountPrivate, HybridCustody.AccountPublic,
ViewResolver.Resolver}>(inboxName, provider: childAddress)
?? panic("child account cap not found")

// Get a reference to the Manager and add the account & add the
child account
let manager = acct.storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(from: HybridCustody.ManagerStoragePath)
?? panic("manager no found")
manager.addAccount(cap: cap)
}
}

```

Multi-Signed Transaction

We can combine the two transactions in Publish and Claim into a single multi-signed transaction to achieve Hybrid Custody in a single step.

:::info

Note that while the following code links both accounts in a single transaction, in practicality you may find it easier to execute publish and claim transactions separately depending on your custodial infrastructure.

:::

```
cadence setupmultisig.cdc
#allowAccountLinking

import "HybridCustody"

import "CapabilityFactory"
import "CapabilityDelegator"
import "CapabilityFilter"

import "MetadataViews"
import "ViewResolver"

transaction(parentFilterAddress: Address?, childAccountFactoryAddress:
Address, childAccountFilterAddress: Address) {
    prepare(childAcct: auth(Storage, Capabilities) &Account, parentAcct:
auth(Storage, Capabilities, Inbox) &Account) {
        // ----- Begin setup of child account -----
        -----
        var optCap: Capability<auth(Storage, Contracts, Keys, Inbox,
Capabilities) &Account>? = nil
        let t = Type<auth(Storage, Contracts, Keys, Inbox, Capabilities)
&Account>()
        for c in childAcct.capabilities.account.getControllers() {
            if c.borrowType.isSubtype(of: t) {
                optCap = c.capability as! Capability<auth(Storage,
Contracts, Keys, Inbox, Capabilities) &Account>
                break
            }
        }
        if optCap == nil {
            optCap = childAcct.capabilities.account.issue<auth(Storage,
Contracts, Keys, Inbox, Capabilities) &Account>()
        }
        let acctCap = optCap ?? panic("failed to get account capability")

        if childAcct.storage.borrow<&HybridCustody.OwnedAccount>(from:
HybridCustody.OwnedAccountStoragePath) == nil {
            let ownedAccount <- HybridCustody.createOwnedAccount(acct:
acctCap)
```

```

        childAcct.storage.save(<-ownedAccount, to:
HybridCustody.OwnedAccountStoragePath)
    }

    for c in childAcct.capabilities.storage.getControllers(forPath:
HybridCustody.OwnedAccountStoragePath) {
        c.delete()
    }

    // configure capabilities

childAcct.capabilities.storage.issue<&{HybridCustody.BorrowableAccount,
HybridCustody.OwnedAccountPublic,
ViewResolver.Resolver}>(HybridCustody.OwnedAccountStoragePath)
    childAcct.capabilities.publish()

childAcct.capabilities.storage.issue<&{HybridCustody.OwnedAccountPublic,
ViewResolver.Resolver}>(HybridCustody.OwnedAccountStoragePath),
    at: HybridCustody.OwnedAccountPublicPath
)

// ----- End setup of child account -----
-----

// ----- Begin setup of parent account -----
-----

var filter: Capability<&{CapabilityFilter.Filter}>? = nil
if parentFilterAddress != nil {
    filter =
getAccount(parentFilterAddress!).capabilities.get<&{CapabilityFilter.Filter}>(CapabilityFilter.PublicPath)
}

if parentAcct.storage.borrow<&{HybridCustody.Manager}>(from:
HybridCustody.ManagerStoragePath) == nil {
    let m <- HybridCustody.createManager(filter: filter)
    parentAcct.storage.save(<- m, to:
HybridCustody.ManagerStoragePath)
}

for c in parentAcct.capabilities.storage.getControllers(forPath:
HybridCustody.ManagerStoragePath) {
    c.delete()
}

parentAcct.capabilities.publish()

parentAcct.capabilities.storage.issue<&{HybridCustody.ManagerPublic}>(Hyb
ridCustody.ManagerStoragePath),
    at: HybridCustody.ManagerPublicPath
)
parentAcct.capabilities.storage.issue<auth(HybridCustody.Manage)>
&{HybridCustody.ManagerPrivate,
HybridCustody.ManagerPublic}>(HybridCustody.ManagerStoragePath)

```

```

// ----- End setup of parent account -----
-----

// Publish account to parent
let owned = childAcct.storage.borrow<auth(HybridCustody.Owner)
&HybridCustody.OwnedAccount>(from: HybridCustody.OwnedAccountStoragePath)
?? panic("owned account not found")

let factory =
getAccount(childAccountFactoryAddress).capabilities.get<&CapabilityFactor
y.Manager>(CapabilityFactory.PublicPath)
assert(factory.check(), message: "factory address is not
configured properly")

let filterForChild =
getAccount(childAccountFilterAddress).capabilities.get<&{CapabilityFilter
.Filter}>(CapabilityFilter.PublicPath)
assert(filterForChild.check(), message: "capability filter is not
configured properly")

owned.publishToParent(parentAddress: parentAcct.address, factory:
factory, filter: filterForChild)

// claim the account on the parent
let inboxName =
HybridCustody.getChildAccountIdentifier(parentAcct.address)
let cap = parentAcct.inbox.claim<auth(HybridCustody.Child)
&{HybridCustody.AccountPrivate, HybridCustody.AccountPublic,
ViewResolver.Resolver}>(inboxName, provider: childAcct.address)
?? panic("child account cap not found")

let manager =
parentAcct.storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(from: HybridCustody.ManagerStoragePath)
?? panic("manager no found")

manager.addAccount(cap: cap)
}
}

```

Onboarding Flows

Given the ability to establish an account and later delegate access to a user, apps are freed from the constraints of dichotomous custodial & self-custodial paradigms. A developer can choose to onboard a user via traditional Web2 identity and later delegate access to the user's wallet account. Alternatively, an app can enable wallet authentication at the outset, creating an app-specific account & linking with the user's wallet account. As specified above, these two flows are known as "walletless" and "blockchain-native" onboarding respectively. Developers can choose to implement one for

simplicity or both for maximum flexibility.

Walletless Onboarding

The following transaction creates an account, funding creation via the signer and adding the provided public key. You'll notice this transaction is pretty much your standard account creation. The magic for you will be how you custody the key for this account (locally, KMS, wallet service, etc.) in a manner that allows your app to mediate onchain interactions on behalf of your user.

```
cadence walletlessonboarding
import "FungibleToken"
import "FlowToken"

transaction(pubKey: String, initialFundingAmt: UFix64) {

    prepare(signer: auth(BorrowValue) &Account) {

        / --- Account Creation --- /
        // NOTE: your app may choose to separate creation depending
        on your custodial model)
        //
        // Create the child account, funding via the signer
        let newAccount = Account(payer: signer)
        // Create a public key for the new account from string value
        in the provided arg
        // NOTE: You may want to specify a different signature algo
        for your use case
        let key = PublicKey(
            publicKey: pubKey.decodeHex(),
            signatureAlgorithm: SignatureAlgorithm.ECDSAP256
        )
        // Add the key to the new account
        // NOTE: You may want to specify a different hash algo &
        weight best for your use case
        newAccount.keys.add(
            publicKey: key,
            hashAlgorithm: HashAlgorithm.SHA3256,
            weight: 1000.0
        )

        / --- (Optional) Additional Account Funding --- /
        //
        // Fund the new account if specified
        if initialFundingAmt > 0.0 {
            // Get a vault to fund the new account
            let fundingProvider =
signer.storage.borrow<auth(FungibleToken.Withdraw) &FlowToken.Vault>(
                from: /storage/flowTokenVault
            )!
            // Fund the new account with the initialFundingAmount
specified
    }
}
```

```

        let receiver =
newAccount.capabilities.get<&FlowToken.Vault>(
            /public/flowTokenReceiver
        ).borrow()!
        let fundingVault <-fundingProvider.withdraw(
            amount: initialFundingAmt
        )
        receiver.deposit(from: <-fundingVault)
    }

    / --- Continue with use case specific setup --- /
    //
    // At this point, the newAccount can further be configured as
suitable for
        // use in your app (e.g. Setup a Collection, Mint NFT,
Configure Vault, etc.)
        // ...
    }
}

```

Blockchain-Native Onboarding

This onboarding flow is really a single-transaction composition of the steps covered above. This is a testament to the power of the complex transactions you can compose on Flow with Cadence!

:::info

Recall the prerequisites needed to be satisfied before linking an account:

1. CapabilityFilter Filter saved and linked
2. CapabilityFactory Manager saved and linked as well as Factory implementations supporting the Capability Types you'll want accessible from linked child accounts as Typed Capabilities.

:::

Account Creation & Linking

Compared to walletless onboarding where a user does not have a Flow account, blockchain-native onboarding assumes a user already has a wallet configured and immediately links it with a newly created app account. This enables the app to sign transactions on the user's behalf via the new child account while immediately delegating control of that account to the onboarding user's main account.

After this transaction, both the custodial party (presumably the client/app) and the signing parent account will have access to the newly created account - the custodial party via key access and the parent account via their

HybridCustody.Manager maintaining the new account's ChildAccount Capability.

```
cadence blockchainnativeonboarding.cdc
#allowAccountLinking

import "FungibleToken"
import "FlowToken"
import "MetadataViews"
import "ViewResolver"

import "HybridCustody"
import "CapabilityFactory"
import "CapabilityFilter"
import "CapabilityDelegator"

transaction(
    pubKey: String,
    initialFundingAmt: UFix64,
    factoryAddress: Address,
    filterAddress: Address
) {

    prepare(parent: auth(Storage, Capabilities, Inbox) &Account, app:
auth(Storage, Capabilities) &Account) {
        / --- Account Creation --- /
        //
        // Create the child account, funding via the signing app account
        let newAccount = Account(payer: app)
        // Create a public key for the child account from string value in
the provided arg
        // NOTE: You may want to specify a different signature algo for
your use case
        let key = PublicKey(
            publicKey: pubKey.decodeHex(),
            signatureAlgorithm: SignatureAlgorithm.ECDSSAP256
        )
        // Add the key to the new account
        // NOTE: You may want to specify a different hash algo & weight
best for your use case
        newAccount.keys.add(
            publicKey: key,
            hashAlgorithm: HashAlgorithm.SHA3256,
            weight: 1000.0
        )

        / --- (Optional) Additional Account Funding --- /
        //
        // Fund the new account if specified
        if initialFundingAmt > 0.0 {
            // Get a vault to fund the new account
            let fundingProvider =
app.storage.borrow<auth(FungibleToken.Withdraw)
&{FungibleToken.Provider}>(from: /storage/flowTokenVault)!
```

```

        // Fund the new account with the initialFundingAmount
specified

newAccount.capabilities.get<&{FungibleToken.Receiver}>(/public/flowTokenR
eceiver) !
    .borrow() !
    .deposit(
        from: <-fundingProvider.withdraw(
            amount: initialFundingAmt
        )
    )
}

/ Continue with use case specific setup /
// At this point, the newAccount can further be configured as
suitable for
// use in your dapp (e.g. Setup a Collection, Mint NFT, Configure
Vault, etc.)
// ...

/ --- Link the AuthAccount Capability --- /
let acctCap = newAccount.capabilities.account.issue<auth(Storage,
Contracts, Keys, Inbox, Capabilities) &Account>()

// Create a OwnedAccount & link Capabilities
let ownedAccount <- HybridCustody.createOwnedAccount(acct:
acctCap)
    newAccount.storage.save(<-ownedAccount, to:
HybridCustody.OwnedAccountStoragePath)

newAccount.capabilities.storage.issue<&{HybridCustody.BorrowableAccount,
HybridCustody.OwnedAccountPublic,
ViewResolver.Resolver}>(HybridCustody.OwnedAccountStoragePath)
    newAccount.capabilities.publish()

newAccount.capabilities.storage.issue<&{HybridCustody.OwnedAccountPublic,
ViewResolver.Resolver}>(HybridCustody.OwnedAccountStoragePath),
    at: HybridCustody.OwnedAccountPublicPath
)

// Get a reference to the OwnedAccount resource
let owned = newAccount.storage.borrow<auth(HybridCustody.Owner)
&HybridCustody.OwnedAccount>(from:
HybridCustody.OwnedAccountStoragePath) !

// Get the CapabilityFactory.Manager Capability
let factory =
getAccount(factoryAddress).capabilities.get<&CapabilityFactory.Manager>(C
apabilityFactory.PublicPath)
    assert(factory.check(), message: "factory address is not
configured properly")

```

```

    // Get the CapabilityFilter.Filter Capability
    let filter =
getAccount(filterAddress).capabilities.get<&{CapabilityFilter.Filter}>(Ca
pabilityFilter.PublicPath)
        assert(filter.check(), message: "capability filter is not
configured properly")

    // Configure access for the delegatee parent account
    owned.publishToParent(parentAddress: parent.address, factory:
factory, filter: filter)

    / --- Add delegation to parent account --- /
    //
    // Configure HybridCustody.Manager if needed
    if parent.storage.borrow<&AnyResource>(from:
HybridCustody.ManagerStoragePath) == nil {
        let m <- HybridCustody.createManager(filter: filter)
        parent.storage.save(<- m, to:
HybridCustody.ManagerStoragePath)

        for c in parent.capabilities.storage.getControllers(forPath:
HybridCustody.ManagerStoragePath) {
            c.delete()
        }

    // configure Capabilities

parent.capabilities.storage.issue<&{HybridCustody.ManagerPrivate,
HybridCustody.ManagerPublic}>(HybridCustody.ManagerStoragePath)
        parent.capabilities.publish()

parent.capabilities.storage.issue<&{HybridCustody.ManagerPublic}>(HybridC
ustody.ManagerStoragePath),
            at: HybridCustody.ManagerPublicPath
        )
    }

    // Claim the ChildAccount Capability
    let inboxName =
HybridCustody.getChildAccountIdentifier(parent.address)
    let cap = parent
        .inbox
        .claim<auth(HybridCustody.Child)
&{HybridCustody.AccountPrivate, HybridCustody.AccountPublic,
ViewResolver.Resolver}>(
            inboxName,
            provider: newAccount.address
        ) ?? panic("child account cap not found")

    // Get a reference to the Manager and add the account
    let managerRef = parent.storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(from: HybridCustody.ManagerStoragePath)

```

```
    ?? panic("manager not found")
    managerRef.addAccount(cap: cap)
}
}
```

Funding & Custody Patterns

Aside from implementing onboarding flows & account linking, you'll want to also consider the account funding & custodial pattern appropriate for the app you're building. The only pattern compatible with walletless onboarding (and therefore the only one showcased above) is one in which the app custodies the child account's key and funds account creation.

In general, the funding pattern for account creation will determine to some extent the backend infrastructure needed to support your app and the onboarding flow your app can support. For example, if you want to create a service-less client (a totally local app without backend infrastructure), you could forego walletless onboarding in favor of a user-funded blockchain-native onboarding to achieve a hybrid custody model. Your app maintains the keys to the app account locally to sign on behalf of the user, and the user funds the creation of the account, linking to their main account on account creation. This would be a user-funded, app custodied pattern.

Again, custody may deserve some regulatory insight depending on your jurisdiction. If building for production, you'll likely want to consider these non-technical implications in your technical decision-making. Such is the nature of building in crypto.

Here are the patterns you might consider:

App-Funded, App-Custodied

If you want to implement walletless onboarding, you can stop here as this is the only compatible pattern. In this scenario, a backend app account funds the creation of a new account and the app custodies the key for said account either on the user's device or some backend KMS.

App-Funded, User-Custodied

In this case, the backend app account funds account creation, but adds a key to the account which the user custodies. In order for the app to act on the user's behalf, it has to be delegated access via `&Account Capability` which the backend app account would maintain in a `HybridCustody.Manager`. This means that the new account would have two parent accounts - the user's and the app.

While this pattern provides the user maximum ownership and authority over the child account, it may present unique considerations and edge cases for you as a builder depending on your app's access to the child account. Also note that this and the following patterns are incompatible with walletless onboarding in that the user must have a wallet pre-configured before onboarding.

User-Funded, App-Custodied

As mentioned above, this pattern unlocks totally service-less architectures – just a local client & smart contracts. An authenticated user signs a transaction creating an account, adding the key provided by the client, and linking the account as a child account. At the end of the transaction, hybrid custody is achieved and the app can sign with the custodied key on the user's behalf using the newly created account.

User-Funded, User-Custodied

While perhaps not useful for most apps, this pattern may be desirable for advanced users who wish to create a shared access account themselves. The user funds account creation, adding keys they custody, and delegates secondary access to some other account.

```
# index.md:
```

```
---
```

```
title: Account Linking (FLIP 72)
```

```
sidebarposition: 4
```

```
---
```

Account Linking

Account linking is a unique Flow concept that enables sharing ownership over accounts. In order to understand how we can achieve that we must first understand how accounts on Flow are accessed.

Accounts on flow can be accessed in Cadence through two types, PublicAccount and Account. As the name implies the PublicAccount type gives access to all public account information such as address, balance, storage capacity, etc., but doesn't allow changes to the account. The Account type (or more specifically, an entitled &Account) allows the same access as PublicAccount but also allows changes to the account, including adding/revoking account keys, managing the deployed contracts, as well as linking and publishing Capabilities.

```
!Flow account structure
```

Accessing Account

Accessing Account allows for modification to account storage, so it's essential to safeguard this access by mandating that transactions are signed by the account being accessed. Account entitlements enable for more granular access control over the specific parts of the account that can be accessed from within the signed transaction. A transaction can list multiple authorizing account it wants to access as part of the prepare section of the transaction. Read more about transaction signing in the transaction documentation.

Since access to the Account object enables state change, the idea of account ownership actually translates to the ability to access the underlying account. Traditionally, you might consider this the same as having key access on an account, but we'll see in just a minute how programmatic, ownership-level access is unlocked with Capabilities on Flow.

Account Capabilities

Before proceeding the reader will need a clear understanding of Cadence capabilities to follow this section. Advanced features such as Account Capabilities are powerful but if used incorrectly can put your app or users at risk.

Cadence allows the creation of Capabilities to delegate access to account storage, meaning any account obtaining a valid Capability to another account object in the storage can access it. This is a powerful feature on its own - accessing another account programmatically without the need for an active key on the accessible account. The access to the object can be limited when creating a Capability so only intended functions or fields can be accessed.

Account linking is made possible by the extension of Capabilities on the Account object itself. Similar to how storage capabilities allow access to a value stored in an account's storage, &Account Capabilities allow delegated access to the issuing Account. These Capabilities allow for access to key assignment, contract deployment, and other privileged actions on the delegating Account - effectively sharing ownership of the account without ever adding or sharing a key. This Capability can of course be revoked at any time by the delegating account.

Creating Account Links

When referring to 'account linking' we mean that an &Account Capability is created by the parent account and published to another account. The account owning the &Account Capability which was made available to another account is the child

account. The account in possession of the Capability given by the child account becomes its parent account.

!Account linking on Flow relational diagram

A link between two existing accounts on Flow can be created in two steps:

1. A child account creates an &Account Capability and publishes it to the parent account.
2. The parent account, claims that Capability and can access the child's account through it.

!Account linking steps on Flow

These two steps are implemented in Cadence as two transactions:

Create capability

The account B creates and publishes the &Account Capability to the account A at the address 0x01

```
cadence
#allowAccountLinking

transaction {
    prepare(signer: auth(IssueAccountCapabilityController,
PublishInboxCapability) &Account) {
        // Issue a fully-entitled account capability
        let capability = signer.capabilities
            .account
            .issue<auth(Storage, Contracts, Keys, Inbox, Capabilities)
&Account>()
        // Publish the capability for the specified recipient
        signer.inbox.publish(capability, name: "accountCapA", recipient:
0x1)
    }
}
```

Claim capability

The account A claims the Capability published by account B.

```
cadence
transaction {
    prepare(signer: auth(ClaimInboxCapability) &Account) {
        let capabilityName = "accountCapB"
        let providerAddress = 0x2
        // Claim the capability published by the account 0x2
        let capability = signer.inbox
            .claim<auth(Storage, Contracts, Keys, Inbox, Capabilities)
&Account>(
                capabilityName,
                provider: providerAddress
```

```

    ) ?? panic(
        "Capability with name ".concat(capabilityName)
        .concat(" from provider
") .concat(providerAddress.toString())
        .concat(" not found")
    )
    // Simply borrowing an Account reference here for demonstration
purposes
    let accountRef = capability.borrow() !
}
}

```

What is account linking most useful for?

Account linking was specifically designed to enable smooth and seamless custodial onboarding of users to your Flow based application without them first requiring a wallet to do so. This pattern overcomes both the technical hurdle, as well as user's reluctance to install a wallet, opening access to Flow applications to every user. Users can experience an app without any delay while still offering a path to self-sovereign ownership.

Naturally, users may expect to use their account with another application, or otherwise move assets stored in that account elsewhere - at minimum from their wallet. When an app initially leverages account linking, the app creates the account instead of the user and stores that user's specific state in the app-created account. At a later point, users can take ownership of the app account providing they possess a full Flow account, typically by installing a wallet app.

Account linking enables users to possess multiple linked child accounts from different apps. Complexities associated with accessing those child accounts are eliminated by abstracting access to them through the user's parent account.

:::info

Simply put, child accounts are accessed and can be treated as a seamless part of the parent account.

:::

All assets in the app account can now jump the walled garden to play in the rest of the Flow ecosystem. The user does not need to rely on the custodial app to execute transactions moving assets from the child account as the parent account already has access to the assets in the child account.

!Multiple parent-child accounts on Flow

This shared control over the digital items in the in-app account enables users to establish real ownership of the items beyond the context of the app, where they can use their parent account to view inventory, take the items to other apps in the ecosystem, such as a marketplace or a game.

Most importantly, users are able to do this without the need to transfer the digital items between accounts, making it seamless to continue using the original app while also enjoying their assets in other contexts.

Security Considerations

Account linking is a very powerful Cadence feature, and thus it must be treated with care. So far in this document, we've discussed account linking between two accounts we own, even if the child account is managed by a third-party application. But, we can't make the same trust assumptions about custodial accounts in the real world.

Creating an &Account Capability and publishing it to an account we don't own means we are giving that account full access to our account. This should be seen as an anti-pattern.

:::warning

Creating an &Account Capability and sharing it with third-party account effectively the same as giving that person your account's private keys.

:::

Because unfiltered account linking can be dangerous, Flow introduces the HybridCustody contract that helps custodial applications regulate access while enabling parent accounts to manage their many child accounts and assets within them.

Hybrid Custody and Account Linking

Apps need assurances that their own resources are safe from malicious actors, so giving out full access might not be the form they want. Using hybrid custody contracts, the app still maintains control of their managed accounts, but they can:

1. Share capabilities freely, with a few built-in controls over the types of capabilities that can be retrieved by parent accounts via helper contracts (the CapabilityFactory, and CapabilityFilter)
2. Share additional capabilities (public or private) with a parent account via a CapabilityDelegator resource

Learn more about it in the Hybrid Custody documentation.

Guides

- Building Walletless Applications Using Child Accounts covers how apps can leverage Account Linking to create a seamless user experience and enable future self-custody.
- Working With Parent Accounts covers features enabled by the core HybridCustody contract to access child account assets from parent accounts. This is useful for apps like marketplaces or wallets that are working with accounts that have potential child accounts.

Resources

- Forum Post where core concepts were introduced and discussed.
- GitHub repository where HybridCustody core contracts and scripts are maintained. Check out the repository for more advanced script or transaction examples.
- Example Account Linking project with Magic.
- Starter template for Niftory Account Linking API.

```
# parent-accounts.md:
```

```
---
```

```
title: Working With Parent Accounts
```

```
sidebarposition: 2
```

```
--
```

In this doc, we'll continue from the perspective of a wallet or marketplace app seeking to facilitate a unified account experience, abstracting away the partitioned access between accounts into a single dashboard for user interactions on all their owned assets.

Objectives

- Understand the Hybrid Custody account model
- Differentiate between restricted child accounts and unrestricted owned accounts
- Get your app to recognize "parent" accounts along with any associated "child" accounts
- View Fungible and NonFungible Token metadata relating to assets across all of a user's associated accounts - their wallet-mediated "parent" account and any hybrid custody model "child" accounts
- Facilitate transactions acting on assets in child accounts

Design Overview

```
:::info
```

TL;DR: An account's HybridCustody.Manager is the entry point for all of a user's associated accounts.

:::

The basic idea in the Hybrid Custody model is relatively simple. A parent account is one that has received delegated (albeit restricted) access on another account. The account which has delegated authority over itself to the parent account is the child account.

In the Hybrid Custody Model, this child account would have shared access between the app - the entity which created and likely custodies the account - and the linked parent account.

How does this delegation occur? Typically when we think of shared account access in crypto, we think keys. However, Cadence enables accounts to link Capabilities on themselves and issue those Capabilities to other parties (more on capability-based access here).

This feature has been leveraged in an ecosystem standard so that apps can implement a hybrid custody model whereby the app creates an account it controls, then later delegates access on that account to the user once they've authenticated with their wallet.

All related constructs are used together in the HybridCustody contract to define the standard.

Parent accounts own a Manager resource which stores Capabilities to ChildAccount (restricted access) and OwnedAccount (unrestricted access) resources, both of which are stored in any given child account.

Therefore, the presence of a Manager in an account implies there are potentially associated accounts for which the owning account has delegated access. This resource is intended to be configured with a public Capability that enables querying of an account's child account addresses via getAccountAddresses() and getOwnedAccountAddresses(). As you can deduce from these two methods, there is a notion of "owned" accounts which we'll expand on in a bit.

A wallet or marketplace wishing to discover all of a user's accounts and assets within them can do so by first looking to the user's Manager.

Identifying Account Hierarchy

To clarify, insofar as the standard is concerned, an account is a parent account if it contains a Manager resource,

and an account is a child account if it contains at minimum an OwnedAccount or additionally a ChildAccount resource.

Within a user's Manager, its mapping of childAccounts points to the addresses of its child accounts in each key, with corresponding values giving the Manager access to those accounts via corresponding ChildAccount Capability.

!HybridCustody Conceptual Overview

Likewise, the child account's ChildAccount.parentAddress (which owns a Manager) points to the user's account as its parent address. This makes it easy to both identify whether an account is a parent, child, or both, and its associated parent/child account(s).

OwnedAccount resources underly all account delegations, so can have multiple parents whereas ChildAccounts are 1:1. This provides more granular revocation as each parent account has its own Capability path on which its access relies.

Restricted vs. Owned Accounts

It's worth noting here that ChildAccount Capabilities enable access to the underlying account according to rules configured by the child account delegating access. The ChildAccount maintains these rules along with an OwnedAccount Capability within which the &Account Capability is stored. Anyone with access to the surface level ChildAccount can then access the underlying Account, but only according the pre-defined rule set. These rules are fundamentally a list of Types that can/cannot be retrieved from an account.

The app developer can codify these rule sets on allowable Capability types in a CapabilityFilter along with a CapabilityFactory defining retrieval patterns for those Capabilities. When delegation occurs, the developer would provide the CapabilityFilter and CapabilityFactory Capabilities to an OwnedAccount resource which stores them in a ChildAccount resource. Then, capabilities are created for the OwnedAccount and ChildAccount resource and are given to the specified parent account.

So, if an app developer wants to enable Hybrid Custody but doesn't want to allow parent accounts to access FungibleToken Vaults, for example, the app developer can codify rule sets enumerating allowable Capability types in a CapabilityFilter along with a CapabilityFactory defining retrieval patterns for those Capabilities.

When delegation occurs, they would provide the CapabilityFilter and CapabilityFactory Capabilities to an

`OwnedAccount`. This `OwnedAccount` then wraps the given filter & factory Capabilities in a `ChildAccount` along with a Capability to itself before publishing the new `ChildAccount` Capability for the specified parent account to claim.

`:::info`

Note that by enumerating allowable Types in your `CapabilityFilter.Filter` implementation, you're by default excluding access to anything other than the Types you declare as allowable.

`:::`

As mentioned earlier, Managers also maintain access to "owned" accounts - accounts which define unrestricted access as they allow direct retrieval of encapsulated `&Account` Capabilities. These owned accounts, found in `Manager.ownedAccounts`, are simply `OwnedAccount` Capabilities instead of `ChildAccount` Capabilities.

`!HybridCustody Total Overview`

Considerations

Do note that this construction does not prevent an account from having multiple parent accounts or a child account from being the parent to other accounts. While initial intuition might lead one to believe that account associations are a tree with the user at the root, the graph of associated accounts among child accounts may lead to cycles of association.

We believe it would be unlikely for a use case to demand a user delegates authority over their main account (in fact we'd discourage such constructions), but delegating access between child accounts could be useful. As an example, consider a set of local game clients across mobile and web platforms, each with self-custodied app accounts having delegated authority to each other while both are child accounts of the user's main account.

Ultimately, it will be up to the implementing wallet/marketplace how far down the graph of account associations they'd want to traverse and display to the user.

Implementation

From the perspective of a wallet or marketplace app, some relevant things to know about the user are:

- Does this account have associated linked (child) accounts?
- What are those associated linked accounts, if any?
- What NFTs are owned by this user across all associated accounts?

- What are the balances of all FungibleTokens across all associated accounts?

And with respect to acting on the assets of child accounts and managing child accounts themselves:

- Accessing an NFT from a linked account's Collection
- Removing a linked account

Examples

Query Whether an Address Has Associated Accounts

This script will return true if a HybridCustody.Manager is stored and false otherwise

```
cadence haschildaccounts.cdc
import "HybridCustody"

access(all) fun main(parent: Address): Bool {
    let acct = getAuthAccount<auth(BorrowValue) &Account>(parent)
    if let manager = acct.storage.borrow<&HybridCustody.Manager>(from:
    HybridCustody.ManagerStoragePath) {
        return manager.getChildAddresses().length > 0
    }
    return false
}
```

Query All Accounts Associated with Address

The following script will return an array of addresses associated with a given account's address, inclusive of the provided address. If a HybridCustody.Manager is not found, the script will revert.

```
cadence getchildaddresses.cdc
import "HybridCustody"

access(all) fun main(parent: Address): [Address] {
    let acct = getAuthAccount<auth(Storage) &Account>(parent)
    let manager = acct.storage.borrow<&HybridCustody.Manager>(from:
    HybridCustody.ManagerStoragePath)
    ?? panic("manager not found")
    return manager.getChildAddresses()
}
```

Query All Owned NFT Metadata

While it is possible to iterate over the storage of all associated accounts in a single script, memory limits prevent this approach from scaling well.

Since some accounts hold thousands of NFTs, we recommend breaking up iteration, utilizing several queries to iterate over accounts and the storage of each account. Batching queries on individual accounts may even be required based on the number of NFTs held.

1. Get all associated account addresses (see above)
2. Looping over each associated account address client-side, get each address's owned NFT metadata

For simplicity, we'll show a condensed query, returning NFT display views from all accounts associated with a given address for a specified NFT Collection path.

```
cadence getnftdisplayviewfrompublic.cdc
import "NonFungibleToken"
import "MetadataViews"
import "HybridCustody"

/// Returns resolved Display from given address at specified path for
each ID or nil if ResolverCollection is not found
///
access(all)
fun getViews( address: Address, resolverCollectionPath: PublicPath): UInt64: MetadataViews.Display} {

    let account: PublicAccount = getAccount(address)
    let views: {UInt64: MetadataViews.Display} = {}

    // Borrow the Collection
    if let collection =
account.capabilities.borrow<&{NonFungibleToken.Collection}>(resolverCollectionPath) {
        // Iterate over IDs & resolve the view
        for id in collection.getIDs() {
            if let nft = collection.borrowNFT(id) {
                if let display =
nft.resolveView(Type<MetadataViews.Display>()) as? MetadataViews.Display
{
                    views.insert(key: id, display)
                }
            }
        }
    }

    return views
}

/// Queries for MetadataViews.Display each NFT across all associated
accounts from Collections at the provided
/// PublicPath
///
access(all)
```

```

fun main(address: Address, resolverCollectionPath: PublicPath): {Address: UInt64: MetadataViews.Display} {
    let allViews: {Address: {UInt64: MetadataViews.Display}} = {
        address: getViews(address, resolverCollectionPath)
    }

    / Iterate over any associated accounts /
    //
    let seen: [Address] = [address]
    if let managerRef = getAuthAccount<auth(BorrowValue)>&Account>(address)
        .storage
        .borrow<&HybridCustody.Manager>(from:
            HybridCustody.ManagerStoragePath) {
        for childAccount in managerRef.getChildAddresses() {
            allViews.insert(key: childAccount, getViews(address,
                resolverCollectionPath))
            seen.append(childAccount)
        }
    }

    for ownedAccount in managerRef.getOwnedAddresses() {
        if seen.contains(ownedAccount) == false {
            allViews.insert(key: ownedAccount, getViews(address,
                resolverCollectionPath))
            seen.append(ownedAccount)
        }
    }
}

return allViews
}

```

At the end of this query, the caller will have a mapping of Display views indexed on the NFT ID and grouped by account Address. Note that this script does not take batching into consideration and assumes that each NFT resolves the MetadataViews.Display view type.

Query All Account FungibleToken Balances

Similar to the previous example, we recommend breaking up this task due to memory limits.

1. Get all linked account addresses (see above)
2. Looping over each associated account address client-side, get each address's owned FungibleToken Vault metadata

However, we'll condense both of these steps down into one script for simplicity:

```
cadence getallvaultbalfromstorage.cdc
```

```

import "FungibleToken"
import "MetadataViews"
import "HybridCustody"

/// Returns a mapping of balances indexed on the Type of resource
containing the balance
///
access(all)
fun getAllBalancesInStorage( address: Address): {Type: UFix64} {
    // Get the account
    let account = getAuthAccount<auth(BorrowValue) &Account>(address)
    // Init for return value
    let balances: {Type: UFix64} = {}
    // Track seen Types in array
    let seen: [Type] = []
    // Assign the type we'll need
    let balanceType: Type = Type<@{FungibleToken.Balance}>()
    // Iterate over all stored items & get the path if the type is what
    we're looking for
    account.forEachStored(fun (path: StoragePath, type: Type): Bool {
        if (type.isInstance(balanceType) || type.isSubtype(of:
balanceType)) && !type.isRecovered {
            // Get a reference to the resource & its balance
            let vaultRef = account.borrow<&{FungibleToken.Balance}>(from:
path)!
            // Insert a new values if it's the first time we've seen the
            type
            if !seen.contains(type) {
                balances.insert(key: type, vaultRef.balance)
            } else {
                // Otherwise just update the balance of the vault
                // (unlikely we'll see the same type twice in
                // the same account, but we want to cover the case)
                balances[type] = balances[type]! + vaultRef.balance
            }
        }
        return true
    })
    return balances
}

/// Queries for FT.Vault balance of all FT.Vaults in the specified
account and all of its associated accounts
///
access(all)
fun main(address: Address): {Address: {Type: UFix64}} {

    // Get the balance for the given address
    let balances: {Address: {Type: UFix64}} = { address:
getAllBalancesInStorage(address) }
    // Tracking Addresses we've come across to prevent overwriting
    balances (more efficient than checking dict entries (?) )
    let seen: [Address] = [address]
}

```

```

    / Iterate over any associated accounts /
    //
    if let managerRef = getAuthAccount<auth(BorrowValue)
&Account>(address)
        .storage
        .borrow<&HybridCustody.Manager>(from:
HybridCustody.ManagerStoragePath) {

        for childAccount in managerRef.getChildAddresses() {
            balances.insert(key: childAccount,
getAllBalancesInStorage(address))
            seen.append(childAccount)
        }

        for ownedAccount in managerRef.getOwnedAddresses() {
            if seen.contains(ownedAccount) == false {
                balances.insert(key: ownedAccount,
getAllBalancesInStorage(address))
                seen.append(ownedAccount)
            }
        }
    }

    return balances
}

```

The above script returns a dictionary of balances indexed on the type and further grouped by account Address.

The returned data at the end of address iteration should be sufficient to achieve a unified balance of all Vaults of similar types across all of a user's associated account as well as a more granular per account view.

You might consider resolving FungibleTokenMetadataViews to aggregate more information about the underlying Vaults.

Access NFT in Child Account from Parent Account

A user with NFTs in their child accounts will likely want to utilize said NFTs. In this example, the user will sign a transaction with their authenticated account that retrieves a reference to a child account's NonFungibleToken.Provider, enabling withdrawal from the child account having signed as the parent account.

```

cadence withdrawnftfromchild.cdc
import "NonFungibleToken"
import "FlowToken"
import "HybridCustody"

transaction(

```

```

    childAddress: Address,           // Address of the child account
    storagePath: StoragePath,       // Path to the Collection in the child
account
    collectionType: Type,          // Type of the requested Collection from
which to withdraw
    withdrawID: UInt64             // ID of the NFT to withdraw
) {

let providerRef: &{NonFungibleToken.Provider}

prepare(signer: auth(BorrowValue) &Account) {
    // Get a reference to the signer's HybridCustody.Manager from
storage
    let managerRef = signer.storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(
        from: HybridCustody.ManagerStoragePath
    ) ?? panic("Could not borrow reference to
HybridCustody.Manager in signer's account at expected path!")

    // Borrow a reference to the signer's specified child account
    let account = managerRef
        .borrowAccount(addr: childAddress)
    ?? panic("Signer does not have access to specified child
account")

    // Get the Capability Controller ID for the requested collection
type
    let controllerID = account.getControllerIDForType(
        type: collectionType,
        forPath: storagePath
    ) ?? panic("Could not find Capability controller ID for
collection type ".concat(type.identifier)
        .concat(" at path ").concat(storagePath.toString()))

    // Get a reference to the child NFT Provider and assign to the
transaction scope variable
    let cap = account.getCapability(
        controllerID: controllerID,
        type: Type<auth(NonFungibleToken.Withdraw)
&{NonFungibleToken.Provider}>()
    ) ?? panic("Cannot access NonFungibleToken.Provider from this
child account")

    // We'll need to cast the Capability - this is possible thanks to
CapabilityFactory, though we'll rely on the relevant
    // Factory having been configured for this Type or it won't be
castable
    self.providerRef = cap as!
Capability<auth(NonFungibleToken.Withdraw) &{NonFungibleToken.Provider}>
}

execute {
    // Withdraw the NFT from the Collection
    let nft <- self.providerRef.withdraw(withdrawID: withdrawID)
}

```

```

    // Do stuff with the NFT
    // NOTE: Without storing or burning the NFT before scope closure,
this transaction will fail. You'll want to
        // fill in the rest of the transaction with the necessary
logic to handle the NFT
        // ...
    }
}

```

At the end of this transaction, you withdrew an NFT from the specified account using an NFT Provider Capability. A similar approach could get you any allowable Capabilities from a signer's child account.

Revoking Secondary Access on a Linked Account

The expected uses of child accounts for progressive onboarding implies that they will be accounts with shared access. A user may decide that they no longer want secondary parties to have access to the child account.

There are two ways a party can have delegated access to an account - keys and &Account Capability. With ChildAccount mediated access, a user wouldn't be able to revoke anyone's access except for their own. With unrestricted access via OwnedAccount, one could remove parents (OwnedAccount.removeParent(parent: Address)) thereby unlinking relevant Capabilities and further destroying their ChildAccount and CapabilityDelegator resources.

For now, we recommend that if users want to revoke secondary access, they transfer any assets from the relevant child account and remove it from their Manager altogether.

Remove a Child Account

As mentioned above, if a user no longer wishes to share access with another party, it's recommended that desired assets be transferred from that account to either their main account or other linked accounts and the linked account be removed from their HybridCustody.Manager. Let's see how to complete that removal.

```

cadence removechildaccount.cdc
import "HybridCustody"

transaction(child: Address) {
    prepare (acct: auth(BorrowValue) &Account) {
        let manager = acct.storage.borrow<auth(HybridCustody.Manage)
&HybridCustody.Manager>(
            from: HybridCustody.ManagerStoragePath
        ) ?? panic("manager not found")
        manager.removeChild(addr: child)
    }
}

```

```
}
```

After removal, the signer no longer has delegated access to the removed account via their Manager and the caller is removed as a parent of the removed child.

Note also that it's possible for a child account to remove a parent. This is necessary to give application developers and ultimately the owners of these child accounts the ability to revoke secondary access on owned accounts.

```
# ios-quickstart.md:
```

```
---
```

```
title: IOS Development
```

```
sidebarlabel: IOS Development
```

```
sidebarposition: 3
```

```
---
```

Overview

The following documentation aims to educate you on building a native mobile application on Flow. It first presents Monster Maker, a starter project we've built to represent simple Flow mobile concepts. Next it presents various developer resources related to building mobile native Flow applications.

Monster Maker

```
!monstermakerlogo.png
```

Monster Maker is a native iOS app that allows users to connect a wallet, sign a transaction to mint an NFT (a monster) and display their collection of NFTs (their monsters) within the app. It's meant to be a lightweight sample project to exemplify how to build a mobile native Flow project. If you're looking to build a native mobile application for Flow, exploring the Monster Maker code base first or even building off of it is a great way to get started.

```
<aside>
```

 Note - We currently only have an iOS project for Monster Maker. That said an Android and web version of the same project is in active development.

```
</aside>
```

Github Repo

The Monster Maker Github Repo can be found here:

<https://github.com/onflow/monster-maker>

Building to Device

Before you run Monster Maker on your device, please make sure you have installed the Xcode14 from Mac App Store. Once you clone the repo, open the MonsterMaker.xcodeproj under the iOS folder.

Xcode should automatically setup the project for you. If you do see any error related to dependencies, run Xcode Menu -> File -> Packages -> Reset Package Cache to resolve the issue.

In the meantime, you can choose a simulator or your iPhone to run. For more detail here is the official doc.

For run in real device, there are a few steps to deal with signing:

1. Add your apple account to the Xcode which can be accessed from Xcode Menu -> Settings -> Add account.
2. Change the Team to your Personal Apple account from the Signing & Capabilities under the project target menu. For more detail, please check the screenshot below.

!XCode Target Setup

Connecting to a Wallet

To connect with wallets, there is native wallet discovery in the app. Once you click on connect, it will bring out the list of the wallets which support HTTP/POST or WC/RPC method.

FCL Config

To make sure, the wallet can recognise your dApp, there is a few field you will need to config before connect to a wallet. The account proof config is optional. In addition, you will need to create a project id from walletconnect cloud before you can connect to the WC/RPC compatible wallet such as dapper self custody or lilico wallet.

```
swift
import FCL

// Config the App
let defaultProvider: FCL.Provider = .dapperPro
let defaultNetwork: Flow.ChainID = .testnet // or .mainnet

// Optinal: Config for account proof
let accountProof = FCL.Metadata.AccountProofConfig(appIdentifier:
"Monster Maker")

// Config for WC/RPC compatible wallet
let walletConnect = FCL.Metadata.WalletConnectConfig(urlScheme: "monster-
maker://", projectID: "12ed93a2aae83134c4c8473ca97d9399")

// Config basic dApp info
let metadata = FCL.Metadata(appName: "Monster Maker",
```

```

        appDescription: "Monster Maker Demo App for
mobile",
        appIcon: URL(string:
"https://i.imgur.com/jscDmDe.png")!,
        location: URL(string: "https://monster-
maker.vercel.app/")!,
        accountProof: accountProof,
        walletConnectConfig: walletConnect)
fcl.config(metadata: metadata,
    env: defaultNetwork,
    provider: defaultProvider)

// Import keywords replacement for cadence query and transaction
fcl.config
    .put("0xFungibleToken", value: "0x631e88ae7f1d7c20")
    .put("0xMonsterMaker", value: "0xfd3d8fe2c8056370")
    .put("0xMetadataViews", value: "0x631e88ae7f1d7c20")
    .put("0xTransactionGeneration", value: "0x44051d81c4720882")

```

Open wallet discovery

!In Monster Maker, the Connect button triggers opening of Wallet Discovery

In Monster Maker, the Connect button triggers opening of Wallet Discovery

For the wallet support HTTP/POST, it will use webview to show the following actions.

For the wallet support WC/RPC, it will use deep-link to the wallet for actions.

You can open the native wallet discovery to make the selection, but also you can connect to the specific wallet as well.

Here is the code snippet of it:

```

swift
import FCL

// Open discovery view
fcl.openDiscovery()

// Or manual connect to specific wallet
try fcl.changeProvider(provider: provider, env: .testnet)
try await fcl.authenticate()

```

Signing a Transaction

!In Monster Maker, Initializing the NFT collection with the Initialize button triggers a transaction.

In Monster Maker, Initializing the NFT collection with the Initialize button triggers a transaction.

Similar to what we have on fcl-js, native sdk also use query and mutate for on-chain interactions. To request a signature from user, you can simply use fcl.mutate method. By default, the user will be the payer, proposer and authorizer, if you want to add custom authorizer please refer to the code from Server and iOS end.

```
swift
guard let user = fcl.currentUser else {
    // Not signin
    return
}

let txId = try await fcl.mutate(
    cadence: """
        transaction(test: String, testInt: Int) {
            prepare(signer: &Account) {
                log(signer.address)
                log(test)
                log(testInt)
            }
        }
    """
    args: [
        .string("Hello"),
        .int(10)
    ],
    gasLimit: 999,
    authorizors: [user]
)

print("txId -> \(txId)")
```

View NFT

The View page in Monster Maker exemplifies showing Monster Maker NFTs held by the connected wallet

The View page in Monster Maker exemplifies showing Monster Maker NFTs held by the connected wallet

During development, you always can query your NFT with fcl.query. Here is an example:

- Query cadence

```
cadence
import NonFungibleToken from 0xNonFungibleToken
```

```

import MonsterMaker from 0xMonsterMaker
import MetadataViews from 0xMetadataViews

access(all) struct Monster {
    access(all) let name: String
    access(all) let description: String
    access(all) let thumbnail: String
    access(all) let itemID: UInt64
    access(all) let resourceID: UInt64
    access(all) let owner: Address
    access(all) let component: MonsterMaker.MonsterComponent
}

init(
    name: String,
    description: String,
    thumbnail: String,
    itemID: UInt64,
    resourceID: UInt64,
    owner: Address,
    component: MonsterMaker.MonsterComponent
) {
    self.name = name
    self.description = description
    self.thumbnail = thumbnail
    self.itemID = itemID
    self.resourceID = resourceID
    self.owner = owner
    self.component = component
}
}

access(all) fun getMonsterById(address: Address, itemID: UInt64): Monster? {
    if let collection =
getAccount(address).capabilities.get<&MonsterMaker.Collection>(MonsterMaker.CollectionPublicPath).borrow() {

        if let item = collection.borrowMonsterMaker(id: itemID) {
            if let view =
item.resolveView(Type<MetadataViews.Display>()) {
                let display = view as! MetadataViews.Display
                let owner: Address = item.owner!.address!
                let thumbnail = display.thumbnail as!
MetadataViews.HTTPFile

                return Monster(
                    name: display.name,
                    description: display.description,
                    thumbnail: thumbnail.url,
                    itemID: itemID,
                    resourceID: item.uuid,
                    owner: address,
                    component: item.component
                )
            }
        }
    }
}

```

```

        )
    }
}

return nil
}

access(all) fun main(address: Address): [Monster] {
    let account = getAccount(address)
    let collectionRef =
account.capabilities.get<&{NonFungibleToken.Collection}>(MonsterMaker.CollectionPublicPath).borrow()
    ?? panic("The account with address "
            .concat(address.toString())
            .concat(" does not have a NonFungibleToken Collection
at ")
            .concat(MonsterMaker.CollectionPublicPath.toString())
            .concat("."). Make sure the account address is correct
and is initialized their account with a MonsterMaker Collection!"))

    let ids = collectionRef.getIDs()

    let monsters : [Monster] = []

    for id in ids {
        if let monster = getMonsterById(address: address, itemID:
id) {
            monsters.append(monster)
        }
    }

    return monsters
}

```

```

swift
let nftList = try await fcl.query(script: cadenceScript,
                                    args: [.address(address)])
                                    .decode([NFTModel].self)

```

External Resources

FCL Swift

FCL Swift is the iOS native SDK for FCL. This SDK is integrated into the Monster Maker sample.

<https://github.com/Outblock/fcl-swift>

FCL Android

FCL Android is the Android native SDK for FCL.

<https://github.com/Outblock/fcl-android>

FCL Wallet Connect 2.0

One of the easiest ways to connect to a wallet via a mobile native dApp is through Flow's new support for Wallet Connect 2.0. This is the pattern that Monster Maker uses to connect to the Dapper Self Custody wallet and Lilico. For more information on FCL Wallet Connect 2.0, check out this page:

FCL Wallet Connect

How to Build a Native iOS Dapp

The Agile Monkeys has written a very comprehensive guide on how to build a native mobile application on iOS and interface with fcl-swift. Found here:

How to Build a Native iOS Dapper Source Code

```
# overview.md:
```

```
---
```

```
title: Overview
sidebarlabel: Overview
sidebarposition: 1
---
```

Building mobile native applications that interact with the blockchain enables a much richer end user experiences and provides access to OS capabilities. With Flow Mobile, developers can build native applications for iOS and Android leveraging SDKs and mobile wallets.

Why Flow

Millions of users with Flow accounts are exploring the ecosystem and looking for applications. Most of these users purchased Flow NFTs and are comfortable with web3 principles.

In addition to the existing userbase, developers can tap into smart contracts deployed on the Flow blockchain. These contracts, including their on-chain state, provide unique possibilities to build experiences that enrich applications users are already using.

The following key capabilities make Flow a standout choice for mobile applications:

- On-device key encryption via Secure Enclave & Keychain

- Mobile wallet compatibility and support for WalletConnect 2.0
- Simple, progressive onboarding experience with postponed account linking
- Seamless in-app experience with on-chain interactions without constant signing requests
- Account flexibility enabling secure account recovery and sharing

Why Flow Mobile

Proven

Flow is built with mainstream adoption in mind. Mobile applications can leverage the best-in-class user experiences millions of users have enjoyed on the web, through applications like NBA TopShot or NFL AllDay.

Best-in-class UX

Flow's Client Library makes it very intuitive to sign up and sign in with their wallet of choice. For transaction signing, Flow offers human readable security, so users get a clear understanding of what they are approving. An increased sense of trust for Flow applications is the outcome.

Furthermore, Flow's powerful account model allows for seamless user flows of on-chain operations. Apps can perform transactions on behalf of the users (with their approval) in the background, without the need to switch between apps. The account model also allows apps to pay for transactions to postpone fiat on-ramps to get them to experience the value of an application before committing to buying tokens.

Last but not least, developers can leverage progressive web3 onboarding, in which any identity provider can be used to authenticate users, without having to deal with keys. Developers can create Flow accounts for the users and link them to a wallet at a later point in time.

Security first

Flow's mobile SDKs use on-device key encryption via Apple's Secure Enclave and Android's Keystore. The flexible account model makes it possible for an account to have multiple keys with different weights, which enables secure social recovery, account sharing, and much more.

Smart contract language inspired by mobile languages

Cadence, Flow's smart contract language, will look and feel very familiar to mobile languages developers are already familiar with. Cadence was inspired by Move, Swift, and Kotlin. This reduces the ramp-up period to develop mobile applications leveraging on-chain logic.

What is available

Developers can leverage the following features to get productive quickly:

- Swift & Kotlin FCL SDKs to auth and interact with the Flow blockchain (query + execute scripts)
- FCL-compatible mobile wallets
- User auth using WalletConnect 2.0
- Basic mobile sample application (MonsterMaker)

Coming soon:

- Samples for key in-app functionality, like in-app purchasing
- Progressive user onboarding

```
# react-native-quickstart.md:
```

```
---
title: React Native Development
sidebarlabel: React Native Development
sidebarposition: 4
---
```

Last Updated: January 11th 2022

> Note: This page will walk you through a very bare bones project to get started building a web3 dapp using the Flow Client Library (FCL). If you are looking for a clonable repo, Flow community members have created quickstart templates for different JavaScript frameworks (e.g. Next.js, SvelteKit, Nuxt). You can consult the complete list [here](#).

Introduction

FCL-JS is the easiest way to start building decentralized applications. FCL (aka Flow Client Library) wraps much of the logic you'd have to write yourself on other blockchains. Follow this quick start and you'll have a solid overview of how to build a shippable dapp on Flow.

We're going to make an assumption that you know or understand React; however, the concepts should be easy to understand and transfer to another framework. While this tutorial will make use of Cadence (Flow's smart contract language), you do not need to know it. Instead, we recommend later diving into learning the Cadence language once you've gotten the core FCL concepts down.

In this tutorial, we are going to interact with an existing smart contract on Flow's testnet known as the Profile Contract. Using this contract, we will create a new profile and edit the profile information, both via a wallet. In order to do this, the FCL concepts we'll cover are:

- Installation
- Configuration
- Authentication
- Querying the Blockchain
- Initializing an Account
- Mutating the Blockchain

And if you ever have any questions we're always happy to help on Discord. There are also links at the end of this article for diving deeper into building on Flow.

Installation

The first step is to generate a React app using Next.js and create-expo-app. From your terminal, run the following:

```
sh  
npx create-expo-app flow-react-native  
cd flow-react-native
```

Next, install FCL so we can use it in our app.

```
sh  
npm install @onflow/fcl@alpha @react-native-async-storage/async-storage  
expo-web-browser --save
```

Now run the app using the following command in your terminal.

```
sh  
npm run start
```

You should now see your app running.

Configuration

Now that your app is running, you can configure FCL. Within the main project directory, create a folder called flow and create a file called config.js. This file will contain configuration information for FCL, such as what Access Node and wallet discovery endpoint to use (e.g. testnet or a local emulator). Add the following contents to the file:

Note: These values are required to use FCL with your app.

```
> Create file: ./flow/config.js  
  
javascript ./flow/config.js  
import { config } from "@onflow/fcl";  
  
config({  
  "accessNode.api": "https://rest-testnet.onflow.org", // Mainnet:  
  "https://rest-mainnet.onflow.org"  
  "discovery.wallet": "https://fcl-discovery.onflow.org/testnet/authn",  
  // Mainnet: "https://fcl-discovery.onflow.org/authn"  
  "discovery.authn.endpoint": "https://fcl-  
  discovery.onflow.org/api/testnet/authn", // Mainnet: "https://fcl-  
  discovery.onflow.org/api/authn"  
})
```

Tip: It's recommended to replace these values with environment variables for easy deployments across different environments like development/production or Testnet/Mainnet.

- The `accessNode.api` key specifies the address of a Flow access node. Flow provides these, but in the future access to Flow may be provided by other 3rd parties, through their own access nodes.
- `discovery.wallet` and `discovery.authn.endpoint` are addresses that point to a service that lists FCL compatible wallets. Flow's FCL Discovery service is a service that FCL wallet providers can be added to, and be made 'discoverable' to any application that uses the `discovery.wallet` and `discovery.authn.endpoint`.

> Learn more about configuring Discovery or setting configuration values.

> If you are running a Wallet Discovery locally and want to use it in the React Native app, change `https://fcl-discovery.onflow.org/` to `http://<LOCALIPADDRESS>:<PORT>/`

> For Example:

> using local Wallet Discovery and local Dev Wallet:

>

> `javascript ./flow/config.js`

> `import { config } from "@onflow/fcl";`

>

> `config({`

> `...`

> `"discovery.wallet": "http://10.0.0.1:3002/local/authn",`

> `"discovery.authn.endpoint": "http://10.0.0.1:3002/api/local/authn",`

> `...`

> })

>

The main screen for React Native apps is located in `./App.js` or in `./App.tsx`. So let's finish configuring our dapp by going in the root directory and importing the config file into the top of our `App.js` file. We'll then swap out the default component in `App.js` to look like this:

> Replace file: `./App.js`

```
jsx ./App.js
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View } from 'react-native';
import './flow/config';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
```

```
    container: {
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center',
    },
  );
});
```

Now we're ready to start talking to Flow!

Authentication

To authenticate a user, you'll need to render a `ServiceDiscovery` component provided by `fcl-react-native`. Alternatively you can build your own component using `useServiceDiscovery`.

`Unauthenticate` is as simple as calling `fcl.unauthenticate()`. Once authenticated, FCL sets an object called `fcl.currentUser` which exposes methods for watching changes in user data, signing transactions, and more. For more information on the `currentUser`, read more [here](#).

Let's add in a few components and buttons buttons for sign up/login and also subscribe to changes on the `currentUser`. When the user is updated (which it will be after authentication), we'll set the user state in our component to reflect this. To demonstrate user authenticated sessions, we'll conditionally render a component based on if the user is or is not logged in.

This is what your file should look like now:

```
> Replace file: ./App.js

jsx ./App.js
import { Text, View, Button } from 'react-native';
import "./flow/config";

import { useState, useEffect } from "react";
import as fcl from "@onflow/fcl/dist/fcl-react-native";

export default function App() {

  const [user, setUser] = useState({loggedIn: null})

  useEffect(() => fcl.currentUser.subscribe(setUser), [])

  const AuthedState = () => {
    return (
      <View>
        <Text>Address: {user?.addr ?? "No Address"}</Text>
        <Button onPress={fcl.unauthenticate} title='Log Out' />
      </View>
    )
  }
}
```

```

        )
    }

    if (user.loggedIn) {
        return <View style={styles.container}>
            <Text>Flow App</Text>
            <AuthedState />
            <StatusBar style="auto" />
        </View>
    }

    return (
        <fcl.ServiceDiscovery fcl={fcl}/>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: '#fff',
        alignItems: 'center',
        justifyContent: 'center',
    },
});

```

You should now be able to log in or sign up a user and unauthenticate them. Upon logging in or signing up your users will see a popup where they can choose between wallet providers. Let's select the Blocto wallet for this example to create an account. Upon completing authentication, you'll see the component change and the user's wallet address appear on the screen if you've completed this properly.

Querying the Blockchain

One of the main things you'll often need to do when building a dapp is query the Flow blockchain and the smart contracts deployed on it for data. Since smart contracts will live on both Testnet and Mainnet, let's put the account address where the smart contract lives into the configuration (remember, it's recommended that you change this later to use environment variables). Let's also give it a key of Profile and prefix it with 0x so that the final key is 0xProfile. The prefix is important because it tells FCL to pull the corresponding addresses needed from the configuration value.

> Replace file: ./flow/config.js

```

javascript ./flow/config.js
import { config } from "@onflow/fcl";

config({
    "accessNode.api": "https://rest-testnet.onflow.org", // Mainnet:
    "https://rest-mainnet.onflow.org"
}

```

```

    "discovery.wallet": "https://fcl-discovery.onflow.org/testnet/authn",
    // Mainnet: "https://fcl-discovery.onflow.org/authn"
    "discovery.authn.endpoint": "https://fcl-
discovery.onflow.org/api/testnet/authn",
    "0xProfile": "0xba1132bc08f82fe2" // The account address where the
Profile smart contract lives on Testnet
})

```

If you want to see the on chain smart contract we'll be speaking with next, you can view the Profile Contract source code but again for this tutorial it's not necessary you understand it.

First, lets query the contract to see what the user's profile name is.

A few things need to happen in order to do that:

1. We need to import the contract and pass it the user's account address as an argument.
2. Execute the script using fcl.query.
3. Set the result of the script to the app state in React so we can display the profile name in our browser.
4. Display "No Profile" if one was not found.

Take a look at the new code. We'll explain each new piece as we go. Remember, the cadence code is a separate language from JavaScript used to write smart contracts, so you don't need to spend too much time trying to understand it. (Of course, you're more than welcome to, if you want to!)

> Replace file: ./App.js

```

jsx ./App.js
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View, Button } from 'react-native';
import { useEffect, useState } from 'react';
import './flow/config'

import as fcl from "@onflow/fcl/dist/fcl-react-native";

export default function App() {

  const [user, setUser] = useState({loggedIn: null})
  const [name, setName] = useState('') // NEW

  useEffect(() => fcl.currentUser.subscribe(setUser), [])

  // NEW
  const sendQuery = async () => {
    const profile = await fcl.query({
      cadence:
        import Profile from 0xProfile

        access(all) fun main(address: Address): Profile.ReadOnly? {
          return Profile.read(address)
        }
    })
}

```

```

        ,
        args: (arg, t) => [arg(user.addr, t.Address) ]
    })

    setName(profile?.name ?? 'No Profile')
}

const AuthedState = () => {
    return (
        <View >
            <Text>Address: {user?.addr ?? "No Address"}</Text>{/ NEW /}
            <Text>Profile Name: {name ?? "--"}</Text>{/ NEW /}
            <Button onPress={sendQuery} title='Send Query'#/ NEW />
            <Button onPress={fcl.unauthenticate} title='Log Out'#/ NEW />
        </View>
    )
}

if (user.loggedIn) {
    return <View style={styles.container}>
        <Text>Flow App</Text>
        <AuthedState />
        <StatusBar style="auto" />
    </View>
}

return (
    <fcl.ServiceDiscovery fcl={fcl}/>
)
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: '#fff',
        alignItems: 'center',
        justifyContent: 'center',
    },
});
```

A few things happened. In our AuthedState component, we added a button to send a query for the user's profile name and a Text to display the result above it. The corresponding useState initialization can be seen at the top of the component.

The other thing we did is build out the actual query inside of sendQuery method. Let's take a look at it more closely:

```
javascript
await fcl.query({
    cadence:
        import Profile from 0xProfile
```

```

access(all) fun main(address: Address): Profile.ReadOnly? {
    return Profile.read(address)
}

,
args: (arg, t) => [arg(user.addr, t.Address)]
);

```

Inside the query you'll see we set two things: cadence and args. Cadence is Flow's smart contract language we mentioned. For this tutorial, when you look at it you just need to notice that it's importing the Profile contract from the account we named `0xProfile` earlier in our config file, then also taking an account address, and reading it. That's it until you're ready to learn more Cadence.

In the args section, we are simply passing it our user's account address from the user we set in state after authentication and giving it a type of `Address`. For more possible types, see this reference.

Go ahead and click the "Send Query" button. You should see "No Profile." That's because we haven't initialized the account yet.

Initializing an Account

For the Profile contract to store a Profile in a user's account, it does so by initializing what is called a "resource." A resource is an ownable piece of data and functionality that can live in the user's account storage. This paradigm is known as "resource-oriented-programming", a principle that is core to Cadence and differentiates its ownership model from other smart contract languages, read more here. Cadence makes it so that resources can only exist in one place at any time, they must be deliberately created, cannot be copied, and if desired, must be deliberately destroyed.

> There's a lot more to resources in Cadence than we'll cover in this guide, so if you'd like to know more, check out this Cadence intro.

To do this resource initialization on an account, we're going to add another function called `initAccount`. Inside of that function, we're going to add some Cadence code which says, "Hey, does this account have a profile? If it doesn't, let's add one." We do that using something called a "transaction." Transactions occur when you want to change the state of the blockchain, in this case, some data in a resource, in a specific account. And there is a cost (transaction fee) in order to do that; unlike a query.

That's where we jump back into FCL code. Instead of query, we use `mutate` for transactions. And because there is a cost, we need to add a few fields that tell Flow who is proposing the transaction, who is authorizing it, who is paying for it, and how much they're willing to pay for it. Those fields – not surprisingly – are called: `proposer`, `authorizer`, `payer`, and `limit`. For more information on these signatory roles, check out this doc.

Let's take a look at what our account initialization function looks like:

```
javascript
const initAccount = async () => {
  const transactionId = await fcl.mutate({
    cadence:
      import Profile from '0xProfile'

    transaction {
      prepare(account: auth(Storage, Capabilities) & Account) {
        // Only initialize the account if it hasn't already been
        initialized
        if (!Profile.check(account.address)) {
          // This creates and stores the profile in the user's account
          account.storage.save(<- Profile.new(), to:
            Profile.privatePath)

          // This creates the public capability that lets applications
          read the profile's info
          let newCap =
            account.capabilities.storage.issue<&Profile.Base>(Profile.privatePath)

          account.capabilities.publish(newCap, at: Profile.publicPath)
        }
      }
    }
  }

  payer: fcl.authz,
  proposer: fcl.authz,
  authorizations: [fcl.authz],
  limit: 50
})

const transaction = await fcl.tx(transactionId).onceSealed()
console.log(transaction)
}
```

You can see the new fields we talked about. You'll also notice `fcl.authz`. That's shorthand for "use the current user to authorize this transaction", (you could also write it as `fcl.currentUser.authorization`). If you want to learn more about transactions and signing transactions, you can view the docs here. For this example, we'll keep it simple with the user being each of these roles.

You'll also notice we are awaiting a response with our transaction data by using the syntax `fcl.tx(transactionId).onceSealed()`. This will return when the blockchain has sealed the transaction and it's complete in processing it and verifying it.

Now your `index.js` file should look like this (we also added a button for calling the `initAccount` function in the `AuthedState`):

```

> Replace file: ./App.js

jsx ./App.js
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View, Button } from 'react-native';
import { useEffect, useState } from 'react';
import './flow/config'

import as fcl from "@onflow/fcl/dist/fcl-react-native";

export default function App() {

  const [user, setUser] = useState({loggedIn: null})
  const [name, setName] = useState('')

  useEffect(() => fcl.currentUser.subscribe(setUser), [])

  const sendQuery = async () => {
    const profile = await fcl.query({
      cadence:
        import Profile from 0xProfile

        access(all) fun main(address: Address): Profile.ReadOnly? {
          return Profile.read(address)
        }
      ,
      args: (arg, t) => [arg(user.addr, t.Address)]
    })
    setName(profile?.name ?? 'No Profile')
  }

  // NEW
  const initAccount = async () => {
    const transactionId = await fcl.mutate({
      cadence:
        import Profile from 0xProfile

        transaction {
          prepare(account: auth(Storage, Capabilities) &Account) {
            // Only initialize the account if it hasn't already been
initialized
            if (!Profile.check(account.address)) {
              // This creates and stores the profile in the user's
account
              account.storage.save(<- Profile.new(), to:
Profile.privatePath)

              // This creates the public capability that lets
applications read the profile's info
              let newCap =
account.capabilities.storage.issue<&Profile.Base>(Profile.privatePath)
            }
          }
        }
    })
  }
}

```

```

        account.capabilities.publish(newCap, at:
Profile.publicPath)
    }
}
}

,
payer: fcl.authz,
proposer: fcl.authz,
authorizations: [fcl.authz],
limit: 50
})

const transaction = await fcl.tx(transactionId).onceSealed()
console.log(transaction)
}

const AuthedState = () => {
return (
<View >
<Text>Address: {user?.addr ?? "No Address"}</Text>
<Text>Profile Name: {name ?? "--"}</Text>
<Button onPress={sendQuery} title='Send Query' />
<Button onPress={initAccount} title='Init Account' />{/ NEW /}
<Button onPress={fcl.unauthenticate} title='Log Out' />
</View>
)
}

if (user.loggedIn) {
return <View style={styles.container}>
<Text>Flow App</Text>
<AuthedState />
<StatusBar style="auto" />
</View>
}

return (
<fcl.ServiceDiscovery fcl={fcl}/>
)
}

const styles = StyleSheet.create({
container: {
flex: 1,
backgroundColor: '#fff',
alignItems: 'center',
justifyContent: 'center',
},
});

```

Press the "Init Account" button you should see the wallet ask you to approve a transaction. After approving, you will see a transaction

response appear in your console (make sure to have that open). It may take a few moments. With the transaction result printed, you can use the transactionId to look up the details of the transaction using a block explorer.

Mutating the Blockchain

Now that we have the profile initialized, we are going to want to mutate it some more. In this example, we'll use the same smart contract provided to change the profile name.

To do that, we are going to write another transaction that adds some Cadence code which lets us set the name. Everything else looks the same in the following code except for one thing: we'll subscribe to the status changes instead of waiting for it to be sealed after the mutate function returns.

It looks like this:

```
javascript
const executeTransaction = async () => {
    const transactionId = await fcl.mutate({
        cadence:
            import Profile from 0xProfile

            transaction(name: String) {
                prepare(account: auth(BorrowValue) &Account) {
                    let profileRef = account.borrow<&Profile.Base>(from:
                        Profile.privatePath)
                    ?? panic("The signer does not store a Profile.Base object
at the path "
                            .concat(Profile.privatePath.toString())
                            .concat(". The signer must initialize their
account with this object first!"))

                    profileRef.setName(name)
                }
            }
        ,
        args: (arg, t) => [arg("Flow Developer", t.String)],
        payer: fcl.authz,
        proposer: fcl.authz,
        authorizations: [fcl.authz],
        limit: 50
    })

    fcl.tx(transactionId).subscribe(res =>
        setTransactionStatus(res.status))
}
```

Here you can see our argument is "Flow Developer" and at the bottom we've called the subscribe method instead of onceSealed.

Let's see how that works inside our whole index.js file. But, let's also set the statuses to our React component's state so we can see on screen what state we're in.

```
> Replace file: ./App.js

jsx ./App.js
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View, Button } from 'react-native';
import { useEffect, useState } from 'react';
import './flow/config'

import as fcl from "@onflow/fcl/dist/fcl-react-native";

export default function App() {

  const [user, setUser] = useState({loggedIn: null})
  const [name, setName] = useState('')
  const [transactionStatus, setTransactionStatus] = useState(null) // NEW

  useEffect(() => fcl.currentUser.subscribe(setUser), [])

  const sendQuery = async () => {
    const profile = await fcl.query({
      cadence:
        import Profile from 0xProfile

        access(all) fun main(address: Address): Profile.ReadOnly? {
          return Profile.read(address)
        }

        args: (arg, t) => [arg(user.addr, t.Address)]
    })
    setName(profile?.name ?? 'No Profile')
  }

  const initAccount = async () => {
    const transactionId = await fcl.mutate({
      cadence:
        import Profile from 0xProfile

        transaction {
          prepare(account: auth(Storage, Capabilities) & Account) {
            // Only initialize the account if it hasn't already been
            initialized
            if (!Profile.check(account.address)) {
              // This creates and stores the profile in the user's
              account
              account.storage.save(<- Profile.new(), to:
                Profile.storagePath)

              // This creates the public capability that lets
              applications read the profile's info
            }
          }
        }
    })
  }
}
```

```

        let newCap =
account.capabilities.storage.issue<&Profile.Base>(Profile.privatePath)

        account.capabilities.publish(newCap, at:
Profile.publicPath)
    }
}
}

,
payer: fcl.authz,
proposer: fcl.authz,
authorizations: [fcl.authz],
limit: 50
})

const transaction = await fcl.tx(transactionId).onceSealed()
console.log(transaction)
}

// NEW
const executeTransaction = async () => {
const transactionId = await fcl.mutate({
cadence:
import Profile from 0xProfile

transaction(name: String) {
    prepare(account: auth(BorrowValue) &Account) {
        let profileRef = account.storage.borrow<&Profile.Base>(from:
Profile.privatePath)
        ?? panic("The signer does not store a Profile.Base object
at the path "
            .concat(Profile.privatePath.toString())
            .concat(". The signer must initialize their
account with this object first!"))

        profileRef.setName(name)
    }
}

,
args: (arg, t) => [arg("Flow Developer", t.String)],
payer: fcl.authz,
proposer: fcl.authz,
authorizations: [fcl.authz],
limit: 50
})

fcl.tx(transactionId).subscribe(res =>
setTransactionStatus(res.status))
}

const AuthedState = () => {
return (
<View>
<Text>Address: {user?.addr ?? "No Address"}</Text>

```

```

        <Text>Profile Name: {name ?? "--"}</Text>
        <Text>Transaction Status: {transactionStatus ?? "--"}</Text>{/ NEW /}
    NEW /}
        <Button onPress={sendQuery} title='Send Query' />
        <Button onPress={initAccount} title='Init Account' />{/ NEW /}
        <Button onPress={executeTransaction} title='Execute
Transaction' />{/ NEW /}
        <Button onPress={fcl.unauthenticate} title='Log Out' />
    </View>
)
}

if (user.loggedIn) {
    return <View style={styles.container}>
        <Text>Flow App</Text>
        <AuthedState />
        <StatusBar style="auto" />
    </View>
}

return (
    <fcl.ServiceDiscovery fcl={fcl}/>
)
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: '#fff',
        alignItems: 'center',
        justifyContent: 'center',
    },
});

```

Now if you click the "Execute Transaction" button you'll see the statuses update next to "Transaction Status." When you see "4" that means it's sealed! Status code meanings can be found [here](#). If you query the account profile again, "Profile Name:" should now display "Flow Developer".

That's it! You now have a shippable Flow dapp that can auth, query, init accounts, and mutate the chain. This is just the beginning. There is so much more to know. We have a lot more resources to help you build. To dive deeper, here are a few good places for taking the next steps:

Cadence

- Cadence Playground Tutorials
- Cadence Hello World Video
- Why Cadence?

Full Stack NFT Marketplace Example

- Beginner Example: CryptoDappy

More FCL

- FCL API Quick Reference
- More on Scripts
- More on Transactions
- User Signatures
- Proving Account Ownership

walletless-pwa.md:

```
---
```

title: Build a Walletless Mobile App (PWA)
sidebarlabel: Build a Walletless Mobile App (PWA)
sidebarposition: 2

Overview

In this tutorial, we delve into the intricacies of crafting an accessible Progressive Web App (PWA) on the Flow blockchain, tackling the challenge of mobile mainstream accessibility in web3. Recognizing the complexity of current onboarding processes, we will guide you through a streamlined approach, featuring a seamless walletless mobile login to alleviate the often daunting task for new users.

Understanding Progressive Web Apps (PWAs)

Progressive Web Apps (PWAs) have garnered attention recently, with platforms like friend.tech leading the way in popularity. PWAs blur the lines between web pages and mobile applications, offering an immersive, app-like experience directly from your browser. You can easily add a shortcut to your home screen, and the PWA operates just like a native application would. Beyond these capabilities, PWAs also boast offline functionality and support for push notifications, among many other features.

Exploring Walletless Onboarding

Walletless onboarding is a groundbreaking feature that enables users to securely interact with decentralized applications (dApps) in a matter of seconds, all without the traditional necessity of creating a blockchain wallet. This method effectively simplifies the user experience, abstracting the complexities of blockchain technology to facilitate swift and straightforward app access. For a deeper dive into walletless onboarding and its integration with Flow, feel free to explore the following resource: [Flow Magic Integration](#).

Detailed Steps

To effectively follow this tutorial, the developer requires a few essential libraries and integrations. Additionally, there is a ready-made flow scaffold called FCL PWA that contains the completed tutorial code, providing a solid foundation for you to build your Progressive Web App (PWA)!

Dependencies

1. Magic Account: Start by setting up an app on magic.link, during which you will obtain an API key crucial for further steps.
2. Magic SDK: Essential for integrating Magic's functionality in your project, and can be found [here](#).
3. Magic Flow SDK: This SDK enables Magic's integration with Flow. You can install it from [this link](#).
4. Flow Client Library (FCL): As the JavaScript SDK for the Flow blockchain, FCL allows developers to create applications that seamlessly interact with the Flow blockchain and its smart contracts.
5. React: Our project will be built using the React framework.

Setting up PWA and Testing Locally

Initiate the creation of a new React app, opting for the PWA template with the following command:

```
bash
npx create-react-app name-of-our-PWA-app --template cra-template-pwa
```

Ensure that `serviceWorkerRegistration.register()` in `index.js` is appropriately configured to support offline capabilities of your PWA.

Proceed to build your application using your preferred build tool. In this example, we will use Yarn:

```
bash
yarn run build
```

Following the build, you can serve your application locally using:

```
bash
npx serve -s build
```

To thoroughly test your PWA, especially on a mobile device, it's highly recommended to use a tool like ngrok. Start ngrok and point it to the local port your application is running on:

```
bash
ngrok http 3000
```

Grab the generated link, and you can now access and test your PWA directly on your mobile device!

You can now grab the link and go to it on your mobile device to test the PWA!

Integrating with Magic

Proceed to install the Magic-related dependencies in your project. Ensure you add your Magic app's key as an environment variable for secure access:

```
bash
yarn add magic-sdk @magic-ext/flow @onflow/fcl
```

Let's create a helper file, `magic.js`, to manage our Magic extension setup. Ensure that your environment variable with the Magic API key is correctly set before proceeding.

```
js
import { Magic } from "magic-sdk";
import { FlowExtension } from "@magic-ext/flow";

const magic = new Magic(process.env.REACTAPPMAGICKEY, {
  extensions: [
    new FlowExtension({
      rpcUrl: "https://rest-testnet.onflow.org",
      network: "testnet",
    }),
  ],
});

export default magic;
```

Anytime you need to interface with chain you will use this magic instance.

React Context and Provider for User Data

currentUserContext.js

This file creates a React context that will be used to share the current user's data across your application.

React Context: It is created using `React.createContext()` which provides a way to pass data through the component tree without having to pass props down manually at every level.

```
js
import React from "react";

const CurrentUserContext = React.createContext();

export default CurrentUserContext;
```

currentUserProvider.js

This file defines a React provider component that uses the context created above. This provider component will wrap around your application's components, allowing them to access the current user's data.

- useState: To create state variables for storing the current user's data and the loading status.
- useEffect: To fetch the user's data from Magic when the component mounts.
- magic.user.isLoggedIn: Checks if a user is logged in.
- magic.user.getMetadata: Fetches the user's metadata.

```
js
import React, { useState, useEffect } from "react";
import CurrentUserContext from "./currentUserContext";
import magic from "./magic"; // You should have this from the previous
part of the tutorial

const CurrentUserProvider = ({ children }) => {
  const [currentUser, setCurrentUser] = useState(null);
  const [userStatusLoading, setUserStatusLoading] = useState(false);

  useEffect(() => {
    const fetchData = async () => {
      try {
        setUserStatusLoading(true);
        const magicIsLoggedIn = await magic.user.isLoggedIn();
        if (magicIsLoggedIn) {
          const metaData = await magic.user.getMetadata();
          setCurrentUser(metaData);
        }
      } catch (error) {
        console.error("Error fetching user data:", error);
      } finally {
        setUserStatusLoading(false);
      }
    };
    fetchData();
  }, []);

  return (
    <CurrentUserContext.Provider
      value={{ currentUser, setCurrentUser, userStatusLoading }}
    >
      {children}
    </CurrentUserContext.Provider>
  );
};

export default CurrentUserProvider;
```

Logging in the User

This part shows how to log in a user using Magic's SMS authentication.

- `magic.auth.loginWithSMS`: A function provided by Magic to authenticate users using their phone number.
 - `setCurrentUser`: Updates the user's data in the context.

```
js
import magic from "./magic";

const login = async (phoneNumber) => {
    if(!phoneNumber) {
        return;
    }

    await magic.auth.loginWithSMS({ phoneNumber });

    const metaData = await magic.user.getMetadata();
    setCurrentUser(metaData);
};


```

Scripts/Transactions with Flow

This example shows how to interact with the Flow blockchain using FCL and Magic for authorization.

- `fcl.send`: A function provided by FCL to send transactions or scripts to the Flow blockchain.
 - `AUTHORIZATIONFUNCTION`: The authorization function provided by Magic for signing transactions.

```
js
import  as fcl from "@onflow/fcl";
import magic from "./magic";

fcl.config({
  "flow.network": "testnet",
  "accessNode.api": "https://rest-testnet.onflow.org",
  "discovery.wallet": https://fcl-discovery.onflow.org/testnet/authn,
})
const AUTHORIZATIONFUNCTION = magic.flow.authorization;

const transactionExample = async (currentUser) => {
  const response = await fcl.send([
    fcl.transaction
      // Your Cadence code here
    ,
    fcl.args([
      fcl.arg(currentUser.publicAddress, fcl.types.Address),
    ]),
    fcl.proposer(AUTHORIZATIONFUNCTION),
  ])
  return response;
}
```

```

        fcl.authorizations([AUTHORIZATIONFUNCTION]),
        fcl.payer(AUTHORIZATIONFUNCTION),
        fcl.limit(9999),
    ]);
const transactionData = await fcl.tx(response).onceSealed();
}

```

Account Linking with Flow

Now we can unlock the real power of Flow. Lets say you have another Flow account and you want to link the "magic" account as a child account so that you can take full custody of whatever is in the magic account you can do this via Hybird Custody.

You can view the hybrid custody repo and contracts here:
<https://github.com/onflow/hybrid-custody>

We will maintain two accounts within the app. The child(magic) account from earlier and new non custodial FCL flow account. I won't go over how to log in with FCL here and use it but you can do the normal process to obtain the parent account.

Once you have the parent account and child(magic) account logged in you can link the account by using the following transaction.

```

cadence
#allowAccountLinking

import HybridCustody from 0x294e44e1ec6993c6

import CapabilityFactory from 0x294e44e1ec6993c6
import CapabilityDelegator from 0x294e44e1ec6993c6
import CapabilityFilter from 0x294e44e1ec6993c6

import MetadataViews from 0x631e88ae7f1d7c20

transaction(parentFilterAddress: Address?, childAccountFactoryAddress: Address, childAccountFilterAddress: Address) {
    prepare(childAcct: AuthAccount, parentAcct: AuthAccount) {
        // ----- Begin setup of child account -----
        var acctCap =
            childAcct.getCapability<&AuthAccount>(HybridCustody.LinkedAccountPrivatePath)
        if !acctCap.check() {
            acctCap =
                childAcct.linkAccount(HybridCustody.LinkedAccountPrivatePath) !
        }

        if childAcct.borrow<&HybridCustody.OwnedAccount>(from: HybridCustody.OwnedAccountStoragePath) == nil {
            let ownedAccount <- HybridCustody.createOwnedAccount(acct: acctCap)
        }
    }
}
```

```

        childAcct.save(<-ownedAccount, to:
HybridCustody.OwnedAccountStoragePath)
    }

    // check that paths are all configured properly
    childAcct.unlink(HybridCustody.OwnedAccountPrivatePath)

childAcct.link<&HybridCustody.OwnedAccount{HybridCustody.BorrowableAccoun
t, HybridCustody.OwnedAccountPublic,
MetadataViews.Resolver}>(HybridCustody.OwnedAccountPrivatePath, target:
HybridCustody.OwnedAccountStoragePath)

    childAcct.unlink(HybridCustody.OwnedAccountPublicPath)

childAcct.link<&HybridCustody.OwnedAccount{HybridCustody.OwnedAccountPubl
ic, MetadataViews.Resolver}>(HybridCustody.OwnedAccountPublicPath,
target: HybridCustody.OwnedAccountStoragePath)
    // ----- End setup of child account -----
-----

    // ----- Begin setup of parent account -----
var filter: Capability<&{CapabilityFilter.Filter}>? = nil
if parentFilterAddress != nil {
    filter =
getAccount(parentFilterAddress!).getCapability<&{CapabilityFilter.Filter}
>(CapabilityFilter.PublicPath)
}

    if parentAcct.borrow<&HybridCustody.Manager>(from:
HybridCustody.ManagerStoragePath) == nil {
        let m <- HybridCustody.createManager(filter: filter)
        parentAcct.save(<- m, to: HybridCustody.ManagerStoragePath)
    }

    parentAcct.unlink(HybridCustody.ManagerPublicPath)
    parentAcct.unlink(HybridCustody.ManagerPrivatePath)

parentAcct.link<&HybridCustody.Manager{HybridCustody.ManagerPrivate,
HybridCustody.ManagerPublic}>(HybridCustody.OwnedAccountPrivatePath,
target: HybridCustody.ManagerStoragePath)

parentAcct.link<&HybridCustody.Manager{HybridCustody.ManagerPublic}>(Hybr
idCustody.ManagerPublicPath, target: HybridCustody.ManagerStoragePath)
    // ----- End setup of parent account -----
-----

    // Publish account to parent
    let owned = childAcct.borrow<&HybridCustody.OwnedAccount>(from:
HybridCustody.OwnedAccountStoragePath)
        ?? panic("owned account not found")

```

```

        let factory =
getAccount(childAccountFactoryAddress).getCapability<&CapabilityFactory.M
anager{CapabilityFactory.Getter}>(CapabilityFactory.PublicPath)
        assert(factory.check(), message: "factory address is not configured
properly")

        let filterForChild =
getAccount(childAccountFilterAddress).getCapability<&{CapabilityFilter.Fi
lter}>(CapabilityFilter.PublicPath)
        assert(filterForChild.check(), message: "capability filter is not
configured properly")

        owned.publishToParent(parentAddress: parentAcct.address, factory:
factory, filter: filterForChild)

        // claim the account on the parent
        let inboxName =
HybridCustody.getChildAccountIdentifier(parentAcct.address)
        let cap =
parentAcct.inbox.claim<&HybridCustody.ChildAccount{HybridCustody.AccountP
rivate, HybridCustody.AccountPublic, MetadataViews.Resolver}>(inboxName,
provider: childAcct.address)
        ?? panic("child account cap not found")

        let manager = parentAcct.borrow<&HybridCustody.Manager>(from:
HybridCustody.ManagerStoragePath)
        ?? panic("manager no found")

        manager.addAccount(cap: cap)
    }
}

```

:::note

For the sake of this example, well use some pre defined factory and filter implementations. You can find them on the repo but on testnet we can use 0x1055970ee34ef4dc and 0xe2664be06bb0fe62 for the factory and filter address respectively. 0x1055970ee34ef4dc provides NFT capabilities and 0xe2664be06bb0fe62 which is the AllowAllFilter. These generalized implementations likely cover most use cases, but you'll want to weigh the decision to use them according to your risk tolerance and specific scenario

:::

Now, for viewing all parent accounts linked to a child account and removing a linked account, you can follow similar patterns, using Cadence scripts and transactions as required.

```

cadence
import HybridCustody from 0x294e44e1ec6993c6

access(all) fun main(child: Address): [Address] {
    let acct = getAuthAccount(child)

```

```

    let o = acct.borrow<&HybridCustody.OwnedAccount>(from:
HybridCustody.OwnedAccountStoragePath)

    if o == nil {
        return []
    }

    return o!.getParentStatuses().keys
}

```

and finally to remove a linked account you can run the following cadence transaction

```

js
await fcl.send([
    fcl.transaction
    import HybridCustody from 0x294e44elec6993c6

    transaction(parent: Address) {
        prepare(acct: AuthAccount) {
            let owned = acct.borrow<&HybridCustody.OwnedAccount>(from:
HybridCustody.OwnedAccountStoragePath)
            ?? panic("owned not found")

            owned.removeParent(parent: parent)

            let manager =
getAccount(parent).getCapability<&HybridCustody.Manager{HybridCustody.Man
agerPublic}>(HybridCustody.ManagerPublicPath)
                .borrow() ?? panic("manager not found")
                let children = manager.getChildAddresses()
                assert(!children.contains(acct.address), message: "removed
child is still in manager resource")
            }
        }
    ,
    fcl.args([fcl.arg(account, t.Address)]),
    fcl.proposer(AUTHORIZATIONFUNCTION),
    fcl.authorizations([AUTHORIZATIONFUNCTION]),
    fcl.payer(AUTHORIZATIONFUNCTION),
    fcl.limit(9999),
]);

```

Video Guide

! [Video Title] (<https://www.youtube.com/watch?v=1ZmvfBFdCxY> "Video Title")

Sample Flow PWA: Balloon Inflation Game

Game Overview

This PWA game revolves around inflating a virtual balloon, with a twist! The players engage with the balloon, witnessing its growth and color transformation, all while being cautious not to pop it. The ultimate goal is to mint the balloon's state as an NFT to commemorate their achievement.

You can view the game here. Visit this on your mobile device (for iOS use Safari).

The full code for this game can be found here:
<https://github.com/onflow/inflation>

!pwaprompt

! [pwamintballoonthumbnail] (<https://drive.google.com/file/d/15ojzoRTtTN6gQXVN3STMa3-JOZ0b6frw/view>)

! [pwalinkaccountthumbnail] (<https://drive.google.com/file/d/1FZzoLmd5LLGBbO4enzk8LpV1Uwbgc-Ry/view>)

Key Game Features:

1. Balloon Inflation:

- As the player inflates the balloon, it expands and changes color.
- A hidden inflation threshold is set; surpassing this limit will result in the balloon bursting.

2. NFT Minting:

- Satisfied with their balloon's size, players have the option to mint it into an NFT, creating a permanent token of their accomplishment.

3. Balloon Collection:

- Post-minting, players can view and showcase their collection of balloon NFTs.

4. Account Linking and Custody:

- Players initially interact with the game in a walletless fashion via Magic.
- When ready to claim full ownership of their balloon NFTs, they can link their Magic account to a non-custodial FCL wallet of their choice.

Integration with Flow and Magic

The entire game is crafted upon the previously discussed setup, ensuring a seamless and user-friendly experience.

Playing the Game:

- Walletless Interaction: Users can jump right into the game, inflating the balloon and enjoying the gameplay without any blockchain wallet setup.
- Inflation and Visuals: The balloon's size and color change in real-time, providing instant visual feedback to the player.

Minting and Viewing NFTs:

- Magic Login for Minting: To mint their balloon as an NFT, players log in using Magic, embracing a walletless experience.
- Viewing NFT Collection: Post-minting, players can easily access and view their collection of balloon NFTs.

Taking Custody with Account Linking:

- Secure Custody: Players wishing to secure their balloon NFTs can utilize Account Linking to connect their Magic account to their personal non-custodial FCL wallet.
- Full Ownership: This step ensures that players have complete control and custody over their digital assets.

Conclusion

The balloon inflation game stands as a testament to the seamless integration of Flow, Magic, and PWA technology, creating a user-friendly blockchain game that is accessible, engaging, and secure. Players can enjoy the game, mint NFTs, and take full ownership of their digital assets with ease and convenience.

deploying.md:

```
---
title: Deploying Contracts
sidebarlabel: Deploying Contracts
description: Guidelines for deploying your project's contracts on Flow Mainnet
sidebarposition: 3
sidebarcustomprops:
  icon: □
---
```

In order to deploy your smart contracts to the mainnet, you need a funded account. If you want to get started on Testnet, look below for information on how to get started.

```
<Callout type="info">
Make sure you handle your mainnet account keys appropriately. Using a Key Management Service is the best practice.
</Callout>
```

Creating an Account

There are two simple methods of creating an account on testnet. Interactive and Manual, both use the Flow CLI. On mainnet you will have to fund your newly created account, there is no faucet. Make sure to install the Flow CLI. Flow CLI has a interactive mode for generating keys.

```
<Callout type="success">
```

Anyone can deploy and update contracts on mainnet. Audits are encouraged but not mandatory to deploying contracts to mainnet. Take every precaution to reduce issues and protect users.

</Callout>

Create and deploy a mainnet project

The tool of choice is Flow CLI, there are quickstarts and guides that use Flow CLI, Getting Started

- It is highly encouraged to test your contracts, transactions and scripts on Testnet, have strong smart contract test coverage and follow any additional guidelines set out here: Smart Contract Testing Guidelines.
- Follow the Flow CLI instructions to Create a Project. You have the Flow CLI installed and ran flow init in your project folder and generating a flow.json file
- Mainnet account: You completed the mainnet account setup, (see above) and have your key pair and mainnet address ready.
- Deploy your project, notice that your account now has contracts deployed on mainnet.
- Deploy a contract to mainnet. You can deploy contracts individually using the account-add-contract command.

<Callout type="info">

All your contract deployment addresses are stored in flow.json. Mainnet, Testnet and local (emulator) are stored as well.

</Callout>

Deploy updated contracts on mainnet

Contracts can be updated and retain the contract address. You can use the Flow CLI contract update command to re-deploy an updated version of your contract:

<Callout type="warning">

If you see Error Code: 1103, your new account does not have enough funds to complete the transaction. Make sure you have enough FLOW and your account is set up correctly, check Flowdiver to verify.

</Callout>

Once all your contracts are deployed, you can visit flow-view-source or run the Flow CLI get account command to confirm the deployment.

Sporks

Currently, historical event data is not migrated between sporks, so you'll need to design your application with this in mind. We recognize the usefulness of historical event data and plan on adding a means of accessing it in the near future. Past spork transactional data is available, See Previous Spork Access Node Info

More Information on Sporks

Testnet

The Flow test network, known as Flow Testnet, exists to help developers test their software and smart contracts against a live network. It's also used as a means of releasing and testing new protocol and smart contract features before they are integrated into Flow's main network (Mainnet).

When the Flow protocol is updated or a new version of Cadence is released, those updates will always be made available on the Flow Emulator before they're integrated into Flow Testnet or Flow Mainnet.

Getting Started on Testnet

If you need to create a flow.json file to store information about accounts and contracts use the flow init command to create a project

<Callout type="info">

To create accounts and generate keys, make sure to install Flow CLI. Flow CLI provides convenient functions to simplifies interacting with the blockchain.

</Callout>

Creating an Account

There is a simple Flow CLI command to run to create an account. flow accounts create command will create a new account and generate a key pair then add the account to your flow.json. The command will try and You can also use the Testnet Faucet to create and fund an account.

More information about Flow CLI and creating accounts.

Creating and deploying a Project

Flow CLI can be used to create a Cadence project and stay organized, Flow CLI: Create a project. This will make deployment much easier and help with the iterative development process.

After you have a project created and want to deploy your Cadence; contracts, transactions and scripts.

flow accounts add-contract <CONTRACTPATH> --signer <ACCOUNTNAME> --network testnet will deploy your contract to testnet.

More information on how to use Flow CLI to deploy.

Make sure Flow project was initialized in the previous step and the flow.json is present.

Making Use of Core Contracts

Flow Testnet comes with some useful contracts already deployed, called core contracts. More information and import addresses for the core contracts.

Once your accounts are set up and you're ready to develop, you can look over some code examples from the Flow Go SDK.

Breaking Changes

The Flow blockchain is improved continuously and thus version updates to Cadence, Flow node software, and the Flow SDKs will contain important updates as well as breaking changes.

You should anticipate future updates and join the community (Forum or Discord) to stay tuned on important announcements. Notices and guidelines for changes will be provided as early as possible.

Testnet Sporking

"Sporking" (soft forking) is the process of upgrading the Flow network node software and migrating the chain state from one version to another.

Currently, historical event data is not migrated between sporks. You'll need to design your application with this in mind. We recognize the usefulness of historical event data and plan on adding a means of accessing it in the near future. Only one previous spork data is available through old Access Node.

```
<Callout type="warning">
Flow Testnet is explicitly for experimentation and testing and should not
be used to exchange "real value" (e.g. developing a fiat money on/off-
ramp for your testnet application).
</Callout>
```

```
# learn-cadence.md:

---
sidebarposition: 1
slug: /build/learn-cadence
title: Learn Cadence ↗
---

<!-- Add redirect to cadence-lang.org in sidebar -->

<meta http-equiv="refresh" content="0; url=https://cadence-
lang.org/docs/" />

# overview.md:

---
title: Smart Contracts on Flow
```

```
sidebarposition: 2
sidebarlabel: Smart Contracts on Flow
sidebarcustomprops:
  icon: ✎
---
```

At its core, a decentralized application is defined by the smart contracts it uses on the blockchain. Rather than relying on centralized application servers and databases, apps model their core application logic using smart contracts, often referred to as the “on-chain” code.

It is therefore helpful to develop a clear model for your app that takes into account the data and logic that will exist in your smart contracts. In particular, it is important to differentiate between the parts of your app that must live on chain and those that should live off chain.

How to Write Smart Contracts on Flow

Smart contracts on the Flow blockchain are implemented in Cadence, a resource-oriented programming language specifically designed for smart contract development.

Onboard to Cadence

To get started with Cadence, we recommended covering the introductory tutorials available in the Flow Playground, a simple web IDE designed for learning Cadence.

Configure Your Local Environment

To build confidently, you will want to set up the appropriate local environment and have an adequate test suite to ensure your smart contracts operate as intended. To do this, familiarize yourself with the following tools:

- **Flow CLI:** A utility to directly interact with the chain and manage accounts and contracts.
- **Flow Emulator:** A lightweight server that simulates the Flow blockchain (strongly recommended during development).
- **Flow Dev Wallet:** A utility to simulate user wallets in development.
- **Visual Studio Code Extension:** An IDE integration for developing smart contracts.

Storing Data on Flow

All apps will store important data on the blockchain, and some more than others -- especially NFT apps. You'll want to consider the following when storing data on the Flow blockchain.

What does your data need to represent?

Permanence is a key property of blockchains; users trust that the data they store will continue to exist for years to come, and this is a defining characteristic of assets like NFTs. Therefore, well-designed

digital assets store the information necessary to retain their value without external dependencies.

Storage Limits & Fees

However, there are practical constraints to storing data on a blockchain. Developer and user accounts must retain a small amount of FLOW tokens, known as the storage fee, for bytes of data stored in their accounts. The minimum storage fee will grant each account a minimum storage amount. If an account holds assets that demand more bytes of storage, the account will need to retain more FLOW tokens to increase the storage amount according to Flow's fee schedule. A more compact data model can keep storage needs down. \

Furthermore, a single Flow transaction has a size limit of 4MB, which limits the rate at which large amounts of data can be transferred to the blockchain.

Lastly, a blockchain is not a content delivery network and therefore cannot serve media assets, such as videos, at the speeds expected by modern applications.

For these reasons, it usually isn't practical to store large media assets such as videos and high-definition images on the Flow blockchain. Instead, consider using an external storage solution.

External Storage Networks

Decentralized storage networks such as IPFS allow you to store large digital assets off chain, but without relying on centralized servers. Rather than saving an entire asset to the Flow blockchain, you can save the content hash (known as a CID on IPFS) on the blockchain and then store the source file off-chain. This way, users can verify that the media file matches the digital asset.

IPFS files can be uploaded via a pinning service such as Pinata; see their NFT tutorial for an example of how to use Pinata with Flow.

It's worth noting that IPFS files are served through gateways, many of which leverage caching to provide fast response times. Cloudflare provides a public IPFS Gateway, and Pinata also supports dedicated gateways with custom domains.

Using Existing Standards

The Flow blockchain has existing smart contract standards for both fungible and non-fungible tokens that you should implement when building your contracts.

Non-Fungible Tokens (NFTs)

All NFTs on the Flow blockchain implement the NonFungibleToken interface, allowing them to be compatible with wallets, marketplaces and other cross-app experiences.

See the NFT Guide for a guide on how to create a basic NFT contract that conforms to the standard.

- Non-Fungible Token (NFT) contract interface

NFT Sales and Trading

Flow has a standard contract to facilitate both the direct sales and peer-to-peer trading of NFTs. The NFT storefront contract is useful for apps that want to provide an NFT marketplace experience.

- NFT Storefront contract

Fungible Tokens

Fungible tokens (i.e. coins, currencies) on the Flow blockchain, including the default cryptocurrency token FLOW, implement the FungibleToken interface.

See the FT Guide for a guide on how to create a basic fungible token contract that conforms to the standard.

- Fungible Token contract interface

```
# testing.md:
```

```
---
title: Testing Your Contracts
sidebarlabel: Testing Your Contracts
description: Testing smart contract Guidelines
sidebarposition: 4
sidebarcustomprops:
  icon: 📃
---
```

Testing is an essential part of smart contract development to ensure the correctness and reliability of your code. The Cadence Testing Framework provides a convenient way to write tests for your contracts, scripts and transactions which allows you to verify the functionality and correctness of your smart contracts.

Install Flow CLI

The Flow CLI is the primary tool for developing, testing, and deploying smart contracts to the Flow network.

If you haven't installed the Flow CLI yet and have homebrew installed, simply run brew install flow-cli. Alternatively, refer to the Flow CLI installation instructions.

Create a new project

In your preferred code editor, create a new directory for your project and navigate to it in the terminal. Then initialize a new Flow project by running the command flow init. This will create a flow.json config file that contains the project's configuration.

```
bash
mkdir test-cadence
cd test-cadence
flow init
```

Write a simple smart contract

In your code editor, create a new file called calculator.cdc and add the following code:

```
cadence calculator.cdc
access(all) contract Calculator {
    access(all)
    fun add(a: Int, b: Int): Int {
        return a + b
    }

    access(all)
    fun subtract(a: Int, b: Int): Int {
        return a - b
    }

    access(all)
    fun multiply(a: Int, b: Int): Int {
        return a * b
    }
}
```

Add the smart contract to the config

Next up, we need to add our new contract in the contracts key in the flow.json config file. More specifically, we need to add the contract name, location and an address alias for the testing environment.

```
json
{
    "contracts": {
        "Calculator": {
            "source": "./calculator.cdc",
            "aliases": {
                "testing": "0x0000000000000007"
            }
        }
    },
    "networks": {...},
    "accounts": {...},
    "deployments": {...}
```

```
}
```

For the time being, the address for the testing alias, can be one of:

- 0x0000000000000005
- 0x0000000000000006
- 0x0000000000000007
- 0x0000000000000008
- 0x0000000000000009
- 0x000000000000000a
- 0x000000000000000b
- 0x000000000000000c
- 0x000000000000000d
- 0x000000000000000e

In the next release, there will be 20 addresses for contract deployment during testing.

Write unit tests

In the same directory, create a new file called calculatortest.cdc and add the following code:

```
cadence calculatortest.cdc
import Test
import "Calculator" // contract name from the previous step

access(all)
fun setup() {
    let err = Test.deployContract(
        name: "Calculator",
        path: "./calculator.cdc",
        arguments: []
    )
    Test.expect(err, Test.beNil())
}

access(all)
fun testAdd() {
    Test.assertEqual(5, Calculator.add(a: 2, b: 3))
}

access(all)
fun testSubtract() {
    Test.assertEqual(2, Calculator.subtract(a: 5, b: 3))
}
```

This code:

- imports the Calculator contract from the calculator.cdc file (according to flow.json)
- deploys the Calculator contract to the address specified in the testing alias
- defines two test cases: testAdd() and testSubtract()

- calls add() and subtract() methods with different input values respectively.

Running the test cases

To run the test cases, use the following command in the terminal:

```
bash
flow test --cover --covercode="contracts" calculatortest.cdc
```

This command uses the Flow CLI to run the test cases and display the output. You should see the following output:

```
bash
Test results: "calculatortest.cdc"
- PASS: testAdd
- PASS: testSubtract
Coverage: 66.7% of statements
```

This output indicates that both test cases ran successfully, and the two smart contract methods are functioning as expected. With the supplied flags (--cover & --covercode="contracts"), we also get code coverage insights for the contracts under testing. The code coverage percentage is 66.7%, because we have not added a test case for the multiply method. By viewing the auto-generated coverage.json file, we see:

```
json
{
  "coverage": {
    "A.0000000000000007.Calculator": {
      "linehits": {
        "14": 0,
        "4": 1,
        "9": 1
      },
      "missedlines": [
        14
      ],
      "statements": 3,
      "percentage": "66.7%"
    }
  }
}
```

Line 14 from the Calculator smart contract is marked as missed. This is the line:

```
cadence
return a b
```

which is the multiply method.

By adding a test case for the above method:

```
cadence calculatortest.cdc
...
access(all)
fun testMultiply() {
    Test.assertEqual(10, Calculator.multiply(a: 2, b: 5))
}
```

our code coverage percentage goes to 100%:

```
bash
flow test --cover --covercode="contracts" calculatortest.cdc

Test results: "calculatortest.cdc"
- PASS: testAdd
- PASS: testSubtract
- PASS: testMultiply
Coverage: 100.0% of statements
```

Advanced Testing Techniques

The Cadence testing framework provides various features and techniques for writing comprehensive test scenarios. Some of these include:

- **Code Coverage:** You can use the `--cover` flag with the `flow test` command to view code coverage results when running your tests. This allows you to identify areas of your code that are not adequately covered by your test inputs.
- **Test Helpers:** Test helpers are reusable functions that help you set up the initial state for your test files. You can define test helpers in a Cadence program and use them in your test files by importing it whenever needed.
- **Assertions:** The testing framework provides built-in assertion functions, such as `assertEqual`, `beNil`, `beEmpty`, `contain`, to help you verify the expected behavior of your smart contracts.
- **Test Suites:** You can organize your test files into test suites to improve the readability and maintainability of your test code. Test suites allow you to group related test cases and set up common test helpers for all the tests in the suite.
- **Integration tests:** In our previous example, we would directly call the available methods on the contract under test. This is generally categorized as unit testing. You can also write integration tests, by executing scripts & transactions to interact with the contracts under testing. If you would like to write your tests in Go, instead of Cadence, you can use Overflow tool to run integration tests against either an local emulator, testnet, mainnet or an in memory instance of the flow-emulator.

By leveraging these advanced testing techniques, you can write more robust and reliable smart contracts in Cadence. In this example, we set up a basic testing environment, wrote a simple smart contract in Cadence, and created a test file to verify its functionality. We then used the Flow CLI to run the test file and confirm that the smart contract is working correctly.

This is a basic example, and there are many more advanced features and techniques you can explore when working with the Cadence Testing Framework.

For more in-depth tutorials and documentation, refer to the official Cadence language documentation and the Flow CLI documentation.

Testing Requirements

It is suggested to follow the following best practices:

- Every publicly exposed feature of a contract and its resources should have unit tests that check both for success with correct input and for failure with incorrect input.

These tests should be capable of being run locally with the Flow emulator, with no or minimal extra resources or configuration, and with a single command.

- Each user story or workflow that uses the smart contracts should have an integration test that ensures that the series of steps required to complete it does so successfully with test data.

Make sure you test all contracts - and the integration into your application extensively before proceeding to the mainnet.

You should aim to replicate all conditions as closely as possible to the usage patterns on mainnet.

Writing Tests

There are official SDKs/frameworks for Flow in Cadence, Go and JavaScript.

In all three cases, the test code will need to deploy the contracts, configure accounts to interact with them and send transactions to them. It will then have to wait for the transactions to be sealed and check the results by catching exceptions, checking for events, and querying state using scripts.

Cadence tests

Cadence comes with built-in support for code coverage, as well as a native testing framework which allows developers to write their tests using Cadence.

This framework is bundled with the Flow CLI tool, which includes a dedicated command for running tests (flow test).

You can find examples of Cadence tests in the following projects: hybrid-custody, flow-nft, flow-ft.

Visit the documentation to view all the available features.

The Hybrid Custody project is a prime example which utilizes both the Cadence testing framework and code coverage in its CI.

!Hybrid Custody CI

There is also a repository which contains some sample contracts and their tests.

!Automated CI Coverage Report

!Coverage Report Visualization

```
<Callout type="info">
The Cadence testing framework utilizes the emulator under the hood.
</Callout>
```

Go Tests

Tests in Go can be written using flow-go-sdk and the go test command.

You can find examples of Go tests in the following projects: flow-core-contracts, flow-nft, flow-ft.

```
<Callout type="info">
These tests are tied to the emulator but can be refactored to run on
testnet
</Callout>
```

Testing Your Application

Automated Testing of Contract Code

All contracts should include test coverage for all contract functions. Make sure you've accounted for success and failure cases appropriately.

Tests should also be runnable in automated environments (CI). You can use the Cadence testing utils to create tests for your smart contract code.

Stress Testing Live Applications Before Mainnet

Once you deployed your application to the testnet, you should record how your application handles non-trivial amounts of traffic to ensure there are no issues.

```
<Callout type="success">
Get familiar with the Cadence anti-patterns to avoid avoid problematic or
unintended behavior.
</Callout>
```

References

- Reference documentation for Cadence testing
- Overflow is a powerful Golang-based DSL for efficient testing and execution of blockchain interactions
- projects that have good examples of robust test cases:
 - hybrid-custody,
 - flow-nft,
 - flow-ft.

```
# contract-upgrades.md:
```

```
---
title: Contract Upgrades with Incompatible Changes
sidebarposition: 4
---
```

Problem

I have an incompatible upgrade for a contract. How can I deploy this?

Solution

Please don't perform incompatible upgrades between contract versions in the same account.

There is too much that can go wrong.

You can make compatible upgrades and then run a post-upgrade function on the new contract code if needed.

If you must replace your contract rather than update it, the simplest solution is to add or increase a suffix on any named paths in the contract code (e.g. /public/MyProjectVault becomes /public/MyProjectVault002) in addition to making the incompatible changes, then create a new account and deploy the updated contract there.

⚠ Flow identifies types relative to addresses, so you will also need to provide upgrade transactions to exchange the old contract's resources for the new contract's ones. Make sure to inform users as soon as possible when and how they will need to perform this task.

If you absolutely must keep the old address when making an incompatible upgrade, then you do so at your own risk. Make sure you perform the following actions in this exact order:

1. Delete any resources used in the contract account, e.g. an Admin resource.
2. Delete the contract from the account.
3. Deploy the new contract to the account.

⚠ Note that if any user accounts contain structs or resources from the old version of the contract that have been replaced with incompatible versions in the new one, they will not load and will cause transactions that attempt to access them to crash. For this reason, once any users

have received structs or resources from the contract, this method of making an incompatible upgrade should not be attempted!

```
# project-development-tips.md:
```

```
---
```

```
title: Flow Smart Contract Project Development Standards
```

```
sidebarlabel: Development Standards
```

```
sidebarposition: 5
```

```
description: "Learn how to effectively organize and manage a Cadence project"
```

```
---
```

Smart Contract Project Development Standards

Context

Smart Contracts are the bedrock piece of security for many important parts of the Flow blockchain, as well as for any project that is deployed to a blockchain.

They are also the most visible technical parts of any project, since users will be querying them for data, building other smart contracts that interact with them, and using them as learning materials and templates for future projects. Furthermore, when deployed they are publicly available code on the blockchain and often also in public Github repos.

Therefore, the process around designing, building, testing, documenting, and managing these projects needs to reflect the critical importance they hold in the ecosystem.

Every software project strikes a balance between effort spent on product/feature delivery vs the many other demands of the software development lifecycle, whether testing, technical debt, automation, refactoring, or documentation etc. Building in Web3 we face the same trade-offs, but in a higher risk and consequence environment than what is typical for most software. A mismanaged or untested smart contract may result in significant financial losses as a result of vulnerabilities which were overlooked then exploited. We highly recommend builders adopt these best practices to help mitigate these risks.

If they do so, they will be able to build better smart contracts, avoid potential bugs, support user and third-party adoption of their projects, and increase their chances of success

by being a model for good software design. Additionally, the more projects that adopt good software design and management standards normalizes this behavior, encouraging other projects in the ecosystem to do the same which creates a healthier and more vibrant community.

Ensuring appropriate levels of testing results in better smart contracts which have pro-actively modeled threats and engineered against them. Ensuring appropriate levels of standards adoption (FungibleToken, NFT Metadata, NFT StoreFront, etc) by dapp builders amplifies the network effects for all in the ecosystem. NFTs in one dapp can be readily consumed by other dapps through on-chain events with no new integration required. With your help and participation we can further accelerate healthy and vibrant network effects across the Flow ecosystem!

Some of these suggestions might seem somewhat unnecessary, but it is important to model what a project can do to manage its smart contracts the best so that hopefully all of the other projects follow suit.

This also assumes standard software design best practices also apply. Indeed, many of these suggestions are more general software design best practices, but there may be others that are assumed but not included here.

Implementing These Practices

This document serves as mostly an outline of best practices the projects should follow. As with all best practices, teams will choose which applies to them and their work process, however, we recommend that teams explicitly define a minimum acceptable set of standards for themselves along with the mechanisms to ensure they are being observed.

Some teams may also have their own set of development standards that achieve a similar goal to these. These recommendations are not meant to be the only paths to success, so if a team disagrees with some of these and wants to do things their own way, they are welcome to pursue that. This document just shows some generic suggestions for teams who might not know how they want to manage their project.

Design Process

Smart contracts usually manage a lot of value, have many users, and are difficult to upgrade for a variety of reasons. Therefore, it is important to have a clearly defined design process for the smart contracts before much code is written so that the team can set themselves up for success.

Here are some recommendations for how projects can organize the foundations of their projects.

Projects should ensure that there is strong technical leadership for their smart contracts

Developing a dapp requires a clear vision for the role of the smart contract and how it's integrated.

Security vulnerabilities may arise from bugs directly in smart contract code (and elsewhere in the system).

Asynchronous interaction vectors may lead to forms of malicious abuse, DOS etc in a contract triggering explosive gas costs for the developer or other problems.

We recommend that engineers leading a project and deploying to mainnet have an understanding of software and security engineering fundamentals and have been thorough in their Cadence skills development. More in-depth resources for learning Cadence are available [here](#).

The technical leader should be someone who understands Cadence well and has written Cadence smart contracts before. Production-level smart contracts are not the place for beginners to get their start.

It should be this person's responsibility to lead design discussions with product managers and the community, write most of the code and tests, solicit reviews, make requested changes and make sure the project gets completed in a timely manner.

The leader should also understand how to sign transactions with the CLI to deploy/upgrade smart contracts, run admin transactions, and troubleshoot problems, etc.

If something goes wrong in relation to the smart contract that needs to be handled with a bespoke transaction, it is important that the owner knows how to build and run transactions and scripts safely to address the issues and/or upgrade the smart contracts.

The project should also have a clear plan of succession in case the original owner is not available or leaves the project. It is important that there are others who

can fill in who have a clear understanding of the code and requirements so they can give good feedback, perform effective reviews, and make changes where needed.

Projects should maintain a well-organized open source Repo for their smart contracts

As projects like NBA Topshot have shown, when a blockchain product becomes successful others can and do to build on top of what you are doing. Whether that is analytics, tools, or other value adds that could help grow your project ecosystem, composability is key and that depends on open source development. If there isn't already an open source repo, builders should consider creating one.

Builders can start from the the Flow open source template and make sure all of their repo is set up with some initial documentation for what the repo is for before any code is written. External developers and users should have an easily accessible home page to go to to understand any given project.

The repo should also have some sort of high-level design document that lays out the intended design and architecture of the smart contract. The project leads should determine what is best for them to include in the document, but some useful things to include are basic user stories, architecture of the smart contracts, and any questions that still need to be answered about it.

- Where applicable, diagrams should be made describing state machines, user flows, etc.
- This document should be shared in an issue in the open source repo where the contracts or features are being developed, then later moved to the README or another important docs page.

A high level design is a key opportunity to model threats and understand the risks of the system. The process of collaborating and reviewing designs together helps ensure that more edge-cases are captured and addressed.

It's also a lot less effort to iterate on a design than on hundreds of lines of Cadence.

Development Process Recommendations

The Development process should be iterative, if possible

The project should develop an MVP first, get reviews, and test thoroughly, then add additional features with tests. This ensures that the core features are designed thoughtfully and makes the review process easier because they can focus on each feature

one at a time instead of being overwhelmed by a huge block of code.

Comments and field/function descriptions are essential!

Our experience writing many Cadence smart contracts has taught us how important documentation is. It especially matters what is documented and for whom, and in that way we are no different from any software language. The Why is super important, if for example something - an event - that happens in one contract leads to outcomes in a different contract. The What helps give context, the reason for the code turning out the way it is. The How, you don't document - you've written the code. Comments should be directed to those who will follow after you in changing the code.

Comments should be written at the same time (or even before) the code is written.

This helps the developer and reviewers understand the work-in-progress code better, as well as the intentions of the design (for testing and reviewing). Functions should be commented with a

- Description
- Parameter descriptions
- Return value descriptions

Top Level comments and comments for types, fields, events, and functions should use /// (three slashes) to be recognised by the Cadence Documentation Generator.

Regular comments within functions should only use two slashes (//)

Testing Recommendations

Summarized below is a list of testing related recommendations which are noteworthy to mention for a typical smart contract project.

Popular testing frameworks to use for cadence are listed here:

- Cadence: Cadence Testing Framework
- Go: Overflow

The same person who writes the code should also write the tests. They have the clearest understanding of the code paths and edge cases.

Tests should be mandatory, not optional, even if the contract is copied from somewhere else.

There should be thorough emulator unit tests in the public repo. See the flow fungible token repo for an example of unit tests in javascript.

Every time there is a new Cadence version or emulator version,

the dependencies of the repo should be updated to make sure the tests are all still passing.

Tests should avoid being monolithic;
Individual test cases should be set up for each part of the contract to test them in isolation.
There are some exceptions, like contracts that have to run through a state machine
to test different cases. Positive and negative cases need to be tested.

Integration tests should also be written to ensure that your app and/or backend can interact properly with the smart contracts.

Managing Project Keys and Deployments

Smart contract keys and deployments are very important and need to be treated as such.

Private Keys should be stored securely

Private Keys for the contract and/or admin accounts should not be kept in plain text format anywhere.

Projects should determine a secure solution that works best for them to store their private keys.

We recommend storing them in a secure key store such as google KMS or something similar.

Deployments to Testnet or Mainnet should be handled transparently

As projects become more successful, communities around them grow.
In a trustless ecosystem, that also means more of others building on your contracts.

Before deploying or upgrading a contract, it is important to maintain clear community communications with sufficient notice, since changes will always bring added risk.

Giving community members time to review and address issues with upgrades before they happen builds trust and confidence in projects.

Here are a few suggestions for how to manage a deployment or upgrade.

- Communicate to all stakeholders well in advance
 - Share the proposal with the community at least a week in advance (unless it is a critical bug fix)
 - Examples of places to share are your project's chat, forum, blog, email list, etc.
 - This will allow the community and other stakeholders to have plenty of time to view the upcoming changes and provide feedback if necessary.
 - Share the time of the deployment and the deployment transaction with branch/commit hash information to ensure the transaction itself is correct.
 - Coordinate deployment with stakeholders to make sure it is done correctly and on time.

Responsibilities to the Community

Web3 brings tremendous possibilities for engineering applications with trustlessness and composability in mind, with Cadence and Flow offering unique features to achieve this. If every project treats their community and the Flow community with respect and care, the things we can all build together will be very powerful.

Projects should have thorough documentation

Encouraging adoption of project contracts to the broader ecosystem raises the bar around code providing clear high-level descriptions, with detailed and useful comments within contracts, transactions, and scripts.

The more that a project can be understood, that it adheres to standards, and can be built upon with ease, the more likely others will build against it in turn.

Each project should have a detailed README.md with these sections:

- Explanation of the project itself with links to the app
- Addresses on various networks
- High-level technical description of the contracts with emphasis on important types and functionality
 - Architecture diagram (if applicable)
 - Include links to tutorials if they are external
 - Flow smart contract standards that a project implements

Additionally, each contract, transaction, and script should have high-level descriptions

at the top of their files. This way, anyone in the community can easily come in and understand what each one is doing without having to parse confusing code.

Projects should engage with and respond to their own Community

Once a contract is deployed, the work doesn't stop there. Project communities require ongoing nurturing and support. As the developer of a public project on a public blockchain, the owners have an obligation to be helpful and responsive to the community so that they can encourage composability and third party interactions.

- Keep issues open in the repo.
- The owner should turn on email notifications for new issue creation in the repo.
- Respond to issues quickly and clean up unimportant ones.
- Consider blog posts to share more details on technical aspects of the project and upcoming changes.

Projects should contribute to the greater Flow and Cadence community

Flow has a vibrant and growing community of contributors around the world.

Through our mutual collaboration we've had numerous community Flow Improvement Proposals (FLIPs) shipped.

If you have an interest in a particular improvement for Flow or Cadence, we host open meetings which you are welcome to join (announced on discord)

and can participate anytime on any of the FLIPs already proposed.

Responsible project maintainers should contribute to discussions about important proposals (new cadence features, standard smart contracts, metadata, etc) and generally be aware about evolving best practices and anti-pattern understandings.

Projects who contribute to these discussions are able to influence them to ensure

that the language/protocol changes are favorable to them and the rest of the app developers in the ecosystem.

It also helps the owner to promote the project and themselves.

Resources for Best Practices:

- cadence/design-pattern
- cadence/anti-patterns
- cadence/security-best-practices

Composability and extensibility should also be priorities while designing, developing, and documenting their projects. (Documentation for these topics coming soon)

If you have any feedback about these guidelines, please create an issue in the onflow/cadence-style-guide repo or make a PR updating the guidelines so we can start a discussion.

security-best-practices.md:

```
---
```

title: Cadence Security Best Practices
sidebarlabel: Security Best Practices
sidebarposition: 3

This is an opinionated list of best practices Cadence developers should follow to write more secure Cadence code.

Some practices listed below might overlap with advice in the Cadence Anti-Patterns section, which is a recommended read as well.

References

References are ephemeral values and cannot be stored. If persistence is required, store a capability and borrow it when needed.

References allow freely upcasting and downcasting, e.g. a restricted type can be cast to its unrestricted type which will expose all access(all) functions and fields of the type.

So even if your capability uses an interface to restrict its functionality, it can still be downcasted to expose all other public functionality.

Therefore, any privileged functionality in a resource or struct that will have a public capability needs to have entitled accecss, for example access(Owner). Then, the only way to access that functionality would be through an entitled reference, like <auth(Owner) &MyResource>.

Account Storage

Don't trust a users' account storage. Users have full control over their data and may reorganize it as they see fit. Users may store values in any path, so paths may store values of "unexpected" types. These values may be instances of types in contracts that the user deployed.

Always borrow with the specific type that is expected. Or, check if the value is an instance of the expected type.

Authorized Accounts

Access to an &Account gives access to whatever is specified in the account entitlements

list when that account reference is created.

Therefore, avoid using Account references as a function parameter or field unless absolutely necessary and only use the minimal set of entitlements required for the specified functionality so that other account functionality cannot be accessed.

It is preferable to use capabilities over direct &Account references when exposing account data. Using capabilities allows the revocation of access by unlinking and limits the access to a single value with a certain set of functionality.

Capabilities

Don't store anything under the public capability storage unless strictly required. Anyone can access your public capability using Account.capabilities.get. If something needs to be stored under /public/, make sure only read functionality is provided by restricting privileged functions with entitlements.

When publishing a capability, the capability might already be present at the given PublicPath.

In that case, Cadence will panic with a runtime error to not override the already published capability.

It is a good practice to check if the public capability already exists with `account.capabilities.get().check` before creating it. This function will return nil if the capability does not exist.

If it is necessary to handle the case where borrowing a capability might fail, the `account.check` function can be used to verify that the target exists and has a valid type.

Ensure capabilities cannot be accessed by unauthorized parties. For example, capabilities should not be accessible through a public field, including public dictionaries or arrays. Exposing a capability in such a way allows anyone to borrow it and perform all actions that the capability allows.

Transactions

Audits of Cadence code should also include transactions, as they may contain arbitrary code, just, like in contracts. In addition, they are given full access to the accounts of the transaction's signers, i.e. the transaction is allowed to manipulate the signers' account storage, contracts, and keys.

Signing a transaction gives access to the `&Account`, i.e. access to the account's storage, keys, and contracts depending on what entitlements are specified.

Do not blindly sign a transaction. The transaction could for example change deployed contracts by upgrading them with malicious statements, revoking or adding keys, transferring resources from storage, etc.

Types

Use restricted types and interfaces. Always use the most specific type possible, following the principle of least privilege. Types should always be as restrictive as possible, especially for resource types.

If given a less-specific type, cast to the more specific type that is expected. For example, when implementing the fungible token standard, a user may deposit any fungible token, so the implementation should cast to the expected concrete fungible token type.

Access Control

Declaring a field as `access(all)` only protects from replacing the field's value, but the value itself can still be mutated if it is mutable. Remember that containers, like dictionaries, and arrays, are mutable.

Prefer non-public access to a mutable state. That state may also be nested. For example, a child may still be mutated even if its parent exposes it through a field with non-settable access.

Do not use the access(all) modifier on fields and functions unless necessary. Prefer access(self), acccess(Entitlement), or access(contract) and access(account) when other types in the contract or account need to have access.

```
# auditors.md:
```

```
---
```

```
sidebarposition: 7
```

```
description: |
```

```
    Third-party auditors for Cadence contract auditing
```

```
sidebarcustomprops:
```

```
    icon: 🔎
```

```
---
```

Auditors

The following companies provide independent contract auditing services of Cadence smart contracts for the Flow ecosystem.

```
<div id="cards" class="cards">
```

Nagra (Kudelski)

Nagra a multinational provider of security services including blockchain security and smart contract auditing of Cadence.

NCC Group

NCC Group a global provider of security services including blockchain security and smart contract auditing of Cadence.

QuantStamp Inc.

Quantstamp Inc. specializing in blockchain security and smart contract auditing including Cadence.

Halborn Inc.

Halborn Inc. specializing in blockchain security and smart contract auditing including Cadence.

Oak Security

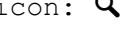
Oak Security a specialist in smart contract auditing security including Cadence.

Emerald City Academy

Emerald City Academy - Shield a Cadence-specialized, community-driven auditing service also offering development support and review for Cadence smart contracts.

```
</div>

# block-explorers.md:

---
sidebarposition: 2
description: |
  User-friendly online tools that provide visual representations of
  blockchain data, facilitating easy navigation through transactions,
  blocks, addresses, and smart contracts while ensuring real-time insights
  and transparency.
sidearcustomprops:
  icon: 
---

```

Flow Block Explorers

Block explorers are user-friendly online tools that visually present blockchain data, allowing users to easily navigate transactions, blocks, addresses, and smart contracts while providing real-time insights and transparency.

```
<div id="cards" className="cards">
```

Flow Diver

Flow Diver provides various tools to explore the blockchain, including:

- Blocks
- Transactions
- Contracts
- Network activity (transaction count)
- Transaction cost (gas fee)
- Validators information

Supported networks:

- Mainnet
- Testnet

Flow View

Flow View offers comprehensive tools to view accounts' information, including:

- Address, balance, and storage
- Public keys and their weights
- Staking information
- Tokens, collections, and listings
- Deployed smart contracts

Supported networks:

- Mainnet
- Testnet
- Emulator (local)

Contract Browser

Contract Browser provides tools for viewing deployed smart contracts, their dependencies, and dependents. Features include:

- Search, view, and verify smart contract source code
- Detailed smart contract information

Testnet Contract Browser is also available for Testnet Flow.

Flowscan [EVM]

Flowscan is based off of the popular Blockscout and provides a user-friendly interface to search for specific EVM transactions, blocks, monitor network health, and track the status of various smart contracts and tokens on for EVM on Flow. Features include:

- Real-Time Data: View live transaction data, blocks, and account activities.
- Search Functionality: Search for specific transactions, blocks, and accounts.
- Smart Contract Analysis: Examine smart contract statuses and interactions.
- Token Tracking: Monitor the creation and transfer of various tokens on the Flow network.

Supported networks:

- Testnet
- Mainnet

</div>

bridges.md:

```
---
sidebarposition: 3
description: Mechanisms that connect different blockchain networks,
allowing secure and decentralized transfer of assets and data across
platforms.
sidearcustomprops:
  icon: 🛡
---
```

Bridges

Bridges are mechanisms that connect different blockchain networks, enabling secure and decentralized transfers of assets and data across various platforms.

```
<div id="cards" className="cards">

Celer cBridge

Celer cBridge is a decentralized and non-custodial asset bridge that supports more than 150 tokens across over 40 blockchains and layer-2 rollups. It is built on top of the Celer Inter-chain Message Framework. cBridge has facilitated over $13 billion in cross-chain asset transfer volume across 40+ blockchains for more than 350,000 unique users. It is rapidly growing and expanding to support more blockchains and layer-2 solutions.

Axelar

Axelar is a decentralized cross-chain network connecting over 55 blockchains, facilitating asset transfers and smart contract programmability. It features a proof-of-stake consensus for security and supports cross-chain applications through General Message Passing (GMP). Integrations with platforms like Squid enable easy token swaps across networks like Ethereum and Polygon.

</div>
```

```
# faucets.md:

---
sidebarposition: 5
description: Get free Flow tokens for testing. Faucets are like taps for tokens, useful for trying Flow without buying tokens.
sidebarcustomprops:
  icon: ♫
---
```

Faucets

Network Faucets provide free Flow tokens for testing purposes, functioning like taps that dispense tokens. They are valuable tools for experimenting with Flow without the need to purchase tokens.

```
<div class="cards">

Flow Faucet

Flow Faucet is a dedicated tool that provides a seamless way to acquire small amounts of Flow tokens for testing and development purposes on the Flow blockchain's testnet environment.
```

Supported Networks

- Testnet

LearnWeb3 Flow Faucet

LearnWeb3 Flow Faucet is a community faucet tool that provides a seamless way to acquire small amounts of Flow tokens for testing and development purposes on the Flow blockchain's testnet environment.

Supported Networks

- Testnet

</div>

Using Flow Faucet

Funding Your Account

If you already have a Flow account, you can fund it directly from the Faucet's landing page. Simply paste the address of the account you want to fund, complete the CAPTCHA, and click "Fund Your Account."

!fund-your-account

After a few seconds, you'll see your account's FLOW balance as a confirmation. Note, the Faucet will automatically determine if the address you paste is a Flow or EVM address and will fund the account accordingly.

Creating a Flow Account

Generate a Key Pair

To create a Flow-native account, you'll need to generate a key pair. You can do this most easily Flow CLI with the keys generate command

```
sh
flow keys generate
```

You'll receive a private key and a public key pair with default ECDSAP256 signature and SHA3256 hash algorithms.

```
sh
> flow keys generate
```

● Store private key safely and don't share with anyone!

Private Key	<PRIVATEKEY>
Public Key	<PUBLICKEY>
Mnemonic	<MNEMONICPHRASE>
Derivation Path	m/44'/539'/0'/0/0
Signature Algorithm	ECDSAP256

You can then use the public key to create a new Flow account on the Faucet. Copy the resulting public key for the next step.

Create a Flow-Native Account

From the Faucet's landing page, click on the "Create Account" button. You'll be prompted to enter your public key. Paste the public key you generated using the Flow CLI and click "Create Account."

:::tip

Know that there is a distinction between Flow native accounts and EVM accounts. Native accounts allow you to interact with the Cadence runtime, while EVM accounts are used for interacting with Flow's EVM. To create an EVM account, you can use EVM tooling to generate an Ethereum Owned Account (EOA) and simply fund the associated address. Alternatively, you can create an EVM account controlled by your Flow native account - known as a Cadence Owned Account (COA) - in which case you'll need a Flow native account and should continue with the steps below.

For more information interacting with EVM via COAs, see the [Interacting With COAs](#) documentation.

:::

`!create-flow-account`

You can then paste your public key into the input field, complete the CAPTCHA, and click "Create Account."

`!input-public-key`

You'll be met with a confirmation screen, showing your Flow account address and the funded balance.

`!account-created`

Using your Flow Account

Once your account has been created, you can add the account to your `flow.json` configuration file under the `accounts` attribute, like so:

```
json
{
  "accounts": {
    "testnet-dev-account": {
      "address": "<YOURADDRESS>",
      "key": "<PRIVATEKEY>"
    }
  }
}
```

:::warning

If you plan on using your `flow.json` in a production environment, you'll want to look at alternative methods to manage your keys more securely, at

minimum using environment variables instead of storing your account private keys in plain text. See [How to Securely Use CLI](#) for more information on alternate key management strategies and how to configure them in your `flow.json` file.

:::

After adding your account to your `flow.json` file, you're ready to use your account in your project. You can now deploy contracts, run transactions, and interact with the Flow blockchain using your new account.

wallets.md:

```
---
```

sidebarposition: 1
description: Store, manage, and interact securely with tokens and digital assets on Flow. Discover a range of wallets that offer convenient ways to handle and safeguard your cryptocurrency holdings, ensuring easy access and enhanced security for your transactions and assets.
sidebarcustomprops:
 icon: 

```
---
```

Wallets

Store, manage, and interact securely with tokens and digital assets on Flow. Discover a range of wallets that offer convenient ways to handle and safeguard your cryptocurrency holdings, ensuring easy access and enhanced security for your transactions and assets.

<div id="cards" className="cards">

Flow Wallet

Flow Wallet is Flow's oldest wallet extension and seamlessly interacts with all Cadence and EVM dApps in the ecosystem.

- <https://wallet.flow.com/>
- <https://frw.gitbook.io/>

Dapper Wallet

Dapper Wallet is a wallet exceptionally friendly for first time crypto collectors to buy and manage digital assets.

<https://www.meetdapper.com/>

Ledger

Ledger is a hardware wallet to secure, buy, exchange, and grow your crypto assets.

<https://www.ledger.com/>

NuFi

NuFi is a non-custodial wallet with staking and Ledger support.

<https://nu.fi/>

Magic.link

Magic is a developer SDK that integrates with your application to enable passwordless Web3 onboarding (no seed phrases) and authentication using magic links (similar to Slack and Medium).

<https://magic.link/>

Niftory

Niftory is a developer platform that offers APIs to create wallets and interact with NFTs.

<https://niftory.com/>

Finoa

Finoa is a platform for institutional investors to safely store and stake their FLOW tokens.

<https://www.finoa.io/flow/>

</div>

Blocto

Blocto is a cross-chain mobile wallet for IOS and Android devices.

<https://www.blocto.io/>

Metamask Wallet

Metamask is a secure and user-friendly crypto wallet for NFTs and digital tokens. Presently only supports EVM on Flow and cannot access Cadence accounts.

<https://metamask.io/>

Flow Dev Wallet

Looking for a way to create mock developer wallets to test your app? The Flow Dev Wallet simulates the protocols used by FCL to interact with the Flow blockchain on behalf of simulated user accounts.

about.md:

```
---
```

title: Why EVM on Flow
sidebarlabel: Why EVM on Flow
sidebarposition: 1

Why EVM on Flow

Flow is an L1 that now supports EVM-equivalency. This means that all of Flow's protocol benefits, such as fast transactions, low costs, and mainstream scalability, are natively available without any additional code changes to solidity contracts. With EVM, solidity devs and builders can now easily tap into Flow's user base and unique IPs without any implementation risk.

Seamless Integration for Ethereum Developers

EVM on Flow is designed to work out-of-the-box with the Ethereum toolchain or other clients. Native EVM transactions also continue to be supported when using Metamask and other EVM-compatible clients. EVM-equivalency on Flow works behind-the-scenes by implementing a minimal transaction script in Cadence, Flow's smart contract language, to integrate Flow features with EVM. This is made possible because EVM transactions are composed and executed within Cadence transactions, enabling novel use-cases and patterns for integration.

Best-In-Class UX

Flow allows for the creation of app on-boarding experiences that meet every type of user exactly where they are at, from web3 beginners to ecosystem veterans. This is possible through Account Linking, which utilizes the account abstraction model on Flow and enables users to immediately use an app without wallet authentication. On-chain accounts can be created as needed by the application which custodies their use for an anonymous user. At some later point these users may choose to link the custodied account to their self-custodial wallet taking full ownership of the account. EVM apps on Flow can also leverage Account Linking to handle creation of accounts and achieve a similarly smooth onboarding user experience.

With Flow, builders can choose to expand EVM capabilities and transcend limitations using Cadence, which offers a powerful new account model, programmable resources, and hybrid ownership.

Instant Cross-VM Token Transfers

EVM and Cadence environments both use FLOW as gas for transactions, sharing a singular token supply across both environments. Fungible and non-fungible tokens can also be seamlessly transferred between environments using the native VM token bridge, taking place instantly in a single atomic transaction.

Scalability, Performance and Low Gas Fees

For sustainable user adoption, apps require the network they build on to be secure, efficient, affordable and fast. Gas fees are ultra-low cost on the network, but Flow goes a step further allowing for gasless experiences through sponsored transactions. Scalable performance is ensured with an innovative multi-node distributed consensus, flexible transaction model and horizontally scaled transaction linearization which solves proposer-builder separation, separation of compute, and settlement - all without sharding.

Flow's state space is extensible to the petabyte scale making it easy to store application data on-chain. This means contracts can maintain a full working dataset - including metadata - together with contract logic.

Flow's transaction throughput peaked to 2M daily transactions during 2023 sustaining a similar average transaction volume as Ethereum. Unlike Ethereum, Flow has always operated well under its maximum throughput ceiling which is presently scalable to 5x more transactions with further performance optimizations to come when parallel execution is released. State scalability on Flow sets the foundations for further significant throughput optimization.

MEV Resilience

The MEV Resilient design on Flow offers DeFi builders improved market efficiency, fairness, trust and long-term viability for their apps. Since EVM on Flow transactions are composed and executed within a Cadence transaction, block production is handled by Flow's multi-role architecture. This heterogeneity between node roles ensures that visibility into block proposal, assembly, asserting block validity and other correctness checks during the block production cycle exposes limited information to each node type on a need to know basis, observing the Principle of Least Privilege. These differences in node and consensus design results in strong economic disincentives for collusion because no individual node has full visibility into the state of block production for the chain. This robust MEV resilience is a significant difference from other EVM-compatible networks and results in reasonably priced, predictable gas fees. The impracticality of frontrunning or other attacks improves the user experience by eliminating failed transactions and invisible fees.

Join the Community

Are you interested in launching an EVM project on Flow or partnering with us? Visit our weekly Flow office hours for discussions on project development and other opportunities for collaboration. You can also chat with us developers-chat in the Flow Discord.

Further Reading and Discussions

- Why EVM on Flow: Beyond Solidity

- Path to EVM Equivalence on Flow

Flow Improvement Proposals (FLIPs)

Those wishing to understand the technical specifics of how EVM on Flow works we recommend reviewing the following improvement proposals.

- Understanding EVM Support on Flow
- Exploring the Flow VM Bridge
- Insights into the Flow EVM Gateway
- Integration of the Cadence Interface

accounts.md:

```
---
```

title: Accounts
sidebarlabel: Accounts
sidebarposition: 6

:::info

Are you a Cadence developer looking for information about Accounts on Cadence? If so, check out the Cadence specific documentation here

:::

Accounts

There are three types of accounts used for EVM on Flow.

1. Externally Owned Accounts (EOA): EOAs are controlled by private individuals using cryptographic keys and can initiate transactions directly. They are the primary account type for users to interact with the blockchain, holding and sending cryptocurrency or calling smart contract functions.
2. Contract Accounts: These accounts hold smart contract code and are governed by this code's logic. Unlike EOAs, Contract Accounts do not initiate transactions on their own but can execute transactions in response to calls they receive from EOAs or other contracts.
3. Cadence Owned Accounts (COA): This is an account type unique to Flow EVM. These accounts are managed by Cadence resources and can be used to interact with the Flow EVM from within the Cadence environment.

EOAs and Contract accounts function the same as on other EVM networks. Users may interact with these accounts using the standard EVM JSON-RPC API (see endpoints [here](#)). You can read more about EOAs and Contract accounts on the Ethereum docs.

However, in order to leverage all the features of Cadence, developers will need to utilize Cadence Owned Accounts.

Cadence Owned Accounts

A Cadence Owned Account (COA) is a natively supported EVM smart contract wallet type that allows a Cadence resource to own and control an EVM address. This native wallet type provides the primitives needed to bridge or control assets across Flow EVM and Cadence facilitating composability between environments.

!Account-Model

Why use COAs?

COAs create powerful new opportunities to improve the UX, functionality and utility of EVM applications by taking advantage of Cadence. Key benefits include:

- Enhanced Composability: Applications written in Solidity can be extended and composed upon within Cadence. This allows developers to build upon existing EVM applications and deliver a more feature-rich user experience.
- Atomic Interactions: Developers are able to execute multiple EVM transactions atomically from a COA. This is particularly useful for applications that require multiple transactions to be executed within a single block, or require all prior transactions' state changes to revert if a single transaction in the batch fails. This is not possible natively using EOAs or with UserOperations when using the ERC-4337 standard; in both cases each individual transaction is distinct and cannot be reverted back once state has changed.
- Native Account Abstraction: COAs are controlled by Cadence resources, which are in turn owned by Flow accounts. Flow accounts have built-in support for multi-signature authentication, key rotation, and account recovery. As a Cadence resource, COAs naturally inherit these features.
- Fine-Grained Access Control: As Cadence resources, access to a COA can be governed by more sophisticated policies than those available with basic EVM accounts. By utilizing powerful Cadence access control primitives such as capabilities and entitlements, developers can restrict who is able to interact with a COA and what actions they are permitted to perform.

Differences from Traditional EVM Accounts

COAs are smart contracts that are deployed to, and are fully accessible within, Flow EVM. However, unlike traditional EVM accounts (e.g. EOAs or smart contract accounts), COAs are owned by a Cadence resource. This means that COAs can be created and controlled natively within the Cadence execution environment.

Unlike EOAs, COAs do not have an associated key, but are assigned a 20-byte EVM address upon creation from Cadence. This address is based on the UUID of the Cadence resource and is prefixed with 0x00000000000000000000000000000002. This address determines the location of the

COA smart contract deployment and is the EVM address that is used to interact with the COA.

A COA may instantiate transactions itself (where the COA's EVM address acts as tx.origin). This behaviour differs from other EVM environments, where only externally owned accounts (EOAs) may instantiate transactions.

Because COAs are owned by Cadence resources, an EVM transaction is not required to trigger a transaction from a COA (e.g. a transaction to make a call to execute or EIP-4337's validateUserOpMethod). Instead, call transactions may be triggered directly from the Cadence resource that owns the COA. By invoking the call method on this resource, a transaction event will be emitted within the EVM environment.

More Information

To learn how to create and interact with COAs in Cadence, see the guide for Interacting with COAs from Cadence.

For more information about Cadence Owned Accounts, see the Flow EVM Support FLIP

```
# data-indexers.md:
```

```
---
```

```
title: Data Indexers
```

```
sidebarlabel: Data Indexers
```

```
sidebarposition: 9
```

```
---
```

Data Indexers

When building applications that leverage Flow data, developers have multiple Data Indexers to choose from. These platforms offer flexible options, allowing you to index all data on Flow, including information from both the Cadence VM and EVM. Alternatively, if your application doesn't require Cadence, you can opt to index only EVM data. This flexibility ensures that you can tailor your data indexing strategy to fit the specific needs of your application.

EVM & Cadence

SimpleHash

SimpleHash is a comprehensive multi-chain NFT data platform that provides developers with easy access to token data across 60+ blockchain networks. It offers a robust API for querying NFT metadata, media, collection details, sales, floor prices, listings, and bids, streamlining the process of building NFT-powered applications.

Getting Started with SimpleHash

Developers can begin using SimpleHash by signing up for an API key on their website. The platform provides comprehensive documentation to help developers integrate SimpleHash into their projects.

EVM Only

Moralis

Moralis provides a robust suite of data APIs designed to support a wide array of blockchain applications. These APIs deliver both indexed and real-time data across 16+ blockchain networks, including comprehensive details on portfolio and wallet balances, NFT data, token metrics, price feeds, candlestick charts, and net worth calculations. Moralis enhances this data with additional layers of metadata, parsed events, and address labels to provide deeper insights and context.

Getting Started with Moralis

To integrate Moralis into your project, begin by creating an account. Detailed API references and integration guides are available in the Moralis documentation. For step-by-step tutorials and use cases, visit their YouTube channel.

Alchemy

Alchemy is a powerful blockchain development platform that provides enhanced APIs and advanced analytics to streamline the process of creating and scaling Web3 applications.

Getting Started with Alchemy

To begin using Alchemy, developers can sign up for an account on the Alchemy website. The platform offers extensive documentation including API references, tutorials, and guides to help developers integrate Alchemy into their projects.

```
# fees.md:
```

```
---
title: Fees
sidebarlabel: Fees
sidebarposition: 5
---
```

```
:::info
```

Are you a Cadence developer looking for information about Fees on Cadence? If so, check out the Cadence specific documentation here

```
:::
```

EVM transactions are ultra low-cost and use the native FLOW token as gas. Externally Owned Accounts (EOAs) function the same on Flow as other EVM networks like Ethereum.

```
<details>
<summary><h2>How Transaction Fees are Computed on EVM</h2></summary>
```

With EVM on Flow, EVM operations can now be called within Cadence transactions. EVM operations also have an associated effort measured in gas which needs to be factored into the execution effort calculation in addition to the Flow computation for any EVM transaction.

Transaction fee on EVM = surge x [inclusion fee + (execution effort unit cost)]

- Surge' factor dynamically accounts for network pressure and market conditions. This is currently constant at 1.0 but subject to change with community approval.
- Inclusion fee accounts for the resources required to process a transaction due to its core properties (byte size, signatures). This is currently constant at 1E-6 FLOW, but subject to change with community approval.
- Execution fee The fee that accounts for the operational cost of running the transaction script, processing the results, sending results for verification, generating verification receipts, etc. and is calculated as a product of execution effort units and the cost per unit.
- Execution Effort (computation) is based on transaction type and operations that are called during the execution of a transaction. The weights determine how "costly" (time consuming) each operation is.
- Execution Effort Unit Cost = 2.49E-07 FLOW (currently constant, but subject to change with community approval)

<h3>Calculation of Execution Effort</h3>

```
Execution Effort (computation) =
  0.00478  functionorloopcall +
  0.00246  GetValue +
  0.00234  SetValue +
  8.65988  CreateAccount +
  EVMGasUsageCost  EVMGasUsage
```

where

EVMGasUsage is reported by EVM as the cost in gas for executing the transaction within the EVM, for instance, 21K gas for a simple send transaction.

EVMGasUsageCost - The ratio that converts EVM gas into Flow computation units (execution effort) is currently set at 1/5000 but subject to revision by community approval

Note: The weights and unit cost mentioned above have been updated recently to accommodate an increased computation limit on Flow, which now supports the deployment of larger EVM contracts. For detailed information, refer to the relevant FLIP and join the ongoing discussion on the community forum post. These values may be adjusted in the future based on community feedback and evolving requirements.

</details>

<details>

<summary><h2>Demonstration of Transaction Fees on EVM</h2></summary>

Assume a simple NFT transfer transaction that makes 31 cadence loop calls, reads 5668 bytes from the storage register, and saves 1668 bytes to the storage register.

- 'functionorloopcall' = 31
- 'GetValue' = 5668
- 'SetValue' = 1668
- 'CreateAccount' = 0

Scenario 1 - Cadence-only Transaction

Execution Effort = 0.00478 (31) + 0.00246 (5668) + 0.00234 (1668) + 8.65988 (0) + EVMGasUsageCost EVMGasUsage

But since EVMGasUsage is 0 for a Cadence transaction,

Execution Effort = 18.04378

Thus

Transaction fee = [1E-6 FLOW + (18.04378 2.49E-07 FLOW)] x 1 = 5.5E-06 FLOW

Scenario 2 - EVM Transaction

If the EVMGasUsage can be assumed to be 21,000 gas (typical for a simple transfer),

Execution Effort = 0.00478 (31) + 0.00246 (5668) + 0.00234 (1668) + 8.65988 (0) + 1/5000 21000 = 22.24378

Thus

Transaction fee = [1E-6 FLOW + (110.97 2.49E-07 FLOW)] x 1 = 6.55E-06 FLOW

Note: Please be aware that this example serves solely for illustrative purposes to elucidate the calculations. Actual transaction fees may differ due to various factors, including the byte size of the transaction.

</details>

Gasless Transactions

Fees needed to execute transactions on a Web3 app are often a major challenge for new users and can be a barrier to adoption. Builders can easily extend their apps with Cadence to create 'gasless' experiences by specifying their app as the sponsor instead of the user.

To learn more about storage fee and transaction fee, visit [Flow Tokenomics](#) page.

```
# how-it-works.md:
```

```
---
```

```
title: How EVM on Flow Works
sidebarlabel: How it Works
sidebarposition: 2
---
```

How EVM on Flow Works

Introduction

The Flow network uses Cadence as its main execution environment. Cadence offers a safe, efficient, and developer-friendly experience for building smart contracts and decentralized applications. Cadence can be used to extend EVM apps built in Solidity by unlocking gasless experiences, new business models and fine-tuned access control. With Flow offering full EVM support, existing applications and tools already deployed in the EVM ecosystem can simply onboard to the network with no code changes.

EVM on Flow is designed with these major goals in mind:

- Supporting EVM equivalency: Ensure that any tools and applications deployed to or run on Ethereum can also be deployed and run on Flow.
- Minimizing breaking changes to the Cadence ecosystem, software and tools
- Maximum composability across environments: Allowing atomic and smooth interaction between EVM and Cadence environments.

EVM - A Smart Contract In Cadence

To satisfy the design goals and thanks to the extensibility properties of the Cadence runtime, EVM on Flow is designed as a higher-level environment incorporated as a smart contract deployed to Cadence. This smart contract is not owned by anyone and has its own storage space, allows Cadence to query, and is updated through EVM transactions. EVM transactions can be wrapped inside Cadence transactions and passed to the EVM contract for execution. The artifacts of EVM transaction execution (e.g. receipts and logs) are emitted as special Cadence events (`TransactionExecuted`, `BlockExecuted`) and available to the upstream process (Flow transaction) to enable atomic operations.

The EVM environment has its own concept of blocks, and every Flow block includes at most one EVM Block. The EVM block is formed at the end of

Flow Block execution and includes all the transaction executed during the EVM block execution. Note that since EVM blocks are formed on-chain and Flow provides fast finality, as long as the user of these events waits for Flow block finality, it doesn't have to worry about EVM block forks, uncle chains, and other consensus-related challenges.

No Shared Memory Design

The interaction between two environments is through atomic calls and none of the environments has access to the raw memory of the other. This maintains the security properties of each environment. Cadence can submit transactions to the EVM environment and EVM transactions can make calls to a special precompiled contract called Cadence Arch. You can read more about this in the Precompiled section.

No New Native Token

EVM on Flow uses the same native token as Cadence (FLOW token). No new token is minted at the genesis block of EVM and all the tokens have to be bridged over from the Cadence side into the EVM side. To facilitate this a native bridge is provided by the EVM contract.

EVM Equivalency

Under the hood, EVM on Flow uses the standard EVM implementation and regularly applies updates through Flow's height-coordinated updates (e.g. Execution layer changes planned for the Ethereum Prague update). This means anything that can run on Ethereum after the Cancun upgrade can run on Flow EVM. This means many useful EIPs such as EIP-1014, EIP-1559, EIP-4844, EIP-5656, EIP-6780, ... are supported automatically.

Yet a small set of differences between EVM on Flow and Ethereum might be seen (mostly of the nature of extension) for two reasons:

- A set of extensions has been added to ensure seamless and easy interaction between the two environments.
- EVM on Flow is secured by the Flow network and benefits from its robust network properties, such as fast block production and finalization, making issues like handling uncle chains irrelevant.

Gateways

As mentioned, EVM on Flow runs on top of the Flow network and its consensus model. EVM on Flow does not leverage geth or introduce new node types to the existing architecture. Operators wishing to participate in securing the network stake tokens and run one of the Flow node types.

To support web3.js clients, the EVM Gateway honors the Ethereum JSON-RPC specification. The gateway integrates with Flow access nodes and can be run by anyone (unstaked). It serves two purposes:

Gateway As A Light Client

The gateway follows Flow's block production, collecting, verifying and indexing EVM-related events. The gateway provides the necessary endpoints and services all JSON-RPC requests for third-party dapps interacting with Flow EVM. EVM events include all information needed to reconstruct the EVM state since the genesis block (replayability). By re-executing transactions, the gateway can collect traces and maintain a local and archival copy of the EVM state over time.

Gateway As a Sequencer

As mentioned, EVM on Flow can be seen as a higher-level environment built on top of Cadence. Thus, all EVM transactions are ultimately handled using a Flow transaction (a wrapped call to the EVM). The gateway accepts EVM transactions, runs an internal mempool of transactions, wraps batches of EVM transactions in Flow transactions, and submits them.

Note that the safety of transaction execution is not dependent on the gateway; they only relay the transaction. The safety measures of the EVM environment (e.g., Nonce, etc.) ensure that each transaction is executed at most once. Since gateways are submitting Flow transactions, they have to pay the related transaction fees. Part of these fees is associated with the computation fees of the EVM transaction.

To facilitate the repayment of fees, the `evm.run` function accepts a coinbase address, which collects gas fees from the transaction, and pays it to the address provided by the gateway node. Essentially, the transaction wrapper behaves similarly to a miner, receives the gas usage fees on an EVM address, and pays for the transaction fees. The gas price per unit of gas creates a marketplace for these 3rd parties to compete over transactions.

Censorship Resistance and MEV Protection

Since EVM on Flow runs on the Flow network, it benefits from Flow's protections against censorship and MEV attacks. The Flow network natively provides censorship & MEV resistance which is achieved by designating specific validators for building transaction bundles that are separated from the validators proposing blocks (proposer-builder separation). More details about this are available in Flow's protocol white papers. For extra protection on the EVM side, the gateway software is designed to be fully configurable and as lightweight as possible. This enables anyone with an account on Flow (e.g., any application) to run their own instances.

Fee Market Change (EIP-1559)

EIP-1559 is supported by the EVM on Flow and Gateway nodes can decide on the inclusion of the transactions based on the tips or gas fees. The parameters for the EIP 1559 are adjustable by the Flow network. Currently, the base fee is set to zero, as EVM transactions are wrapped by the Flow transactions.

Opcodes

EVM on Flow supports opcodes listed here, except for the following changes.

- COINBASE (block.coinbase)

Similar to Ethereum it returns the address of block's beneficiary address. In the case of EVM on Flow, it returns the address of the current sequencer's fee wallet (see Gateway section for more details).

- PREVRANDAO (block.prevrandao)

On Ethereum, this value provides access to beacon chain randomness (see EIP-4399), Since Flow uses a different approach in consensus and verifiable randomness generation, this value is filled with a random number provided by the Flow protocol. While EVM on Flow provides such opcode, it is recommended not to rely on this value for security-sensitive applications, as it is the case on Ethereum. In order to benefit from the full power of secure randomness on Flow, it's recommended to use the Cadence Arch precompiles.

Precompiled Contracts

Besides all the precompiled contracts supported by Ethereum (see here for the list), EVM on Flow has augmented this with a unique precompiled contract, the Cadence Arch, that provides access to the Cadence world.

Cadence Arch is a multi-function smart contract (deployed at 0x0000000000000000000000000000000100000000000000000000000000000001) that allows any smart contract on Flow EVM a limited set of interactions with the Cadence environment.

Functions currently available on the Cadence Arch smart contract are:

- FlowBlockHeight() uint64 (signature: 0x53e87d66) returns the current Flow block height, this could be used instead of Flow EVM block heights to trigger scheduled actions given it's more predictable when a block might be formed.
- VerifyCOAOwnershipProof(bytes32 hash, bytes memory signature) (bool success) returns true if the proof is valid. An ownership proof verifies that a Flow wallet controls a COA account (see the next section for more details on COA).
- revertibleRandom() uint64 returns a safe pseudo-random value that is produced by the Flow VRF (using Flow internal randomness beacon). The function invokes Cadence's revertibleRandom<uint64> described here. Although the random value is safe, a transaction may revert its results in the case of an unfavourable outcome. The function should only be used by trusted calls where there is no issue with reverting the results. getRandomSource must be used instead with untrusted calls.
- getRandomSource(uint64) bytes32 should be used when implementing a commit-reveal scheme. It returns a secure random source from the Cadence randomness history contract. Learn more about the secure usage of randomness on Flow here.

Here is a sample demonstrating how to call the Cadence Arch.

```
solidity
    address constant public cadenceArch =
0x0000000000000000000000000000000100000000000000000000000000000001;

    function flowBlockHeight() public view returns (uint64) {
        (bool ok, bytes memory data) =
cadenceArch.staticcall(abi.encodeWithSignature("flowBlockHeight()"));
        require(ok, "failed to fetch the flow block height through
cadence arch");
        uint64 output = abi.decode(data, (uint64));
        return output;
    }
}
```

Special Addresses

Native Token Bridge

Both Cadence and EVM on Flow use the same token (FLOW) to run their operations. No new token is minted on the EVM side. Moving FLOW tokens easily across two environments has been supported natively by the EVM smart contract. Because the EVM have limited visibility into Cadence and to make tracking funds easier, every time Flow tokens are withdrawn from the Cadence side and deposited into an EVM address, the balance would be added to a special address 0x00000000000000000000000000000000100000000000000000000 (native token bridge) and then transferred to the destination EVM address. The bridge address always maintains a balance of zero. Clearly, this EOA address is a network address and is not controlled by public key.

Cadence-Owned Accounts (COAs)

COA is a natively supported EVM smart contract wallet type that allows a Cadence resource to own and control an EVM address. This native wallet provides the primitives needed to bridge or control assets across Flow EVM and Cadence. From the EVM perspective, COAs are smart contract wallets that accept native token transfers and support several ERCs including ERC-165, ERC-721, ERC-777, ERC-1155, ERC-1271.

These smart contract wallets are only deployable through the Cadence environment and their address starts with the prefix 0x00000000000000000000000000000002. The address 0x0000000000000000000000000000000020000000000000000000000000000000 is reserved for COA factory, an address that deploys contracts for COA accounts.

A COA is not controlled by a key. Instead, every COA account has a unique resource accessible on the Cadence side, and anyone who owns that resource submits transactions on behalf of this address. These direct transactions have COA's EVM address as the tx.origin and a new EVM transaction type (TxType = 0xff) is used to differentiate these transactions from other types of EVM transactions (e.g., DynamicFeeTxType (0x02)). Currently, to make integration and tracking of these transactions byte EVM ecosystem tools, these types of transactions are encoded as

legacy EVM transactions (hash computation is based on legacy tx rlp encoding).

Controlling through a resource makes a COA a powerful smart contract wallet. It makes the transfer of ownership of the EVM address super easy without the need to transfer all the assets that an EVM address owns. It also allows a Cadence smart contract to take ownership of an EVM address and makes fully decentralized exchange and bridges across environments possible.

To learn more about how to interact with a COA from the Cadence side, see [here](#).

Proofs

Inclusion proof of execution artifacts (logs and receipts)

Similar to other EVM environments, proof can be constructed for artifacts such as receipts. As mentioned earlier, all the EVM execution artifacts are collected as part of a Cadence event. Cadence events are similar to EVM logs, and the root hash of all events (event commitment) of a block is included in Flow's block content. The Cadence event inclusion proof functionality enables constructing proofs for any artifact. For example, if one wants to construct an external proof for the inclusion of a specific EVM log or receipt, here are the steps:

- Flow block validation: Anyone following the Flow blocks can validate the Flow block headers.
- EVM block validation: Since each Flow block has the root hash of Cadence events emitted during block execution. It can construct and verify the inclusion of the specific Event. In this case, every time an EVM block is executed an `evm.BlockExecuted` event is emitted that contains the full EVM block information.
- Receipt inclusion: Each EVM block includes the root hash for the receipts generated during block execution. Similar to other EVM chains, a proof can be constructed to prove inclusion of a log or receipt.

Inclusion proof of transactions

Each Flow EVM block (`TransactionHashRoot`) includes the Merkle root hash of all the transaction hashes executed during this block. Despite similar functionality, this root is a bit different than `TransactionRoot` provided by Ethereum, It is a commitment over the list of transaction hashes instead of transactions, so each leaf node in the Merkle tree has the transaction hash as the value instead of full RLP encoding of transaction. So verifying the proofs requires an extra calling to the hash function.

Account proofs

Another type of proof that EVM environments provide is proof for the state of accounts. These proofs depend on the trie structure of the

execution environment. EVM on Flow benefits from the advanced storage and proof system that makes Flow's multi-role architecture possible.

Flow's state system provides ways to construct inclusion and non-inclusion proofs and one can construct proofs for EVM account's meta data (account balances, nonce, ...). A less common proof type is proof over the storage state of an account (mostly used for smart contracts). The first release of EVM on Flow won't support these type of proofs.

```
# networks.md:
```

```
---
```

```
title: Networks
```

```
sidebarlabel: Networks
```

```
sidebarposition: 4
```

```
---
```

Networks

EVM on Flow has the following public RPC nodes available:

Mainnet

Name	Value
Network Name	EVM on Flow
Description	The public RPC URL for Flow Mainnet
RPC Endpoint	https://mainnet.evm.nodes.onflow.org
Chain ID	747
Currency Symbol	FLOW
Block Explorer	https://evm.flowscan.io

Testnet

Name	Value
Network Name	EVM on Flow Testnet
Description	The public RPC URL for Flow Testnet
RPC Endpoint	https://testnet.evm.nodes.onflow.org
Chain ID	545
Currency Symbol	FLOW
Block Explorer	https://evm-testnet.flowscan.io

```
# direct-calls.md:
```

```
---
```

```
title: Direct Calls from Cadence to Flow EVM
```

```
sidebarlabel: Direct Calls to Flow EVM
```

```
sidebarposition: 5
```

```
---
```

Direct calls from Cadence to Flow EVM are essential for enabling Cadence smart contracts to interact seamlessly with the EVM environment hosted on the Flow blockchain. These calls facilitate a range of functionalities including state queries and transaction initiations, allowing Cadence contracts to leverage EVM-based tools and assets.

Making Direct Calls

Accessing Flow EVM

To interact with Flow EVM, Cadence contracts must first import EVM from its service address:

```
js
import EVM from <ServiceAddress>
```

Next, create an EVMAddress with a sequence of 20 bytes representing the EVM address:

```
js
let addr = EVM.EVMAddress(bytes: bytes)
```

Once you have access to an EVMAddress, you can query various pieces of state information such as:

- balance() EVM.Balance provides the balance of the address. It returns a balance object rather than a basic type to avoid errors when converting from flow to atto-flow.
- nonce() UInt64 retrieves the nonce associated with the address.
- code(): [UInt8] fetches the code at the address; it returns the smart contract code if applicable, and is empty otherwise.

```
cadence
import EVM from <ServiceAddress>

access(all)
fun main(bytes: [UInt8; 20]): EVM.Balance {
    let addr = EVM.EVMAddress(bytes: bytes)
    let bal = addr.balance()
    return bal
}
```

Alternatively, you can use the EVM contract's native deserialization to access the balance provided a hex string representing the address:

```
cadence
import EVM from <ServiceAddress>

access(all)
fun main(addressHex: String): UFix64 {
    let addr = EVM.addressFromString(addressHex)
```

```
        return addr.balance().inFLOW()
    }
```

Sending Transactions to Flow EVM

To send transactions to Flow EVM, use the run function which executes RLP-encoded transactions. RLP (Recursive Length Prefix) encoding is used to efficiently encode data into a byte-array format, suitable for Ethereum-based environments. Here's an example of wrapping and sending a transaction:

```
cadence
import EVM from <ServiceAddress>

transaction(rlpEncodedTransaction: [UInt8], coinbaseBytes: [UInt8; 20]) {

    prepare(signer: &Account) {
        let coinbase = EVM.EVMAddress(bytes: coinbaseBytes)
        let result = EVM.run(tx: rlpEncodedTransaction, coinbase:
coinbase)
        assert(
            runResult.status == EVM.Status.successful,
            message: "tx was not executed successfully."
        )
    }
}
```

Using run restricts an EVM block to a single EVM transaction, while a future batchRun will offer the capability to execute multiple EVM transactions in a batch.

Handling Transaction Responses

Handling responses correctly is crucial to manage the state changes or errors that occur during EVM transactions:

When calling EVM.run, it's important to understand that this method does not revert the outer Flow transaction. Developers must therefore carefully handle the response based on the result.Status of the EVM transaction execution. There are three main outcomes to consider:

- Status.invalid: This status indicates that the transaction or call failed at the validation step, such as due to a nonce mismatch. Transactions with this status are not executed or included in a block, meaning no state change occurs.
- Status.failed: This status is assigned when the transaction has technically succeeded in terms of being processable, but the EVM reports an error as the outcome, such as running out of gas. Importantly, a failed transaction or call is still included in a block. Attempting to resubmit a failed transaction will result in an invalid status on the second try due to a now incorrect nonce.

- `Status.successful`: This status is given when the transaction or call is successfully executed and no errors are reported by the EVM.

For scenarios where transaction validity is critical, developers may choose to use the `mustRun` variation, which reverts the transaction in the case of a validation failure, providing an added layer of error handling.

Understanding Gas Usage in EVM Transactions

Direct calls to Flow EVM require gas, it's important to understand how gas usage is calculated and billed. During the execution of methods that interact with the EVM:

- **Gas Aggregation**: The gas used by each call is aggregated throughout the transaction.
- **Gas Adjustment**: The total gas used is then adjusted based on a multiplier. This multiplier is determined by the network and can be adjusted by the service account to reflect operational costs and network conditions.
- **Payment of Gas Fees**: The adjusted total gas amount is added to the overall computation fees of the Flow transaction. These fees are paid by the transaction initiator, commonly referred to as the payer.

Keep Learning

For more information and a deeper dive into the `EVMAddress`, `Result`, and `Status` objects, see the contract [here](#).

interacting-with-coa.md:

```
---
```

title: Interacting with COAs from Cadence
sidebarlabel: Interacting with COAs
sidebarposition: 4

Cadence Owned Accounts (COAs) are EVM accounts owned by a Cadence resource and are used to interact with Flow EVM from Cadence.

COAs expose two interfaces for interaction: one on the Cadence side and one on the EVM side. In this guide, we will focus on how to interact with COAs with Cadence.

In this guide we will walk through some basic examples creating and interacting with a COA in Cadence. Your specific usage of the COA resource will depend on your own application's requirements (e.g. the COA resource may not live directly in `/storage/evm` as in these examples, but may instead be a part of a more complex resource structure).

COA Interface

To begin, we can take a look at a simplified version of the EVM contract, highlighting parts specific to COAs.

You can learn more about the EVM contract here and the full contract code can be found on GitHub.

```
cadence
access(all)
contract EVM {
    //...
    access(all)
    resource CadenceOwnedAccount: Addressable {
        /// The EVM address of the cadence owned account
        /// -> could be used to query balance, code, nonce, etc.
        access(all)
        view fun address(): EVM.EVMAddress

        /// Get balance of the cadence owned account
        /// This balance
        access(all)
        view fun balance(): EVM.Balance

        /// Deposits the given vault into the cadence owned account's
balance
        access(all)
        fun deposit(from: @FlowToken.Vault)

        /// The EVM address of the cadence owned account behind an
entitlement, acting as proof of access
        access(EVM.Owner | EVM.Validate)
        view fun protectedAddress(): EVM.EVMAddress

        /// Withdraws the balance from the cadence owned account's
balance
        /// Note that amounts smaller than 10nF (10e-8) can't be
withdrawn
        /// given that Flow Token Vaults use UFix64s to store balances.
        /// If the given balance conversion to UFix64 results in
        /// rounding error, this function would fail.
        access(EVM.Owner | EVM.Withdraw)
        fun withdraw(balance: EVM.Balance): @FlowToken.Vault

        /// Deploys a contract to the EVM environment.
        /// Returns the address of the newly deployed contract
        access(EVM.Owner | EVM.Deploy)
        fun deploy(
            code: [UInt8],
            gasLimit: UInt64,
            value: Balance
        ): EVM.EVMAddress

        /// Calls a function with the given data.
        /// The execution is limited by the given amount of gas
        access(EVM.Owner | EVM.Call)
        fun call(
            to: EVMAddress,
```

```

        data: [UInt8],
        gasLimit: UInt64,
        value: Balance
    ): EVM.Result
}

// Create a new CadenceOwnedAccount resource
access(all)
fun createCadenceOwnedAccount(): @EVM.CadenceOwnedAccount
// ...
}

```

Importing the EVM Contract

The CadenceOwnedAccount resource is a part of the EVM system contract, so to use any of these functions, you will need to begin by importing the EVM contract into your Cadence code.

To import the EVM contract into your Cadence code using the simple import syntax, you can use the following format (learn more about configuring contracts in `flow.json` here):

```

cadence
// This assumes you are working in the Flow CLI, FCL, or another
// tool that supports this syntax
// The contract address should be configured in your project's flow.json
file
import "EVM"
// ...

```

However, if you wish to use manual address imports instead, you can use the following format:

```

cadence
// Must use the correct address based on the network you are interacting
with
import EVM from 0x1234
// ...

```

To find the deployment addresses of the EVM contract, you can refer to the EVM contract documentation.

Creating a COA

To create a COA, we can use the `createCadenceOwnedAccount` function from the EVM contract. This function takes no arguments and returns a new CadenceOwnedAccount resource which represents this newly created EVM account.

For example, we can create this COA in a transaction, saving it to the user's storage and publishing a public capability to its reference:

```

cadence
import "EVM"

// Note that this is a simplified example & will not handle cases where
the COA already exists
transaction() {
    prepare(signer: auth(SaveValue, IssueStorageCapabilityController,
PublishCapability) &Account) {
        let storagePath = /storage/evm
        let publicPath = /public/evm

        // Create account & save to storage
        let coa: @EVM.CadenceOwnedAccount <-
EVM.createCadenceOwnedAccount()
        signer.storage.save(<-coa, to: storagePath)

        // Publish a public capability to the COA
        let cap =
signer.capabilities.storage.issue<&EVM.CadenceOwnedAccount>(storagePath)
        signer.capabilities.publish(cap, at: publicPath)
    }
}

```

Getting the EVM Address of a COA

To get the EVM address of a COA, you can use the address function from the EVM contract. This function returns the EVM address of the COA as an EVM.Address struct. This struct is used to represent addresses within Flow EVM and can also be used to query the balance, code, nonce, etc. of an account.

For our example, we could query the address of the COA we just created with the following script:

```

cadence
import "EVM"

access(all)
fun main(address: Address): EVM.EVMAddress {
    // Get the desired Flow account holding the COA in storage
    let account = getAuthAccount<auth(Storage) &Account>(address)

    // Borrow a reference to the COA from the storage location we saved
    it to
    let coa = account.storage.borrow<&EVM.CadenceOwnedAccount>(
        from: /storage/evm
    ) ?? panic("Could not borrow reference to the COA")

    // Return the EVM address of the COA
    return coa.address()
}

```

If you'd prefer the hex representation of the address, you instead return using the `EVMAddress.toString()` function:

```
cadence
return coa.address().toString()
```

The above will return the EVM address as a string; however note that Cadence does not prefix hex strings with `0x`.

Getting the Flow Balance of a COA

Like any other Flow EVM or Cadence account, COAs possess a balance of FLOW tokens. To get the current balance of our COA, we can use the COA's balance function. It will return a `EVM.Balance` struct for the account - these are used to represent balances within Flow EVM.

This script will query the current balance of our newly created COA:

```
cadence
import "EVM"

access(all)
fun main(address: Address): EVM.Balance {
    // Get the desired Flow account holding the COA in storage
    let account = getAuthAccount<auth(Storage) &Account>(address)

    // Borrow a reference to the COA from the storage location we saved
    it to
        let coa = account.storage.borrow<&EVM.CadenceOwnedAccount>(
            from: /storage/evm
        ) ?? panic("Could not borrow reference to the COA")

    // Get the current balance of this COA
    return coa.balance()
}
```

You can also easily get the `UFix64` FLOW balance of any EVM address with this script:

```
cadence
import "EVM"

access(all)
fun main(addressHex: String): UFix64 {
    let addr = EVM.addressFromString(addressHex)
    return addr.balance().inFLOW()
}
```

The above script is helpful if you already know the COA address and can provide the hex representation directly.

Depositing and Withdrawing Flow Tokens

Tokens can be seamlessly transferred between the Flow EVM and Cadence environment using the deposit and withdraw functions provided by the COA resource. Anybody with a valid reference to a COA may deposit Flow tokens into it, however only someone with the Owner or Withdraw entitlements can withdraw tokens.

Depositing Flow Tokens

The deposit function takes a FlowToken.Vault resource as an argument, representing the tokens to deposit. It will transfer the tokens from the vault into the COA's balance.

This transaction will withdraw Flow tokens from a user's Cadence vault and deposit them into their COA:

```
cadence
import "EVM"
import "FungibleToken"
import "FlowToken"

transaction(amount: UFix64) {
    let coa: &EVM.CadenceOwnedAccount
    let sentVault: @FlowToken.Vault

    prepare(signer: auth(BorrowValue) &Account) {
        // Borrow the public capability to the COA from the desired
        account
        // This script could be modified to deposit into any account with
        a EVM.CadenceOwnedAccount capability
        self.coa =
        signer.capabilities.borrow<&EVM.CadenceOwnedAccount>(/public/evm)
        ?? panic("Could not borrow reference to the COA")

        // Withdraw the balance from the COA, we will use this later to
        deposit into the receiving account
        let vaultRef = signer.storage.borrow<auth(FungibleToken.Withdraw)
        &FlowToken.Vault>(
            from: /storage/flowTokenVault
            ) ?? panic("Could not borrow reference to the owner's Vault")
        self.sentVault <- vaultRef.withdraw(amount: amount) as!
        @FlowToken.Vault
    }

    execute {
        // Deposit the withdrawn tokens into the COA
        self.coa.deposit(from: <-self.sentVault)
    }
}
```

```
:::info
This is a basic example which only transfers tokens between a single
user's COA & Flow account. It can be easily modified to transfer these
tokens between any arbitrary accounts.
```

You can also deposit tokens directly into other types of EVM accounts using the EVM.EVMAddress.deposit function. See the EVM contract documentation for more information.

```
:::
```

Withdrawing Flow Tokens

The withdraw function takes a EVM.Balance struct as an argument, representing the amount of Flow tokens to withdraw, and returns a FlowToken.Vault resource with the withdrawn tokens.

We can run the following transaction to withdraw Flow tokens from a user's COA and deposit them into their Flow vault:

```
cadence
import "EVM"
import "FungibleToken"
import "FlowToken"

transaction(amount: UFix64) {
    let sentVault: @FlowToken.Vault
    let receiver: &{FungibleToken.Receiver}

    prepare(signer: auth(BorrowValue) &Account) {
        // Borrow a reference to the COA from the storage location we
        saved it to with the EVM.Withdraw entitlement
        let coa = signer.storage.borrow<auth(EVM.Withdraw)
&EVM.CadenceOwnedAccount>(
            from: /storage/evm
        ) ?? panic("Could not borrow reference to the COA")

        // We must create a EVM.Balance struct to represent the amount of
        Flow tokens to withdraw
        let withdrawBalance = EVM.Balance(attoflow: 0)
        withdrawBalance.setFLOW(flow: amount)

        // Withdraw the balance from the COA, we will use this later to
        deposit into the receiving account
        self.sentVault <- coa.withdraw(balance: withdrawBalance) as!
        @FlowToken.Vault

        // Borrow the public capability to the receiving account (in this
        case the signer's own Vault)
        // This script could be modified to deposit into any account with
        a FungibleToken.Receiver capability
        self.receiver =
        signer.capabilities.borrow<&{FungibleToken.Receiver}>(/public/flowTokenRe
ceiver)!
```

```

        execute {
            // Deposit the withdrawn tokens into the receiving vault
            self.receiver.deposit(from: <-self.sentVault)
        }
    }
}

```

:::info

This is a basic example which only transfers tokens between a single user's COA & Flow account. It can be easily modified to transfer these tokens between any arbitrary accounts.

:::

Direct Calls to Flow EVM

To interact with smart contracts on the EVM, you can use the call function provided by the COA resource. This function takes the EVM address of the contract you want to call, the data you want to send, the gas limit, and the value you want to send. It will return a EVM.Result struct with the result of the call - you will need to handle this result in your Cadence code.

This transaction will use the signer's COA to call a contract method with the defined signature and args at a given EVM address, executing with the provided gas limit and value:

```

cadence
import "EVM"

transaction(evmContractHex: String, signature: String, args: [AnyStruct],
gasLimit: UInt64, flowValue: UFix64) {
    let coa: auth(EVM.Call) &EVM.CadenceOwnedAccount

    prepare(signer: auth(BorrowValue) &Account) {
        // Borrow an entitled reference to the COA from the storage
        location we saved it to
        self.coa = signer.storage.borrow<auth(EVM.Call)
        &EVM.CadenceOwnedAccount>(
            from: /storage/evm
        ) ?? panic("Could not borrow reference to the COA")
    }

    execute {
        // Deserialize the EVM address from the hex string
        let contractAddress = EVM.addressFromString(evmContractHex)
        // Construct the calldata from the signature and arguments
        let calldata = EVM.encodeABIWithSignature(
            signature,
            args
        )
        // Define the value as EVM.Balance struct
        let value = EVM.Balance(attoflow: 0)
        value.setFLOW(flow: flowValue)
    }
}

```

```

        // Call the contract at the given EVM address with the given
        data, gas limit, and value
        // These values could be configured through the transaction
        arguments or other means
        // however, for simplicity, we will hardcode them here
        let result: EVM.Result = self.coa.call(
            to: contractAddress,
            data: calldata,
            gasLimit: gasLimit,
            value: value
        )

        // Revert the transaction if the call was not successful
        // Note: a failing EVM call will not automatically revert the
        Cadence transaction
        // and it is up to the developer to use this result however it
        suits their application
        assert(
            result.status == EVM.Status.successful,
            message: "EVM call failed"
        )
    }
}

```

:::info

Notice that the calldata is encoded in the scope of the transaction. While developers can encode the calldata outside the scope of the transaction and pass the encoded data as an argument, doing so compromises the human-readability of Cadence transactions.

It's encouraged to either define transactions for each COA call and encoded the hardcoded EVM signature and arguments, or to pass in the human-readable arguments and signature and encode the calldata within the transaction. This ensures a more interpretable and therefore transparent transaction.

:::

Deploying a Contract to Flow EVM

To deploy a contract to the EVM, you can use the deploy function provided by the COA resource. This function takes the contract code, gas limit, and value you want to send. It will return the EVM address of the newly deployed contract.

This transaction will deploy a contract with the given code using the signer's COA:

```

cadence
import "EVM"

transaction(bytecode: String) {
    let coa: auth(EVM.Deploy) &EVM.CadenceOwnedAccount

    prepare(signer: auth(BorrowValue) &Account) {

```

```

        // Borrow an entitled reference to the COA from the storage
        location we saved it to
        self.coa = signer.storage.borrow<auth(EVM.Deploy)
&EVM.CadenceOwnedAccount>(
            from: /storage/evm
        ) ?? panic("Could not borrow reference to the COA")
    }

    execute {
        // Deploy the contract with the given compiled bytecode, gas
        limit, and value
        self.coa.deploy(
            code: bytecode.decodeHex(),
            gasLimit: 15000000, // can be adjusted as needed, hard coded
here for simplicity
            value: EVM.Balance(attoflow: 0)
        )
    }
}

```

More Information

For more information about Cadence Owned Accounts, see [Flow EVM Accounts](#).

Other useful snippets for interacting with COAs can be found [here](#).

vm-bridge.md:

```
---
title: Cross-VM Bridge
sidebarlabel: Cross-VM Bridge
sidebarposition: 6
---
```

Cross-VM Bridge

Flow provides the Cross-VM Bridge which enables the movement of fungible and non-fungible tokens between Cadence & EVM. The Cross-VM Bridge is a contract-based protocol enabling the automated and atomic bridging of tokens from Cadence into EVM with their corresponding ERC-20 and ERC-721 token types.

In the opposite direction, it supports bridging of arbitrary ERC-20 and ERC-721 tokens from EVM to Cadence as their corresponding FT or NFT token types.

The Cross-VM Bridge internalizes the capabilities to deploy new token contracts in either VM state as needed, resolving access to, and maintaining links between associated contracts. It additionally automates account and contract calls to enforce source VM asset burn or lock, and target VM token mint or unlock.

Developers wishing to use the Cross-VM Bridge will be required to use a Cadence transaction. Cross-VM bridging functionality is not currently available natively in EVM on Flow. By extension, this means that the EVM account bridging from EVM to Cadence must be a CadenceOwnedAccount (COA) as this is the only EVM account type that can be controlled from the Cadence runtime.

This FLIP outlines the architecture and implementation of the VM bridge. This document will focus on how to use the Cross-VM Bridge and considerations for fungible and non-fungible token projects deploying to either Cadence or EVM.

Deployments

The core bridge contracts can be found at the following addresses:

Contracts	Testnet	Mainnet
All Cadence Bridge contracts	0xdxfc20aee650fcfdf	0x1e4aa0b87d10b141
FlowEVMBridgeFactory.sol	0xf8146b4aef631853f0eb98dbe28706d029e52c52	0x1c6dea788ee774cf15bcd3d7a07ede892ef0be40
FlowEVMBridgeDeploymentRegistry.sol	0x8781d15904d7e161f421400571dea24cc0	db6938 0x8fdec2058535a2cb25c2f8cec65e8e0d0691f7b0
FlowEVMBridgedERC20Deployer.sol	0x4d45CaD104A71D19991DE3489ddC5C7B284cf2	63 0x49631Eac7e67c417D036a4d114AD9359c93491e7
FlowEVMBridgedERC721Deployer.sol	0x1B852d242F9c4C4E9Bb91115276f659D1D1f7	c56 0xe7c2B80a9de81340AE375B3a53940E9aeEAd79Df

And below are the bridge escrow's EVM addresses. These addresses are COAs and are stored in the same Flow account as you'll find the Cadence contracts (see above).

Network	Address
Testnet	0x0000000000000000000000000000000023f946ffbc8829bfd
Mainnet	0x00000000000000000000000000000000249250a5c27ecab3b

Interacting With the Bridge

:::info

All bridging activity in either direction is orchestrated via Cadence on COA EVM accounts. This means that all bridging activity must be initiated via a Cadence transaction, not an EVM transaction regardless of the directionality of the bridge request. For more information on the interplay between Cadence and EVM, see How EVM on Flow Works.

:::

Overview

The Flow EVM bridge allows both fungible and non-fungible tokens to move atomically between Cadence and EVM. In the context of EVM, fungible tokens are defined as ERC20 tokens, and non-fungible tokens as ERC721 tokens. In Cadence, fungible tokens are defined by contracts implementing the FungibleToken interface and non-fungible tokens implement the NonFungibleToken interface.

Like all operations on Flow, there are native fees associated with both computation and storage. To prevent spam and sustain the bridge account's storage consumption, fees are charged for both onboarding assets and bridging assets. In the case where storage consumption is expected, fees are charged based on the storage consumed at the current network storage rate.

Onboarding

Since a contract must define the asset in the target VM, an asset must be "onboarded" to the bridge before requests can be fulfilled.

Moving from Cadence to EVM, onboarding can occur on the fly, deploying a template contract in the same transaction as the asset is bridged to EVM if the transaction so specifies.

Moving from EVM to Cadence, however, requires that onboarding occur in a separate transaction due to the fact that a Cadence contract is initialized at the end of a transaction and isn't available in the runtime until after the transaction has executed.

Below are transactions relevant to onboarding assets:

```
<details>
<summary>onboardbytype.cdc</summary>

cadence onboardbytype.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/onboarding/onboardbytype.cdc

</details>

<details>
<summary>onboardbyevmaddress.cdc</summary>

cadence onboardbyevmaddress.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/onboarding/onboardbyevmaddre
ss.cdc

</details>
```

Bridging

Once an asset has been onboarded, either by its Cadence type or EVM contract address, it can be bridged in either direction, referred to by its Cadence type. For Cadence-native assets, this is simply its native type. For EVM-native assets, this is in most cases a templated Cadence contract deployed to the bridge account, the name of which is derived from the EVM contract address. For instance, an ERC721 contract at address 0x1234 would be onboarded to the bridge as EVMVMBridgedNFT0x1234, making its type identifier A.<BRIDGEADDRESS>.EVMVMBridgedNFT0x1234.NFT.

To get the type identifier for a given NFT, you can use the following code:

```
cadence
// Where nft is either a @{NonFungibleToken.NFT} or
&{NonFungibleToken.NFT}
nft.getType().identifier
```

You may also retrieve the type associated with a given EVM contract address using the following script:

```
<details>

<summary>getassociatedtype.cdc</summary>

cadence getassociatedtype.cdc
!from https://github.com/onflow/flow-evm-
bridge/blob/main/cadence/scripts/bridge/getassociatedtype.cdc
```

```
</details>
```

Alternatively, given some onboarded Cadence type, you can retrieve the associated EVM address using the following script:

```
<details>

<summary>getassociatedaddress.cdc</summary>

cadence getassociatedaddress.cdc
!from https://github.com/onflow/flow-evm-
bridge/blob/main/cadence/scripts/bridge/getassociatedevmaddress.cdc

</details>
```

NFTs

Any Cadence NFTs bridging to EVM are escrowed in the bridge account and either minted in a bridge-deployed ERC721

contract or transferred from escrow to the calling COA in EVM. On the return trip, NFTs are escrowed in EVM - owned by the bridge's COA - and either unlocked from escrow if locked or minted from a bridge-owned NFT contract.

Below are transactions relevant to bridging NFTs:

```
<details>

<summary>bridgenfttoevm.cdc</summary>

cadence bridgenfttoevm.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/nft/bridgenfttoevm.cdc

</details>

<details>

<summary>bridgenftfromevm.cdc</summary>

cadence bridgenftfromevm.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/nft/bridgenftfromevm.cdc

</details>
```

Fungible Tokens

Any Cadence fungible tokens bridging to EVM are escrowed in the bridge account only if they are Cadence-native. If the bridge defines the tokens, they are burned. On the return trip the pattern is similar, with the bridge burning bridge-defined tokens or escrowing them if they are EVM-native. In all cases, if the bridge has authority to mint on one side, it must escrow on the other as the native VM contract is owned by an external party.

With fungible tokens in particular, there may be some cases where the Cadence contract is not deployed to the bridge account, but the bridge still follows a mint/burn pattern in Cadence. These cases are handled via TokenHandler implementations. Also know that moving \$FLOW to EVM is built into the EVMAddress object so any requests bridging \$FLOW to EVM will simply leverage this interface; however, moving \$FLOW from EVM to Cadence must be done through the COA resource.

Below are transactions relevant to bridging fungible tokens:

```
<details>

<summary>bridgetokenstoevm.cdc</summary>
```

```

cadence bridgetokenstoevm.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/tokens/bridgetokenstoevm.cdc

</details>

<details>

<summary>bridgetokensfromevm.cdc</summary>

cadence bridgetokensfromevm.cdc
!from https://www.github.com/onflow/flow-evm-
bridge/blob/main/cadence/transactions/bridge/tokens/bridgetokensfromevm.c
dc

</details>

```

Prep Your Assets for Bridging

Context

To maximize utility to the ecosystem, this bridge is permissionless and open to any fungible or non-fungible token as defined by the respective Cadence standards and limited to ERC20 and ERC721 Solidity standards. Ultimately, a project does not have to do anything for users to be able to bridge their assets between VMs. However, there are some considerations developers may take to enhance the representation of their assets in non-native VMs. These largely relate to asset metadata and ensuring that bridging does not compromise critical user assumptions about asset ownership.

EVMBridgedMetadata

Proposed in [@onflow/flow-nft/pull/203](#), the EVMBridgedMetadata view presents a mechanism to both represent metadata from bridged EVM assets as well as enable Cadence-native projects to specify the representation of their assets in EVM. Implementing this view is not required for assets to be bridged, but the bridge does default to it when available as a way to provide projects greater control over their EVM asset definitions within the scope of ERC20 and ERC721 standards.

The interface for this view is as follows:

```

cadence
access(all) struct URI: MetadataViews.File {
    /// The base URI prefix, if any. Not needed for all URIs, but helpful
    /// for some use cases For example, updating a whole NFT collection's
    /// image host easily
    access(all) let baseURI: String?
    /// The URI string value

```

```

    /// NOTE: this is set on init as a concatenation of the baseURI and
the
    /// value if baseURI != nil
access(self) let value: String

access(all) view fun uri(): String

}

access(all) struct EVMBridgedMetadata {
    access(all) let name: String
    access(all) let symbol: String

    access(all) let uri: {MetadataViews.File}
}

```

This uri value could be a pointer to some offchain metadata if you expect your metadata to be static. Or you could couple the uri() method with the utility contract below to serialize the onchain metadata on the fly. Alternatively, you may choose to host a metadata proxy which serves the requested token URI content.

SerializeMetadata

The key consideration with respect to metadata is the distinct metadata storage patterns between ecosystem. It's critical for NFT utility that the metadata be bridged in addition to the representation of the NFTs ownership. However, it's commonplace for Cadence NFTs to store metadata onchain while EVM NFTs often store an onchain pointer to metadata stored offchain. In order for Cadence NFTs to be properly represented in EVM platforms, the metadata must be bridged in a format expected by those platforms and be done in a manner that also preserves the atomicity of bridge requests. The path forward on this was decided to be a commitment of serialized Cadence NFT metadata into formats popular in the EVM ecosystem.

For assets that do not implement EVMBridgedMetadata, the bridge will attempt to serialize the metadata of the asset as a JSON data URL string. This is done via the SerializeMetadata contract which serializes metadata values into a JSON blob compatible with the OpenSea metadata standard. The serialized metadata is then committed as the ERC721 tokenURI upon bridging Cadence-native NFTs to EVM. Since Cadence NFTs can easily update onchain metadata either by field or by the ownership of sub-NFTs, this serialization pattern enables token URI updates on subsequent bridge requests.

Opting Out

It's also recognized that the logic of some use cases may actually be compromised by the act of bridging, particularly in such a unique partitioned runtime environment. Such cases might include those that do not maintain ownership assumptions implicit to ecosystem standards.

For instance, an ERC721 implementation may reclaim a user's assets after a month of inactivity. In such a case, bridging that ERC721 to Cadence would decouple the representation of ownership of the bridged NFT from the actual ownership in the defining ERC721 contract after the token had been reclaimed - there would be no NFT in escrow for the bridge to transfer on fulfillment of the NFT back to EVM. In such cases, projects may choose to opt-out of bridging, but importantly must do so before the asset has been onboarded to the bridge.

For Solidity contracts, opting out is as simple as extending the BridgePermissions.sol abstract contract which defaults allowsBridging() to false. The bridge explicitly checks for the implementation of IBridgePermissions and the value of allowsBridging() to validate that the contract has not opted out of bridging.

Similarly, Cadence contracts can implement the IBridgePermissions.cdc contract interface.

This contract has a single method allowsBridging() with a default implementation returning false. Again, the bridge explicitly checks for the implementation of IBridgePermissions and the value of allowsBridging() to validate that the contract has not opted out of bridging. Should you later choose to enable bridging, you can simply override the default implementation and return true.

In both cases, allowsBridging() gates onboarding to the bridge. Once onboarded - a permissionless operation anyone can execute - the value of allowsBridging() is irrelevant and assets can move between VMs permissionlessly.

Under the Hood

For an in-depth look at the high-level architecture of the bridge, see FLIP #237

Additional Resources

For the current state of Flow EVM across various task paths, see the following resources:

- Flow EVM Equivalence forum post
- EVM Integration FLIP #223
- Gateway & JSON RPC FLIP #235

```
# ethers.md:

---
title: Ethers.js on Flow Blockchain
sidebarlabel: Ethers
sidebarposition: 1
---

Ethers.js
```

Ethers.js is a powerful JavaScript library for interacting with Ethereum and other EVM-compatible blockchain networks.

In this guide, we'll walk you through how to use ethers.js to interact with smart contracts on the Flow Blockchain.

```
---
```

Installation

To begin using ethers.js in your project, you'll need to install the package. You can do this by running the following command:

```
bash
bashCopy code
npm install --save ethers
```

Setup

After installing ethers.js, the next step is to import it into your project.

You can do this by adding the following line of code at the beginning of your JavaScript file:

```
jsx
const ethers = require('ethers');
```

Connecting to Flow

To connect to the Flow Blockchain using ethers.js, you need to create a new JsonRpcProvider instance with the appropriate RPC URL for Flow:

```
jsx
const ethers = require('ethers');

const url = 'https://testnet.evm.nodes.onflow.org/';
const provider = new ethers.providers.JsonRpcProvider(url);
```

Note: If you want to connect to the Flow testnet, replace the above URL with <https://mainnet.evm.nodes.onflow.org>.

Reading Data from the Blockchain

Once your provider is set up, you can start reading data from the Flow Blockchain. For instance, to retrieve the latest block number, you can use the `getBlockNumber` method:

```
jsx
async function getLatestBlock() {
  const latestBlock = await provider.getBlockNumber();
  console.log(latestBlock);
}
```

Writing Data to the Blockchain

To send transactions or write data to the Flow Blockchain, you need to create a Signer. This can be done by initializing a new `Wallet` object with your private key and the previously created Provider:

```
jsx
const privateKey = 'YOURPRIVATEKEY';
const signer = new ethers.Wallet(privateKey, provider);
```

Note: Replace '`YOURPRIVATEKEY`' with the actual private key of the wallet you want to use.

Interacting with Smart Contracts

`ethers.js` also enables interaction with smart contracts on the Flow Blockchain. To do this, create a `Contract` object using the ABI (Application Binary Interface) and the address of the deployed contract:

```
jsx
const abi = [
  // ABI of deployed contract
];

const contractAddress = 'CONTRACTADDRESS';

// read-only contract instance
const contract = new ethers.Contract(contractAddress, abi, provider);
```

For contracts that require writing, you'll need to provide a `Signer` object instead of a `Provider`:

```
jsx
// write-enabled contract instance
const contract = new ethers.Contract(contractAddress, abi, signer);
```

Note: Replace 'CONTRACTADDRESS' with the actual address of your deployed contract.

After setting up your Contract object, you can call methods on the smart contract as needed:

```
jsx
async function setValue(value) {
  const tx = await contract.set(value);
  console.log(tx.hash);
}

async function getValue() {
  const value = await contract.get();
  console.log(value.toString());
}
```

web3.js.md:

```
---
title: Web3.js on Flow Blockchain
sidebarlabel: Web3.js
sidebarposition: 2
---
```

Web3.js

Web3.js is a Javascript library for building on EVM-compatible networks.

It allows developers to interact with smart contracts, send transactions, and retrieve data from the network.

Prerequisites

```
:::info
This guide assumes you have the latest version of Node.js installed.
:::
```

To install web3 in your project, run the following command:

```
sh
npm install web3
```

Initializing Web3 with Flow

To use web3 in your project, start by importing the module and initializing your Web3 instance with a Flow RPC endpoint.

```
js
const { Web3 } = require('web3');
```

```
const web3 = new Web3('https://testnet.evm.nodes.onflow.org');
```

Note: If you want to connect to the Flow testnet, replace the above URL with <https://mainnet.evm.nodes.onflow.org>.

Querying The Blockchain

web3 provides a number of methods for querying the blockchain, such as getting the latest block number, querying account balances, and more.

You can try using some of these methods to verify that your web3 instance is working correctly.

```
js
// Get the latest block number
const blockNumber = await web3.eth.getBlockNumber();
console.log(blockNumber); // Latest block number

// Get the balance of an account
const balance = await web3.eth.getBalance('0x1234'); // Replace with any
address
console.log(balance); // Balance in attoFlow

// Get the chain ID
const chainId = await web3.eth.getChainId();
console.log(chainId);

// Get the gas price
const gasPrice = await web3.eth.getGasPrice();
console.log(gasPrice); // Gas price in attoFlow
```

For more information about other queries you can make web3, please see the official documentation.

Interacting with Smart Contracts

The web3 library allows developers to interact with smart contracts via the web3.eth.Contract API.

For this example we will use the following Storage contract.

We recommend deploying your own contract, which can be done using Hardhat or Remix.

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Storage {
    uint256 public storedData;

    function store(uint256 x) public {
```

```

        storedData = x;
    }

function retrieve() public view returns (uint256) {
    return storedData;
}
}

```

The ABI for this contract can be generated using the solc compiler, or another tool such as Hardhat or Remix.

Now that we have both the ABI and address of the contract, we can create a new Contract object for use in our application.

```

js
// Replace with the ABI of the deployed contract
const abi = [
    {
        inputs: [],
        stateMutability: 'nonpayable',
        type: 'constructor',
    },
    {
        inputs: [
            {
                internalType: 'uint256',
                name: 'x',
                type: 'uint256',
            },
        ],
        name: 'store',
        outputs: [],
        stateMutability: 'nonpayable',
        type: 'function',
    },
    {
        inputs: [],
        name: 'retrieve',
        outputs: [
            {
                internalType: 'uint256',
                name: '',
                type: 'uint256',
            },
        ],
        stateMutability: 'view',
        type: 'function',
    },
];
// Replace with the address of the deployed contract
const contractAddress = '0x4c7784ae96e7cfef0224a95059573e96f03a4e70';

```

```
// Create a new contract object with the ABI and address
const contract = new web3.eth.Contract(abi, contractAddress);
```

We can now interact with the contract on the network by using the contract object.

Reading State

State can be read from the contract by using the call function with one of the contract's methods. This will not change the state and will not send a transaction.

```
js
// Retrieve the current value stored in the contract
// (this is using the retrieve method from the contract with no
arguments)
const result = await contract.methods.retrieve().call();

console.log(result); // Current value stored in the contract
```

Changing State

We can mutate the state of the contract by sending a transaction to the network.

In order to send a transaction to the network, you will need an account with sufficient funds to pay for the transaction.

```
:::info
If you do not have an account yet, you can create one using the following command from your project's root directory:
```

```
sh
node -e "console.log(require('web3').eth.accounts.create())"
```

Note that this is not a secure way to generate an account, and you should use a more secure method in a production environment.

You can fund your account using the Flow Faucet.

```
:::
```

We can use the privateKeyToAccount function to create an Web3Account object from our account's private key.

```
js
// You must replace this with the private key of the account you wish to use
const account = web3.eth.accounts.privateKeyToAccount('0x1234');
```

Then, we can sign a transaction using the user's account and send it to the network.

```
js
const newValue = 1337; // Replace with any value you want to store

// Sign a transaction that stores a new value in the contract
// (this is using the store method from the contract with the new value
// as an argument)
let signed = await account.signTransaction({
  from: account.address,
  to: contractAddress,
  data: contract.methods.store(newValue).encodeABI(),
  gas: 10000000n, // Replace with the gas limit you want to use
  gasPrice: await web3.eth.getGasPrice(), // Replace with the gas price
  you want to use
});

// Send signed transaction to the network
const result = await
web3.eth.sendSignedTransaction(signed.rawTransaction);

// { status: 1, transactionHash: '0x1234', ... }
// status=1 means the transaction was successful
console.log(result);
```

Now that the transaction has been sent, the contract's state should have been updated. We can verify this by querying the contract's state again:

```
js
const result = await contract.methods.retrieve().call();
console.log(result); // New value stored in the contract
```

For more information about using smart contracts in web3.js, see the official documentation.

foundry.md:

```
---
title: Using Foundry with Flow
description: 'Using Foundry to deploy a Solidity contract to EVM on Flow.'
sidebarlabel: Foundry
sidebarposition: 5
---
```

Using Foundry with Flow

Foundry is a suite of development tools that simplifies the process of developing and deploying Solidity contracts to EVM networks. This guide will walk you through the process of deploying a Solidity contract to

Flow EVM using the Foundry development toolchain. You can check out the official Foundry docs [here](#).

In this guide, we'll deploy an ERC-20 token contract to Flow EVM using Foundry. We'll cover:

- Developing and testing a basic ERC-20 contract
- Deploying the contract to Flow EVM using Foundry tools
- Querying Testnet state
- Mutating Testnet state by sending transactions

Overview

To use Flow across all Foundry tools you need to:

1. Provide the Flow EVM RPC URL to the command you are using:

```
shell  
--rpc-url https://testnet.evm.nodes.onflow.org
```

2. Use the --legacy flag to disable EIP-1559 style transactions. Flow will support EIP-1559 soon and this flag won't be needed.

As an example, we'll show you how to deploy a fungible token contract to Flow EVM using Foundry. You will see how the above flags are used in practice.

Example: Deploying an ERC-20 Token Contract to Flow EVM

ERC-20 tokens are the most common type of tokens on Ethereum. We'll use OpenZeppelin starter templates with Foundry on Flow Testnet to deploy our own token called MyToken.

Installation

The best way to install Foundry, is to use the foundryup CLI tool. You can get it using the following command:

```
shell  
curl -L https://foundry.paradigm.xyz | bash
```

Install the tools:

```
shell  
foundryup
```

This will install the Foundry tool suite: forge, cast, anvil, and chisel.

You may need to reload your shell after foundryup installation.

Check out the official Installation guide for more information about different platforms or installing specific versions.

Wallet Setup

We first need to generate a key pair for our EVM account. We can do this using the cast tool:

```
shell  
cast wallet new
```

cast will print the private key and address of the new account. We can then paste the account address into the Faucet to fund it with some Testnet FLOW tokens.

You can verify the balance of the account after funding. Replace \$YOURADDRESS with the address of the account you funded:

```
shell  
cast balance --ether --rpc-url https://testnet.evm.nodes.onflow.org  
$YOURADDRESS
```

Project Setup

First, create a new directory for your project:

```
shell  
mkdir mytoken  
cd mytoken
```

We can use init to initialize a new project:

```
shell  
forge init
```

This will create a contract called Counter in the contracts directory with associated tests and deployment scripts. We can replace this with our own ERC-20 contract. To verify the initial setup, you can run the tests for Counter:

```
shell  
forge test
```

The tests should pass.

Writing the ERC-20 Token Contract

We'll use the OpenZeppelin ERC-20 contract template. We can start by adding OpenZeppelin to our project:

```
shell
forge install OpenZeppelin/openzeppelin-contracts
```

Rename `src/Counter.sol` to `src/MyToken.sol` and replace the contents with the following:

```
solidity
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(uint256 initialMint) ERC20("MyToken", "MyT") {
        mint(msg.sender, initialMint);
    }
}
```

The above is a basic ERC-20 token with the name `MyToken` and symbol `MyT`. It also mints the specified amount of tokens to the contract deployer. The amount is passed as a constructor argument during deployment.

Before compiling, we also need to update the test file.

Testing

Rename `test/Counter.t.sol` to `test/MyToken.t.sol` and replace the contents with the following:

```
solidity
pragma solidity ^0.8.20;

import {Test, console2, stdError} from "forge-std/Test.sol";
import {MyToken} from "../src/MyToken.sol";

contract MyTokenTest is Test {
    uint256 initialSupply = 420000;

    MyToken public token;
    address ownerAddress = makeAddr("owner");
    address randomUserAddress = makeAddr("user");

    function setUp() public {
        vm.prank(ownerAddress);
        token = new MyToken(initialSupply);
    }

    /*
     * Test general ERC-20 token properties
     */
    function testTokenProps() public view {
        assertEq(token.name(), "MyToken");
```

```

        assertEquals(token.symbol(), "MyT");
        assertEquals(token.decimals(), 18);
        assertEquals(token.totalSupply(), initialSupply);
        assertEquals(token.balanceOf(address(0)), 0);
        assertEquals(token.balanceOf(ownerAddress), initialSupply);
    }

    /
    Test Revert transfer to sender with insufficient balance
/
function testtransferRevertInsufficientBalance() public {
    vm.prank(randomUserAddress);

    vm.expectRevert(abi.encodeWithSignature("ERC20InsufficientBalance(address,uint256,uint256)", randomUserAddress, 0, 42));
    token.transfer(ownerAddress, 42);
}

/
Test transfer
/
function testtransfer() public {
    vm.prank(ownerAddress);
    assertEquals(token.transfer(randomUserAddress, 42), true);
    assertEquals(token.balanceOf(randomUserAddress), 42);
    assertEquals(token.balanceOf(ownerAddress), initialSupply - 42);
}

/
Test transferFrom with approval
/
function testtransferFrom() public {
    vm.prank(ownerAddress);
    token.approve(randomUserAddress, 69);

    uint256 initialRandomUserBalance =
token.balanceOf(randomUserAddress);
    uint256 initialOwnerBalance = token.balanceOf(ownerAddress);

    vm.prank(randomUserAddress);
    assertEquals(token.transferFrom(ownerAddress, randomUserAddress, 42),
true);
    assertEquals(token.balanceOf(randomUserAddress),
initialRandomUserBalance + 42);
    assertEquals(token.balanceOf(ownerAddress), initialOwnerBalance -
42);
    assertEquals(token.allowance(ownerAddress, randomUserAddress), 69 -
42);
}
}

```

You can now make sure everything is okay by compiling the contracts:

```
shell
forge compile
```

Run the tests:

```
shell
forge test
```

They should all succeed.

Deploying to Flow Testnet

We can now deploy MyToken using the forge create command. We need to provide the RPC URL, private key from a funded account using the faucet, and constructor arguments that is the initial mint amount in this case. We need to use the --legacy flag to disable EIP-1559 style transactions. Replace \$DEPLOYERPRIVATEKEY with the private key of the account you created earlier:

```
shell
forge create --rpc-url https://testnet.evm.nodes.onflow.org \
--private-key $DEPLOYERPRIVATEKEY \
--constructor-args 42000000 \
--legacy \
src/MyToken.sol:MyToken
```

The above will print the deployed contract address. We'll use it in the next section to interact with the contract.

Verifying a Smart Contract

Once deployed, you can verify the contract so that others can see the source code and interact with it from Flow's block explorer. You can use the forge verify-contract command:

```
shell
forge verify-contract --rpc-url https://testnet.evm.nodes.onflow.org/ \
--verifier blocksout \
--verifier-url https://evm-testnet.flowscan.io/api \
$DEPLOYEDMYTOKENADDRESS \
src/MyToken.sol:MyToken
```

:::info

When verifying a Mainnet contract, be sure to use the Mainnet RPC and block explorer URLs.

:::

Querying Testnet State

Based on the given constructor arguments, the deployer should own 42,000,000 MyT. We can check the MyToken balance of the contract owner. Replace \$DEPLOYEDMYTOKENADDRESS with the address of the deployed contract and \$DEPLOYERADDRESS with the address of the account you funded earlier:

```
shell
cast balance \
--rpc-url https://testnet.evm.nodes.onflow.org \
--erc20 $DEPLOYEDMYTOKENADDRESS \
$DEPLOYERADDRESS
```

This should return the amount specified during deployment. We can also call the associated function directly in the contract:

```
shell
cast call $DEPLOYEDMYTOKENADDRESS \
--rpc-url https://testnet.evm.nodes.onflow.org \
"balanceOf(address) (uint256)" \
$DEPLOYERADDRESS
```

We can query other data like the token symbol:

```
shell
cast call $DEPLOYEDMYTOKENADDRESS \
--rpc-url https://testnet.evm.nodes.onflow.org \
"symbol() (string)"
```

Sending Transactions

Let's create a second account and move some tokens using a transaction. You can use cast wallet new to create a new test account. You don't need to fund it to receive tokens. Replace \$NEWADDRESS with the address of the new account:

```
shell
cast send $DEPLOYEDMYTOKENADDRESS \
--rpc-url https://testnet.evm.nodes.onflow.org \
--private-key $DEPLOYERPRIVATEKEY \
--legacy \
"transfer(address,uint256) (bool)" \
$NEWADDRESS 42
```

We can check the balance of the new account:

```
shell
cast balance \
--rpc-url https://testnet.evm.nodes.onflow.org \
--erc20 $DEPLOYEDMYTOKENADDRESS \
$NEWADDRESS
```

The deployer should also own less tokens now:

```
shell
cast balance \
  --rpc-url https://testnet.evm.nodes.onflow.org \
  --erc20 $DEPLOYEDMYTOKENADDRESS \
  $DEPLOYERADDRESS
```

hardhat.md:

```
---
title: Flow Hardhat Guide
description: 'Using Hardhat to deploy a Solidity contract to EVM on
Flow.'
sidebarlabel: Hardhat
sidebarposition: 2
---
```

Flow Hardhat Guide

Hardhat is an Ethereum development tool designed to facilitate the deployment, testing, and debugging of smart contracts. It provides a streamlined experience for developers working with Solidity contracts.

Prerequisites

Node

Node v18 or higher, available for download [here](#).

For those new to Hardhat, we recommend exploring the official documentation to get acquainted. The following instructions utilize npm to initialize a project and install dependencies:

Wallet

You'll also need a wallet that supports EVM. For this guide, a MetaMask account and its corresponding private key will work.

```
shell
mkdir hardhat-example
cd hardhat-example

npm init

npm install --save-dev hardhat

npx hardhat init
```

> When prompted, select TypeScript and to use @nomicfoundation/hardhat-toolbox to follow along with this guide.

Fund Your Wallet

To deploy smart contracts, ensure your wallet has \$FLOW. Obtain funds by navigating to the Flow Faucet and entering your wallet address.

Deploying a Smart Contract with Hardhat

This section guides you through the process of deploying smart contracts on the Flow network using Hardhat.

Configuration

First, incorporate the Testnet network into your hardhat.config.ts:

```
javascript
import { HardhatUserConfig } from 'hardhat/config';
import '@nomicfoundation/hardhat-toolbox';

const config: HardhatUserConfig = {
  solidity: '0.8.24',
  networks: {
    testnet: {
      url: 'https://testnet.evm.nodes.onflow.org',
      accounts: [<PRIVATEKEY>], // In practice, this should come from an
      environment variable and not be committed
      gas: 500000, // Example gas limit
    },
  },
};

export default config;
```

To keep this example straightforward, we've included the account's private key directly in hardhat.config.ts. However, it is crucial to avoid committing private keys to your Git repository for security reasons. Instead, opt for using environment variables for safer handling of sensitive information.

Deploying HelloWorld Smart Contract

HelloWorld Smart Contract

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HelloWorld {
  // Declare a public field of type string.
  string public greeting;
```

```

// Constructor to initialize the greeting.
// In Solidity, the constructor is defined with the "constructor"
keyword.
constructor() {
    greeting = "Hello, World!";
}

// Public function to change the greeting.
// The "public" keyword makes the function accessible from outside
the contract.
function changeGreeting(string memory newGreeting) public {
    greeting = newGreeting;
}

// Public function that returns the greeting.
// In Solidity, explicit return types are declared.
function hello() public view returns (string memory) {
    return greeting;
}
}

```

Deploying:

1. Create a file named HelloWorld.sol under contracts directory.
2. Add above HelloWorld.sol contract code to new file.
3. Create a deploy.ts file in scripts directory.
4. Paste in the following TypeScript code.

```

javascript
import { ethers } from 'hardhat';

async function main() {
    const [deployer] = await ethers.getSigners();

    console.log('Deploying contracts with the account:', deployer.address);

    const deployment = await ethers.deployContract('HelloWorld');

    console.log('HelloWorld address:', await deployment.getAddress());
}

main()
    .then(() => process.exit(0))
    .catch((error) => {
        console.error(error);
        process.exit(1);
    });
}

```

5. Run npx hardhat run scripts/deploy.ts --network testnet in the project root.
6. Copy the deployed HelloWorld address. This address will be used in other scripts.

Output should look like this (with the exception that your address will be different):

```
shell
> npx hardhat run scripts/deploy.ts --network testnet
Deploying contracts with the account: ...
HelloWorld address: 0x3Fe94f43Fb5Cdb8268A801f274521a07F7b99dfb
```

You can now search for your deployed contract on the Flowscan block explorer!

Get HelloWorld Contract Greeting

Now, we want to get the greeting from the deployed HelloWorld smart contract.

```
javascript
import { ethers } from 'hardhat';
import HelloWorldABI from
'../artifacts/contracts/HelloWorld.sol/HelloWorld.json';

async function main() {
    // Replace with your contract's address
    const contractAddress = '0x3Fe94f43Fb5Cdb8268A801f274521a07F7b99dfb';
    // Get hardhat provider
    const provider = ethers.provider;
    // Create a new contract instance
    const helloWorldContract = new ethers.Contract(
        contractAddress,
        HelloWorldABI.abi,
        provider,
    );
    // Call the greeting function
    const greeting = await helloWorldContract.hello();
    console.log('The greeting is:', greeting);
}

main().catch((error) => {
    console.error(error);
    process.exit(1);
});
```

Steps:

1. Create a `getGreeting.ts` file in the `scripts` directory.
2. Paste contents of script above. Make sure to update the contract address with the one from deployment in earlier step.
3. Call script to get the greeting, `npx hardhat run scripts/getGreeting.ts --network testnet`
4. The output should be as follows:

```
shell
> npx hardhat run scripts/getGreeting.ts --network testnet
The greeting is: Hello, World!
```

Update Greeting on HelloWorld Smart Contract

Next, we'll add a script to update the greeting and log it.

```
javascript
import { ethers } from 'hardhat';
import HelloWorldABI from
'../artifacts/contracts/HelloWorld.sol/HelloWorld.json';

async function main() {
  const contractAddress = '0x3Fe94f43Fb5CdB8268A801f274521a07F7b99dfb';

  const newGreeting = process.env.NEWGREETING;
  if (!newGreeting) {
    console.error('Please set the NEWGREETING environment variable.');
    process.exit(1);
  }

  // Signer to send the transaction (e.g., the first account from the
  // hardhat node)
  const [signer] = await ethers.getSigners();

  // Contract instance with signer
  const helloWorldContract = new ethers.Contract(
    contractAddress,
    HelloWorldABI.abi,
    signer,
  );

  console.log('The greeting is:', await helloWorldContract.hello());

  // Create and send the transaction
  const tx = await helloWorldContract.changeGreeting(newGreeting);
  console.log('Transaction hash:', tx.hash);

  // Wait for the transaction to be mined
  await tx.wait().catch((error: Error) => {});
  console.log('Greeting updated successfully!');
  console.log('The greeting is:', await helloWorldContract.hello());
}

main().catch((error) => {
  console.error(error);
  process.exit(1);
});
```

Here are the steps to follow:

1. Create an updateGreeting.ts script in the scripts directory.
2. Paste in the TypeScript above, make sure to update the contract address with the one from deployment in earlier step.
3. Call the new script, NEWGREETING='Howdy!' npx hardhat run ./scripts/updateGreeting.ts --network testnet
4. The output should be

```
shell
> NEWGREETING='Howdy!' npx hardhat run ./scripts/updateGreeting.ts --
network testnet
The greeting is: Hello, World!
Transaction hash:
0x03136298875d405e0814f54308390e73246e4e8b4502022c657f04f3985e0906
Greeting updated successfully!
The greeting is: Howdy!
```

Verifying Contract

To verify your contract on Flowscan, you can update your Hardhat config file as such including the correct chainID, apiURL and browserURL:

```
javascript
import { HardhatUserConfig } from 'hardhat/config';
import '@nomicfoundation/hardhat-toolbox';
import "@nomicfoundation/hardhat-verify";

const PRIVATEKEY = vars.get("EVMPRIVATEKEY");

const config: HardhatUserConfig = {
  solidity: '0.8.24',
  networks: {
    testnet: {
      url: 'https://testnet.evm.nodes.onflow.org',
      accounts: [PRIVATEKEY], // In practice, this should come from an
      environment variable and not be committed
      gas: 500000, // Example gas limit
    },
    etherscan: {
      apiKey: {
        // Is not required by blockscout. Can be any non-empty string
        'testnet': "abc"
      },
      customChains: [
        {
          network: "testnet",
          chainId: 545,
          urls: {
            apiURL: "https://evm-testnet.flowscan.io/api",
            browserURL: "https://evm-testnet.flowscan.io/",
          }
        }
      ]
    }
  }
}
```

```
        ],
    },
    sourcify: {
        enabled: false
    }
};

export default config;
```

The verify plugin requires you to include constructor arguments with the verify task and ensures that they correspond to expected ABI signature. However, Blockscout ignores those arguments, so you may specify any values that correspond to the ABI. Execute the following command to verify the contract:

```
shell
npx hardhat verify --network testnet DEPLOYEDCONTRACTADDRESS "Constructor
argument 1"
```

integrating-metamask.md:

```
---
title: Integrating Metamask
sidebarposition: 1
---
```

Wallets & Configurations

This document shows how to integrate the Flow Network programmatically with your Dapp via MetaMask.

Metamask

Integrating additional networks into MetaMask can pose challenges for users who lack technical expertise and may lead to errors. Simplifying this process can greatly enhance user onboarding for your application. This guide demonstrates how to create a straightforward button within your frontend application to streamline the addition of the Flow network to MetaMask.

EIP-3035 & MetaMask

EIP-3035 is an Ethereum Improvement Proposal that defines an RPC method for adding Ethereum-compatible chains to wallet applications. Since March 2021 MetaMask has implemented that EIP as part of their MetaMask Custom Networks API.

Flow Network configuration

To add the Flow Testnet network to Metamask, add the following network configuration:

```
js
export const TESTNETPARAMS = {
  chainId: '0x221',
  chainName: 'Flow',
  rpcUrls: ['https://testnet.evm.nodes.onflow.org'],
  nativeCurrency: {
    name: 'Flow',
    symbol: 'FLOW',
    decimals: 18,
  },
  blockExplorerUrls: ['https://evm-testnet.flowscan.io/']
};
```

Adding Flow Network

To add this configuration to MetaMask, call the `walletaddEthereumChain` method which is exposed by the `web3` provider.

```
js
function addFlowTestnet() {
  injected.getProvider().then((provider) => {
    provider
      .request({
        method: 'walletaddEthereumChain',
        params: [TESTNETPARAMS],
      })
      .catch((error: any) => {
        console.log(error);
      });
  });
}
```

The variable, `injected`, is initialized as a `web3-react/injected-connector` used to interface with MetaMask APIs. Usage for other popular web frameworks is similar.

The typical usage would be to expose this button if you get errors when attempting to connect to MetaMask (i.e. Wrong Network or Error Connecting).

User Experience

Users of your app will need to first approve a connection to Metamask. After doing this, if you don't detect a successful Web3 network connection, you may present a dialog asking them to add the Flow network to their wallet.

!Metamask Network

After they approve, your app will be connected to the Flow network.

By using this approach to add the Flow network to Metamask, you can avoid manual user data entry and ensure that users are ready to interact with your dApp!

```
# remix.md:  
---  
title: Flow Remix Guide  
description: 'Using Remix to deploy a solidity contract to EVM on Flow.'  
sidebarlabel: Remix  
sidebarposition: 3  
---
```

Using Remix

Remix is an open-source, web-based development environment tailored for EVM smart contract development. It offers developers a comprehensive suite of tools for writing, deploying, and testing smart contracts in Solidity. For more information, visit [Remix](#).

Add the Flow Network to MetaMask

!Add Flow Network

Navigate to the Using EVM page to find the button to add the Flow network information to your metamask.

Fund Your Flow Account

Navigate to the Flow Testnet Faucet to obtain FLOW tokens necessary for deploying a smart contract.

Deploying a Smart Contract Using Remix

!Deploy Smart Contract

HelloWorld Smart Contract

```
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;  
  
contract HelloWorld {  
    // Declare a public field of type string.  
    string public greeting;  
  
    // Constructor to initialize the greeting.  
    // In Solidity, the constructor is defined with the "constructor"  
    keyword.  
    constructor() {  
        greeting = "Hello, World!";  
    }  
}
```

```

// Public function to change the greeting.
// The "public" keyword makes the function accessible from outside
the contract.
function changeGreeting(string memory newGreeting) public {
    greeting = newGreeting;
}

// Public function that returns the greeting.
// In Solidity, explicit return types are declared.
function hello() public view returns (string memory) {
    return greeting;
}
}

```

Steps to Deploy the HelloWorld Smart Contract

1. Create a file named HelloWorld.sol.
2. Select Solidity Compiler and compile HelloWorld.sol.
3. Select Deploy & Run Transactions.
4. Make sure to select Injected Provider - Metamask in Environment dropdown.
5. Deploy the HelloWorld smart contract.

Calling the Deployed Smart Contract

!Call Smart Contract

Using Ethers.js to Call the HelloWorld Smart Contract

1. Create a new get-greeting.js file under scripts.
2. Paste in the JavaScript code below.
3. Click on green play button to run.
4. Verify the greeting is "Hello World!".

```

javascript
// Import ethers from the ethers.js library
const { ethers } = require('ethers');

// Define the contract ABI
const contractABI = ['function hello() public view returns (string
memory)'];

// Define the contract address
const contractAddress = '0x8a120383e6057b1f3aef4fa9b89c2f1b0a695926';

// Connect to the Ethereum network
// This example uses the default provider from ethers.js, which connects
to the Ethereum mainnet.
// For a testnet or custom RPC, use
ethers.getDefaultProvider('networkName') or new
ethers.providers.JsonRpcProvider(url)

```

```

const provider = new ethers.providers.Web3Provider(window?.ethereum);

// Create a new contract instance
const contract = new ethers.Contract(contractAddress, contractABI,
provider);

// Call the hello function of the contract
async function getGreeting() {
  const greeting = await contract.hello();
  console.log(greeting);
}

// Execute the function
getGreeting();

```

Follow the steps below to change the greeting and retrieve the new greeting.

Updating the Deployed Smart Contract

!Update Smart Contract

1. Select the HelloWorld.sol file.
2. Select the Deploy and Run Transaction page.
3. Make sure to select Injected Provider - Metamask in Environment dropdown.
4. Type a new greeting in the text input next to orange changeGreeting button.
5. Click on the orange changeGreeting button.
6. Sign the Metamask transaction.
7. Verify the greeting has changed by re-running get-greeting.js script above.

vrf.md:

```
---
title: VRF (Randomness) in Solidity
sidebarlabel: VRF (Randomness) in Solidity
sidebarposition: 6
---
```

Introduction

Flow provides secure, native on-chain randomness that developers can leverage through Cadence Arch, a precompiled contract available on the Flow EVM environment. This guide will walk through how Solidity developers can use Cadence Arch to access Flow's verifiable randomness using Solidity.

What is Cadence Arch?

Cadence Arch is a precompiled

smart contract that allows Solidity developers on Flow EVM to interact with Flow's randomness and other network features like block height. This contract can be accessed using its specific address, and Solidity developers can make static calls to retrieve random values and other information.

Prerequisites

- Basic Solidity knowledge.
- Installed Metamask extension.
- Remix IDE for compilation and deployment.
- Flow EVM Testnet setup in Metamask.

Network Information for Flow EVM

Parameter	Value
Network Name	EVM on Flow Testnet
RPC Endpoint	https://testnet.evm.nodes.onflow.org
Chain ID	545
Currency Symbol	FLOW
Block Explorer	https://evm-testnet.flowscan.io

Steps to Connect Flow EVM Testnet to Metamask

1. Open Metamask and click Networks -> Add Network.
2. Enter the following details:
 - Network Name: EVM on Flow Testnet
 - RPC URL: <https://testnet.evm.nodes.onflow.org>
 - Chain ID: 545
 - Currency Symbol: FLOW
 - Block Explorer: <https://evm-testnet.flowscan.io>
3. Click Save and switch to the Flow EVM Testnet.

!image.png

Obtaining Testnet FLOW

You can fund your account with testnet FLOW using the Flow Faucet. Enter your Flow-EVM testnet address, and you'll receive testnet FLOW tokens to interact with smart contracts.

Solidity Code Example: Retrieving Random Numbers

Below is a simple Solidity contract that interacts with the Cadence Arch contract to retrieve a pseudo-random number.

```
solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract CadenceArchCaller {
    // Address of the Cadence Arch contract
    address constant public cadenceArch =
0x0000000000000000000000000000000100000000000000000000000000000001;

    // Function to fetch a pseudo-random value
    function revertibleRandom() public view returns (uint64) {
        // Static call to the Cadence Arch contract's revertibleRandom
        function
            (bool ok, bytes memory data) =
cadenceArch.staticcall(abi.encodeWithSignature("revertibleRandom()"));
        require(ok, "Failed to fetch a random number through Cadence
        Arch");
        uint64 output = abi.decode(data, (uint64));
        // Return the random value
        return output;
    }
}
```

Explanation of the Contract

1. Cadence Arch Address:

The `cadenceArch` variable stores the address of the Cadence Arch precompiled contract (`0x0000000000000000000000000000000100000000000000000000000000000001`), which is constant across Flow EVM.

2. Revertible Random:

The `revertibleRandom()` function makes a static call to the `revertibleRandom<uint64>()` function to fetch a pseudo-random number. If the call is successful, it decodes the result as a `uint64` random value.

Deploying and Testing the Contract

Compile and Deploy the Contract

1. Open Remix IDE.
2. Create a new file and paste the Solidity code above.

!image.png

3. Compile the contract by selecting the appropriate Solidity compiler version (0.8.x).

!image.png

4. Connect Remix to your Metamask wallet (with Flow EVM testnet) by selecting Injected Web3 as the environment.

!image.png

5. Deploy the contract.

!image.png

Call revertibleRandom

After deployment, you can interact with the contract to retrieve a random number.

Call the revertibleRandom() function in the left sidebar on the deployed contract. This will fetch a pseudo-random number generated by Flow's VRF.

!image.png

The result will be a uint64 random number generated on Flow EVM.

Generating Random Numbers in a Range

For use-cases like games and lotteries, it's useful to generate a random number within a specified range, the following example shows how to get a value between a min and max number.

```
solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract RandomInRange {
    address constant public cadenceArch =
0x0000000000000000000000000000000100000000000000001;

    // Generate a random number between min and max
    function getRandomInRange(uint64 min, uint64 max) public view returns
(uint64) {
        // Static call to the Cadence Arch contract's revertibleRandom
        // function
        (bool ok, bytes memory data) =
cadenceArch.staticcall(abi.encodeWithSignature("revertibleRandom()"));
        require(ok, "Failed to fetch a random number through Cadence
Arch");
        uint64 randomNumber = abi.decode(data, (uint64));
    }
}
```

```
// Return the number in the specified range
return (randomNumber % (max + 1 - min)) + min;
}

:::warning The above code is susceptible to the modulo bias,
particularly if the random number range is not a multiple of your desired range. To avoid this, you can use a more complex algorithm like rejection sampling, an example for which is provided in this repository. :::
```

Secure Randomness with Commit-Reveal Scheme in Solidity

The `revertibleRandom()` function can be directly used to generate a pseudo-random number. However, in certain situations, especially involving untrusted callers, this function exposes a vulnerability: the ability of a transaction to revert after seeing the random result.

The Issue with Using `revertibleRandom()` Directly:

- When an untrusted party calls a contract function that uses `revertibleRandom()`, they receive the random number during the transaction execution.
- Post-selection is the ability of the caller to abort the transaction if the random outcome is unfavorable. In this case, the user could choose to revert the transaction (for example, if they lose a bet) and attempt to call the function again in hopes of a better outcome.
- This can lead to a form of transaction reversion attack, where the randomness can be exploited by repeatedly attempting transactions until a favorable result is obtained.

Read More

For further details on Flow's randomness and secure development practices, check out the [Flow Randomness Documentation](#).

You can also view an example in both Solidity and Cadence of a random coin toss implementation using the VRF.

This documentation was contributed by Noah Naizir, a community developer.

```
# wagmi.md:
```

```
---
```

```
title: Viem & Wagmi
```

```
description: 'Using Wagmi to interact with Solidity contract to EVM on Flow.'
sidebarlabel: Viem & Wagmi
sidebarposition: 4
---

:::info
Make sure to use viem version 2.9.6 or greater. This version contains flow EVM networks
:::

Using viem
```

Flow networks have been added to viem chain definitions viem networks. This allows for convenient flow network configuration when using viem and wagmi.

Viem Flow Config

The configuration below uses Flow Testnet. Since this configuration is already in viem various properties are already set, like block explorer and json-rpc endpoint. See how this configuration is used in a nextjs wagmi web application below.

```
javascript
import { http, createConfig } from '@wagmi/core';
import { flowTestnet } from '@wagmi/core/chains';
import { injected } from '@wagmi/connectors';

export const config = createConfig({
  chains: [flowTestnet],
  connectors: [injected()],
  transports: {
    [flowTestnet.id]: http(),
  },
});
```

Using Next.js and Wagmi

This tutorial will guide you through creating a simple web application, connect to an EVM capable wallet and interact with the "HelloWorld" smart contract to get and set greetings. We will not dive into managing transactions.

Prerequisites

- Node.js installed on your machine
- A code editor (e.g., Visual Studio Code)
- Basic knowledge of React and Next.js

Step 1: Setting Up the Next.js Project

This tutorial will be following Wagmi getting-started manual tutorial

First, let's create a Wagmi project named flow-evm-wagmi. We will use npm but you are welcome to use yarn or bun.

```
bash
npm create wagmi@latest

project name flow-evm-wagmi
Select 'React' then 'next'
```

After Wagmi automatic installation procedure.

```
bash
cd flow-evm-wagmi
npm install
```

Step 2: Configuring Wagmi and Connecting the Wallet

Make sure you have Metamask installed and Flow network configured. Metamask and Flow blockchain.

Wagmi needs to know what networks to be aware of. Let's configure to use Flow Testnet by updating config.ts file with the following:

```
javascript
import { http, createConfig } from '@wagmi/core';
import { flowTestnet } from '@wagmi/core/chains';
import { injected } from '@wagmi/connectors';

export const config = createConfig({
  chains: [flowTestnet],
  connectors: [injected()],
  transports: {
    [flowTestnet.id]: http(),
  },
});
```

By default Wagmi configures many wallets, MetaMask, Coinbase Wallet, and WalletConnect as wallet providers. Above we simplify the code to only be interested in the Injected Provider, which we are interested in Metamask. Verify page.tsx code looks like the following.

```
javascript
'use client'

import { useAccount, useConnect, useDisconnect } from 'wagmi'

function App() {
  const account = useAccount()
  const { connectors, connect, status, error } = useConnect()
  const { disconnect } = useDisconnect()
```

```

    return (
      <>
      <div>
        <h2>Account</h2>

        <div>
          status: {account.status}
          <br />
          addresses: {JSON.stringify(account.addresses)}
          <br />
          chainId: {account.chainId}
        </div>

        {account.status === 'connected' && (
          <button type="button" onClick={() => disconnect()}>
            Disconnect
          </button>
        )}
      </div>

      <div>
        <h2>Connect</h2>
        {connectors.map((connector) => (
          <button
            key={connector.uid}
            onClick={() => connect({ connector })}
            type="button"
          >
            {connector.name}
          </button>
        ))}
        <div>{status}</div>
        <div>{error?.message}</div>
      </div>
    </>
  )
}

export default App

```

!Connect Metamask

This step relies on an already deployed HelloWorld contract. See Using Remix to deploy a smart contract on flow evm blockchain. Create or edit the simple page.tsx file in the app directory to have better styles, that's beyond this tutorial. We will modify page.tsx to add a new HelloWorld.tsx. Replace YOURCONTRACTADDRESS with your deployed address.

Step 3: Creating the Interface for HelloWorld Contract

Now, let's create a component to interact with the HelloWorld contract. Assume your contract is already deployed, and you have its address and ABI.

- Create a new file, `HelloWorld.ts`, in the components directory.
- Use Wagmi's hooks to read from and write to the smart contract:

```
javascript
import { useState } from 'react';
import {
  useContractRead,
  useContractWrite,
  useAccount,
  useConnect,
} from 'wagmi';
import contractABI from './HelloWorldABI.json'; // Import your contract's ABI

const contractAddress = 'YOURCONTRACTADDRESS';

const HelloWorld = () => {
  const [newGreeting, setNewGreeting] = useState('');
  const { address, isConnected } = useAccount();
  const { connect } = useConnect();

  const { data: greeting } = useContractRead({
    addressOrName: contractAddress,
    contractInterface: contractABI,
    functionName: 'hello',
  });

  const { write: changeGreeting } = useContractWrite({
    addressOrName: contractAddress,
    contractInterface: contractABI,
    functionName: 'changeGreeting',
    args: [newGreeting],
  });

  if (!isConnected) {
    return <button onClick={() => connect()}>Connect Wallet</button>;
  }

  return (
    <div>
      <p>Current Greeting: {greeting}</p>
      <input
        value={newGreeting}
        onChange={(e) => setNewGreeting(e.target.value)}
        placeholder="New greeting"
      />
      <button onClick={() => changeGreeting()}>Update Greeting</button>
    </div>
  );
}
```

```
export default HelloWorld;
```

Reminder: Replace YOURCONTRACTADDRESS with the actual address of your deployed HelloWorld contract.

Also notice you need the HelloWorld contract ABI, save this to a new file called HelloWorld.json in the app directory.

```
json
{
  "abi": [
    {
      "inputs": [],
      "stateMutability": "nonpayable",
      "type": "constructor"
    },
    {
      "inputs": [
        {
          "internalType": "string",
          "name": "newGreeting",
          "type": "string"
        }
      ],
      "name": "changeGreeting",
      "outputs": [],
      "stateMutability": "nonpayable",
      "type": "function"
    },
    {
      "inputs": [],
      "name": "greeting",
      "outputs": [
        {
          "internalType": "string",
          "name": "",
          "type": "string"
        }
      ],
      "stateMutability": "view",
      "type": "function"
    },
    {
      "inputs": [],
      "name": "hello",
      "outputs": [
        {
          "internalType": "string",
          "name": "",
          "type": "string"
        }
      ],
      "stateMutability": "view",
      "type": "function"
    }
  ]
},
```

```

        "stateMutability": "view",
        "type": "function"
    }
]
}

```

Step 4: Integrating the HelloWorld Component

Finally, import and use the HelloWorld component in your pages.tsx, throw it at the bottom of the render section.

```

javascript
import HelloWorld from './helloWorld'

// put at the bottom of the Connect section.
<div>
  <h2>Connect</h2>
  {connectors.map((connector) => (
    <button
      key={connector.uid}
      onClick={() => connect({ connector })}
      type="button"
    >
      {connector.name}
    </button>
  ))}
  <div>{status}</div>
  <div>{error?.message}</div>
</div>

// ⚡⚡⚡⚡⚡⚡⚡⚡⚡⚡
<div>
  <HelloWorld />
</div>

```

Now, you have a functional App that can connect to Metamask, display the current greeting from the "HelloWorld" smart contract, and update the greeting.

Test it by updating the greeting, signing a transaction in your Metamask then wait a minute then refresh the website. Handling transactions are outside of this tutorial. We'll leave that as a future task. Checkout Wagmi documentation

```
!Update HelloWorld Greeting
```

```
# governance.md:
```

```
---
title: Governance
```

description: Learn about Flow's governance model and how it's empowering our community of users and builders.

sidebarposition: 5
sidebarcustomprops:

icon: 

Participation

Participating in the governance process can take three forms:

- Being elected as a council member on the governing committee
- Putting forth a proposal for the community to vote on
- Staking to receive voting rights

Votes will be weighted based on locked tokens. All tokens staked by node operators will be eligible for voting, but other users can lock up their tokens to be given voting power. Anyone will be able to stake a Flow token to vote on issues (even if they aren't participating as a staked node).

Token Holder Rights

Tokens may be staked for operation or governance rights which gives holders the right to participate in running a node and/or to participate in public votes.

Process

Proposals can be brought forward on a public forum where they will be evaluated by the governing committee. All decisions are made publicly and any stakeholder has the opportunity to organize grassroots action to veto specific decisions or to vote in or remove council members.

To ensure the progress of the network, the elected council first assesses the proposal and selects an answer they agree to be the "default choice". Voters can freely vote how they choose, but having a well-considered default allows forward progress without being blocked by passive participants. All decisions are voted on by all participants and decisions made by the council must be ratified by a public vote on the network.

Timing

Vote outcomes and upcoming votes will be published every Friday by 7am PT. All upcoming votes are available for review and voting for at least two weeks following their publication.

Protocol Set Parameters

The following parameters will be set on the network on day 1 and will not be candidates for a public vote when the network first launches.

- The staking ratio preserved between each node type

- The maximum inflation rate
- The role of FLOW as the main reserve asset for collateralized secondary tokens (e.g. stablecoins)
- The mechanism through which transaction inclusion, computation, and storage fees are determined and paid for

Early Governance of the Protocol

Once governance is enabled, the community can participate in the following:

- Protocol upgrades, including things like: - the consensus algorithm - the low-level network communication structure - the execution environment
- the number of seats available for each node type
- Management of Ecosystem Development Fund, including: - issuance of grants - bug & feature bounties
- Selecting council members
- Committee budgets for each of the operational arms of the Foundation, including the executive, technical, operational, legal, pricing, financial, and marketing branches.
- Management of legal affairs, including: - enforcing license and patent infringements - issuing takedown notices and copyright infringement - freezing accounts if illegal activity occurs - updating the community, security, contribution policies

During the Bootstrapping Phase, anyone may apply online to be set as a Validator by the Company. Approved Validators must then Stake a fixed minimum of FLOWS based on Validator type. Other FLOW holders may become "Delegators" when they dedicate or "Delegate" their FLOWS to approved Node Operators as a signal that they believe that Validator to be an effective and honest participant of the network. Staking and Delegation features are already enabled as of the Effective Date.

Each Validator makes an individual decision of which Protocol Version they choose to use. Since the value of blockchain networks is primarily due to the collectively verified execution state, there is a strong incentive for Validators to choose a Protocol Version that is compatible with the Protocol Version selected by the majority of other Validators. As a practical matter, the Protocol Version chosen by the overwhelming majority of Validators is likely to be the most recent Protocol Version produced and recommended by the Core Team, provided the proposed changes are not contentious. However, if a significant fraction of the community disagrees with any aspect of the most recent Protocol Version, they can band together to use a previous Protocol Version, or some other Protocol Version defined independently from the Core Team. This process of a "contentious forking" is rare, but does have several precedents in other networks (REF: Ethereum Classic, Bitcoin Cash).

The process by which the Core Team chooses the updates for each new Protocol Version follows the open process described above, using GitHub as an open discussion platform to gauge the priorities and needs of the entire Flow ecosystem. The proposed changes by the Core Team will be announced and discussed well before they are implemented, and any community member can propose their own changes or contribute code updates

to implement any proposed changes. The details of a new Protocol Version are publicly available no less than 14 days before that version is formally recommended for use by Validators (a "Release"), with the complete implementation source code visible for no less than 7 days before a Release.

index.md:

```
---
```

```
sidebarposition: 1
```

```
---
```

```
import DocCardList from '@theme/DocCardList';
import { isSamePath } from '@docusaurus/theme-common/internal';
import { useDocsSidebar } from '@docusaurus/plugin-content-docs/client';
import { useLocation } from '@docusaurus/router';
```

Networks

```
<DocCardList items={useDocsSidebar().items.filter(item =>
!isSamePath(item.href, useLocation().pathname))}/>
```

accessing-mainnet.md:

```
---
```

```
title: Flow Mainnet
sidebarlabel: Mainnet
sidebarposition: 2
description: Guide to mainnet access
---
```

Accessing Flow Mainnet

The Flow Mainnet is available for access at this URL:

access.mainnet.nodes.onflow.org:9000

For example, to access the network using the Flow Go SDK:

```
go
import "github.com/onflow/flow-go-sdk/client"

func main() {
    flowAccessAddress := "access.mainnet.nodes.onflow.org:9000"
    flowClient,   := client.New(flowAccessAddress, grpc.WithInsecure())
    // ...
}
```

Account Creation

You can follow the Flow Port account creation steps to create a new mainnet account.

If you prefer watching a video, check out this tutorial:

```
<iframe
  width="560"
  height="315"
  src="https://www.youtube-nocookie.com/embed/vXui7uO4cIQ"
  title="YouTube video player"
  frameborder="0"
  allow="accelerometer; autoplay; clipboard-write; encrypted-media;
gyroscope; picture-in-picture"
  allowfullscreen
></iframe>
```

Generating a Non-Custodial Account

A non-custodial account will make sure you are the only one holding the keys to your account.

You can follow the following steps to add a non-custodial account:

First, generate a new key pair with the Flow CLI:

```
sh
> flow keys generate --network=mainnet
```

- Store private key safely and don't share with anyone!

```
Private Key      5b438...
Public Key       1bdc5...
```

> Note: By default, this command generates an ECDSA key pair on the P-256 curve. Keep in mind the CLI is intended for development purposes only and is not recommended for production use. Handling keys using a Key Management Service is the best practice.

Take a note of the public key and go back to Flow Port. Open the "Create a new account" page.

On the page, enter your public key from the CLI, ensure the hash algorithm is set to SHA3256 and the weight is set to 1000. Finally, check the box confirming correctness and hit 'Submit'.

> Important: Your account needs to have at least 0.002 FLOW for the account creation. More details can be found in this guide.

Once the transaction is sealed, you should scroll down to the events section and locate the flow.AccountCreated event with the newly generated address.

```
!flow-port-sealed
```

Make sure to take a note of the address. If you want to verify the public key for this address, you can visit [flow-view-source](#).

Important Mainnet Smart Contract Addresses

You can review all available core contracts deployed to the mainnet to identify which ones you want to import.

```
# accessing-testnet.md:
```

```
---
```

```
title: Flow Testnet
sidebarlabel: Testnet
sidebarposition: 3
description: Guide to Testnet access
---
```

About Flow Testnet

Flow Testnet is Flow's official testing and development network. It is intended to provide a staging and testing environment for dApp developers.

It aims to balance similarity with Mainnet with being a productive development environment, resulting in the following key differences:

- Testnet has significantly fewer validator nodes, resulting in a faster block rate compared to Mainnet
- Testnet is configured with shorter epochs (about 12 hours, compared to 7 days on Mainnet)
- Testnet receives software upgrades up to 2 weeks before Mainnet

Accessing Flow Testnet

Flow Testnet is available for access at this URL:

```
access.devnet.nodes.onflow.org:9000
```

For example, to access the network using the Flow Go SDK:

```
go
import "github.com/onflow/flow-go-sdk/client"

func main() {
    flowAccessAddress := "access.devnet.nodes.onflow.org:9000"
    flowClient,   := client.New(flowAccessAddress, grpc.WithInsecure())
    // ...
}
```

Generating Testnet Key Pair

You can generate a new key pair with the Flow CLI as follows:

```
sh  
> flow keys generate
```

 If you want to create an account on Testnet with the generated keys use this link:
<https://testnet-faucet.onflow.org/?key= cc1c3d72...>

- Store private key safely and don't share with anyone!

```
Private Key      246256f3...  
Public Key       cc1c3d72...
```

Note: By default, this command generates an ECDSA key pair on the P-256 curve. Keep in mind, the CLI is intended for development purposes only and is not recommended for production use. Handling keys using a Key Management Service is the best practice.

Account Creation and Token Funding Requests

Accounts and tokens for testing can be obtained through the testnet faucet. If you generated the keypair through the CLI, you can click on the URL provided to create an account and request testnet FLOW tokens.

Important Smart Contract Addresses

You can review all available core contracts deployed to the Testnet to identify which ones you want to import.

```
# index.md:
```

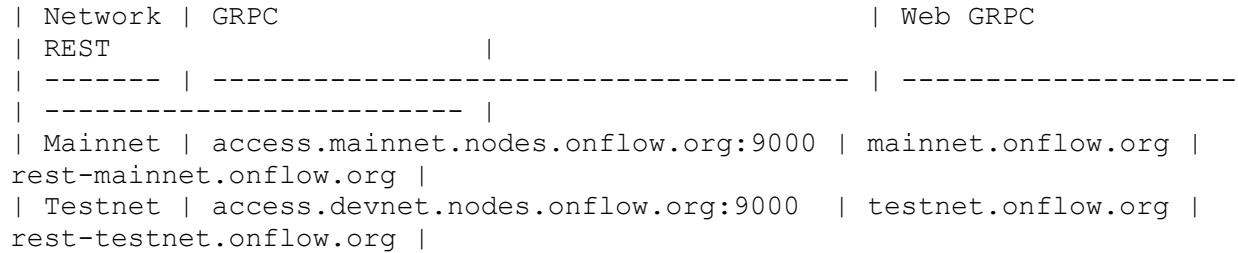
```
---  
title: Flow Networks  
sidebarposition: 1  
---
```

About Flow Networks

In addition to Mainnet, developers have access to the Testnet environment, which serves as an essential testing ground for applications and smart contracts prior to their deployment on Mainnet. This ensures that any potential issues can be identified and resolved in a controlled setting, mitigating risks associated with live deployment.

Furthermore, during network upgrades, Testnet receives updates ahead of Mainnet. This preemptive update process allows developers to comprehensively test their apps against the latest versions of the nodes, enhancements to the Cadence programming language, and core contract upgrades. This strategy guarantees that when these updates are eventually applied to Mainnet, applications and smart contracts will operate seamlessly, enhancing overall network stability and user experience.

How To Access These Networks?



For more information on how to access these networks, refer to the following guides:

- Flow Testnet
- Flow Mainnet

Network

There are two primary ways to access on-chain data within the Flow network; Access Nodes and Light nodes. Access Nodes are the node type that are most useful for developers, as they provide access to the Flow network via the following API endpoints:

- Flow Access API
 - Mainnet: `access.mainnet.nodes.onflow.org:9000`
 - Testnet: `access.devnet.nodes.onflow.org:9000`
- Status Page - Network status page

Running Your Own Node

If you're getting started you don't need to run your own node and you can use the above public nodes. The public access nodes are rate-limited, so as your product matures you might want to run your own node. There are multiple options available:

- Start with a Light (Observer) Node.
- You can also use a third-party provider like Quicknode.

Check out Running a Node for more information.

```
# index.md:
```

```
---
```

```
title: Flow Port
```

```
description: How to use Flow Port
```

```
--
```

Flow Port is an account management tool for Flow. Flow Port allows you to create an account using a wallet provider, manage your account and assets across Flow's VMs and perform staking and delegating actions on Flow.

Typically, your wallet provider will support most of these features. However, should your wallet provider not do so, or should you wish to use this tool for any other reason, Flow Foundation makes it available for you.

Creating an Account

In order to access Flow Port, you must have a valid Flow address. If you do not have a Flow address you can create one by installing a Flow compatible wallet.

Flow Wallet

Creating Account Through Flow Port: Navigate To Flow Port

1. Using Google Chrome, Navigate to Flow Port.
2. Click on 'Sign Up'
3. Click on Flow Wallet and choose Chrome extension or Mobile
4. You should be logged into Flow Port! You can now see your account address in Flow Port and access Flow features for your account

Ledger

Before You Start

1. Ensure you have:
 - a.) Ledger Live installed on your computer
 - b.) Initialized your Ledger Device.

Install the Flow App

1. Connect your Ledger Device to your computer and open Ledger Live.
2. Make sure your Ledger device firmware is up to date. You can check this by clicking on 'Manager' from the side navigation bar. Choose to install the update if one is available
 - a.) NOTE: Sometimes the install option doesn't show up, or it is not clickable. If this is the case, wait for a little bit of time to see if it appears, or restart the ledger live app if necessary.
3. On the Manager screen in Ledger Live and search for 'Flow'.
4. You should see the Flow App. Install it and follow the instructions on the device.
 - a.) NOTE: If the Flow App does not appear, it may be because you are on an outdated version. Please ensure you are on the most updated version.

Navigate to Flow Port to Create an Address

1. Navigate to Flow Port.
2. Click on 'Sign Up' if you need to create a new Flow Account.
3. Click on Ledger.
4. Follow the prompts on the screen. Plug in your Ledger device and open the Flow App.
5. Click on Create an account. Follow the prompts on your Ledger device.
6. Once your account address is created, you will be automatically logged into Flow Port.

Staking & Delegating

For a detailed walkthrough on how to use Flow Port for staking and delegating, please read the Flow Port staking walkthrough [How Do I Stake or Delegate?](#)

So you have decided you want to be a part of the Flow Network. Welcome! You are joining a group of people from all around the world that are a part of a movement centered around bringing decentralization, user empowerment, and transparency into the world. Below is a step-by-step guide that will assist you in the staking & delegation process.

Staking via a Custody Provider

If you are using a custody provider who controls your account and private keys for you, such as Kraken, Finoa, or Coinlist, they all have different policies and processes for what you need to do to stake your tokens, the rewards you receive, and the fees that they take from your staking rewards.

Starting a Manual Staking Transaction

1. You need to have FLOW in order to stake. Please see the FLOW Token reference for information on how to become a FLOW holder.
2. Once you have FLOW tokens in your account, you can start staking through Flow Port or, if applicable, with your custody provider.
3. If you are using Flow Port, log-in with your Flow account address and navigate to the Stake/Delegate page. See the Manual Staking/Delegating section below for more information about what to do next.

Manual Staking/Delegating

If you are not using a custody provider, there is more responsibility that you have to accept, because you have complete control of your tokens. You need to ensure that you are well informed about the staking process and potentially node operation process because you will have to manage those on your own. Please read the staking documentation before continuing with this guide.

Below are the various options you can choose. Please be aware, that at this time you can only have 1 stake or 1 delegate per account. This means that if you want to do multiple stakes, multiple delegates, or a mixture of stakes and delegates, you will need to create multiple accounts to do so. Please read them carefully as it will help you understand which route is best for your situation:

- Staking your own Node: You are responsible for running and maintaining a Flow Node. You are also solely responsible for providing the minimum stake for your selected node (minimum 135,000 FLOW) and you have the technical know-how and bandwidth to run and operate a node in the Flow protocol.
- Delegating: You have FLOW tokens and you want to stake, without having to run your own node and/or have the full minimum stake required to run your own node. You can 'delegate' any amount of your FLOW tokens to an existing node operator and you will earn rewards.

Please see a list here for all node operators that you can delegate to. This list will be updated as new node operators are onboarded onto the network.'

Staking Your Own Node

1. Once you have navigated to the staking/delegating page in Flow Port, click on the 'Stake a Node' option.
2. Next, select the type of node you will be running.
3. Input the amount of Flow you wish to stake with that node. You must stake at least the minimum in order for your stake request to be successfully processed. You are able to provide the minimum stake across multiple transactions. Meaning, you could execute your stake transaction with half of the minimum required. Then, before the next epoch, you can choose to 'Add Flow' to that pending stake to get it to the minimum stake required.
4. Run the bootstrapping instructions and provide the remaining technical details needed to stake a node.

Delegating

1. Once you have navigated to the staking/delegating page in Flow Port, click on the Delegate option.
2. Next, you will specify which node operator you would like to delegate to and how many tokens you want to delegate to them.
3. Execute the transaction. You will now see your pending delegation that will be processed during the next epoch.
4. At this point, you can also cancel the pending delegation. On the pending delegation, you will see an X that you can click to initiate the cancellation transaction.

I Have Successfully Executed a Stake Transaction, Now What?

- Now that you have executed a stake transaction in either Flow Port or your custody provider's portal, that transaction will sit in a pending

status until it is processed, which will be at the next Epoch Date (which is currently weekly).

- During the next Epoch, the transaction will be processed. If successful, the provided FLOW will be staked and the associated Node would be either a) included in the network protocol if it is a new node or b) continue to operate as is in the network protocol.

- You are now a part of Flow, and will begin to earn rewards for being a valued member of the network!

What Else Can I Do?

- Add additional stake to your existing stake. Any added FLOW will again sit in a pending status and be processed at the next epoch.

- Withdraw/re-stake your earned rewards. If you decide to withdraw your rewards, this action will happen instantly. If you decide to re-stake your rewards, the request will again sit in a pending status and will be processed at the next Epoch.

- Withdraw Rewards and send your earnings to other accounts. If you decide that you want to withdraw your rewards and send those earnings to other accounts via the 'Send FLOW' function, you should first withdraw your rewards. Once in your account, you can send these funds to any other account via the 'Send FLOW' option.

- Request to be unstaked from the network. The unstake request will sit in a pending status for two epochs. Once it is processed, the amount that has been unstaked will sit in your unstaked FLOW amount and can now be withdrawn or re-staked.

- Change the node you are staked/delegated to. If your staked/delegated node has no FLOW actively staked and you have completely withdrawn all unstaked amounts and rewards associated with the node, then you can move your stake to a different node. Click on the Change Node button to initiate this process. Please note that this feature is only visible once you get your active stake/delegate into the appropriate status.

FAQs

1. Why do I have multiple 'Keys' on my account?

If you created your account with Blocto, you will see that you have multiple keys that exist on your account in the 'Dashboard':

1 with weight 1 (device key): This is generated on Blocto and sent to users' device when they login with email.

1 with weight 999 (Blocto service key): This is kept in Blocto's secure key management service and is used to sign transaction.

1 with weight 1000 (recovery key): This is kept in Blocto's secure key management service and is only used when user wants to switch to non-custodial mode.

Normally if a user wants to send a Flow transaction, it requires signature from both the key on users' device and a key from Blocto service. Making it harder for hackers to steal your assets.

2. Where can I find a list of node operators to delegate to?

- a.) Please see a list here for all node operators that you can delegate to. This list will be updated as new node operators are onboarded onto the network.

3. I am currently running a node on the network already and have already gone through the staking process once. Do I need to execute a new stake every time there is a new epoch?

- a.) Once you successfully stake your node and become part of the network, you do not need to submit a new staking request each and every epoch. Your node will be automatically staked from epoch to epoch. This also means that your Node ID will remain the same from epoch to epoch. If you want to unstake your node from the network, then you will follow the process of unstaking your node.

4. I have a Blocto account and I see that I can stake both in Flow Port and in Blocto's mobile app. What is the difference?

- a.) If you go through Flow Port, you can choose any node operator within the Flow network to delegate any amount of your Flow Tokens to. If you go through Blocto's mobile site, you will only be able to stake to Blocto run nodes. You can read more about Blocto's staking process by referencing [here](#).

5. Do I need to use my Ledger device to view information about my account (e.g. my balance and current staked or delegated FLOW)?

- a.) No you do not! You only need your Ledger device to sign transactions. If you want to view your account, you can do so without your Ledger. You can do this by navigating directly to the appropriate desired page URL, while inputting your address into the URL itself. For quick reference, below is a list of these URLs and where you would input your address:

- Dashboard: [https://port.onflow.org/account/\[AccountAddress\]](https://port.onflow.org/account/[AccountAddress])
- Stake & Delegate: [https://port.onflow.org/stake-delegate/\[AccountAddress\]](https://port.onflow.org/stake-delegate/[AccountAddress])

6. I am clicking 'submit' to execute a transaction, but nothing is happening. How can I unblock myself?

- a.) Please disable any pop-up blockers and ad blockers you have and refresh the page. If you are still experiencing issues, please reach out via Discord in the appropriate channel.

```
# staking-guide.md:
```

```
---
```

```
title: Flow Port Staking Guide
```

```
---
```

This guide provides step-by-step instructions for using the Flow Port to stake your FLOW tokens and start earning rewards.

Currently, Flow Port only supports staking or delegating using tokens held in Blocto or Ledger wallets. If you're new to the concepts of staking and delegating you can read this guide to learn more.

First Step

When you arrive in Port, select Stake & Delegate from the left-hand menu. You should be taken to this page.

!Flow Port Staking pt. 0

From here you can decide whether to stake or delegate.

- Select Stake if you plan to stake a node you're running.
- Select Delegate to delegate your stake to another Node Operator. You don't need to know which Node Operator, you'll be provided with a list to choose from. If you are not running your own node you can skip directly to the delegation section

Stake a Node

Users who will be running their own nodes can stake them using the Flow Port.

Prerequisites

In order to stake your node, you'll need to have the required amount of FLOW for your node type.

You'll also need the following information about your node:

- Node ID
- Network Address
- Networking Key
- Staking Key
- Machine Account Public Key (for collection/consensus nodes only)

If you don't have this information, go here for instructions on how to acquire it.

Begin Staking

First, select the type of node you'll be running by choosing from the list. You must have the required amount of locked FLOW in your account.

!Flow Port Staking

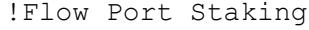
Once you selected your node type, click next and specify how much you'd like to stake. The minimum amount for your node type is required, but you may stake as much as you like beyond that. Here's the screen you should see:

!Flow Port Staking

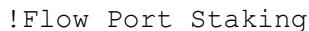
Clicking next will take you to the final screen, where you'll need to enter information about your node you previously obtained.

If you don't have this information, go here for instructions on how to acquire it.

Here's the screen you should see:

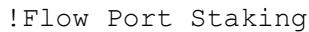
!Flow Port Staking

Clicking next will take you to a confirmation screen. This is your chance to double-check that you've entered your information correctly. If you're ready, check the box confirming your information and click submit to send the transaction that will stake your node! You should see a transaction status screen like this:

!Flow Port Staking

Note: If your transaction fails, double-check the information you provided.

If you return to the home screen, you'll be able to see your staking request in progress!

!Flow Port Staking

Delegating

Delegating is the process of staking your locked FLOW to nodes which are being run by another party.

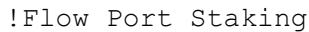
Prerequisites

In order to delegate your stake to another node, you'll need to know the node operator ID of the operator who is running the nodes you wish to stake.

Here is a list of node operator IDs you can delegate to: [List of Available Node Operators](#)

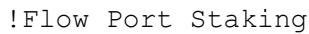
Enter a Node Operator ID

Simply enter the ID of the node operator of your choice and click next.

!Flow Port Staking

Enter an amount

Next you'll enter an amount of FLOW you would like to delegate. When delegating you may send any amount to the node operator.

!Flow Port Staking

Click next to reach the confirmation screen. Confirm the details of your delegation request and click submit!

!Flow Port Staking

Once your transaction is submitted, you can monitor its status from this screen, or return to the Flow Port home screen.

!Flow Port Staking

Note: If your transaction fails, double-check the information you provided.

That's it! You've successfully delegated stake to your chosen node operator!

Returning to Port

Within Flow Port, navigate to the 'Stake & Delegate' page to see details about your existing staked and/or delegated tokens.

This will also show you the rewards you have earned for your staked/delegated tokens.

!Flow Port Staking pt. 1

From here, you can do a few different things with your rewards:

- You can choose to re-stake them to the associated node.
- You can choose to withdraw them to your wallet.

Re-staking

Flow Port will not automatically re-stake your rewards.

To re-stake your rewards, simply hover your cursor over the 3 dots next to the rewards field:

!Flow Port Re-Staking

Click on the Restake option. This will take you to a screen that looks like the below. Input the amount of rewards you want to re-stake, acknowledge the transaction inputs and click submit:

!Flow Port Re-Staking

Once the transition is processed, you can reference the Stake & Delegate page again to see the pending stake now:

!Flow Port Re-Staking

Withdraw your Rewards

To withdraw your rewards, simply hover your cursor over the 3 dots next to the rewards field, and click on 'Withdraw'.

!Flow Port Re-Staking

Input the amount that you want to withdraw to your wallet, acknowledge the transaction inputs and click submit:

!Flow Port Re-Staking

Once the transition is processed, you can now see the withdrawn rewards in your balance and you are now free to do other actions with them (send them to other accounts, delegate to a node, etc).

index.md:

```
---
```

title: Flow's Network Architecture
sidebarposition: 1

Flow has pioneered a new paradigm of multi-role architecture that solves the core problem of today's blockchains. The result is a scalable, decentralized, and secure network which ensures user safety and long-term sustainability.

```
<div style={{textAlign:'center'}}>  
  
!flowgif  
  
</div>
```

To better understand the architecture, lets first understand the problems with the current blockchain. Then lets look at how Flow multi-role architecture solves these problems.

What are the biggest problems solved by Flow's Multi-role Architecture?

1. The blockchain trilemma

A blockchain should be fully decentralized, highly scalable and extremely secure. However a well-known problem with all blockchain is the blockchain trilemma - optimizing for any one edge comes at the cost of the other two.

You can have a chain that is decentralized and secure but not scalable e.g. Bitcoin and Ethereum or you can have a chain that is scalable and secure but not as decentralized e.g. Solana, Aptos and Sui.

While multi-chain systems like Cosmos, Layer 2 solutions (L2s) like Polygon, and cross-chain bridges offer innovative approaches to address these challenges, they divide the trust into separate and independent security zones and such zones with fewer validators can be more vulnerable to attacks and therefore less secure.

!scenario1

2. Disadvantaging end-users

Most blockchains, regardless of the number of participating nodes, inherently disadvantage individual end-users. This is because (colluding) nodes can censor user transactions or unfairly extract value from users in a phenomenon commonly known as Miner Extractable Value [MEV]. As a result, individual end users can end up paying an "invisible tax" or otherwise seeing their transactions fail due to MEV.

3. Energy inefficient and unsustainable

It is well established that Proof-of-Work chains like Bitcoin consume massive amounts of energy, require perpetual hardware upgrades for the miners to stay competitive, and are therefore extremely harmful to the environment. A Proof-of-Stake chain's environmental impact is less severe, but as web3 applications achieve mainstream adoption, every node in these chains will have to provide more and more hardware resources to meet the increasing throughput demand and the ever growing on-chain state. Vertically scaling the nodes implies higher energy consumption and environmental footprint.

Multi-role Architecture on Flow

!banner

In first-generation smart contract blockchains like Ethereum and Bitcoin, every node in the network performs all of the work associated with processing every transaction (including the entire network's history, account balances, smart contract code, etc.). While highly secure, it's also incredibly inefficient, and does not scale throughput (transaction per second, transaction latency) and capacity (on-chain data storage).

Most second-generation blockchain networks focus on improving performance in one of two ways:

1. They compromise decentralization by requiring that participating nodes run on powerful servers (e.g. Solana); or
2. They dramatically increase smart developer complexity by breaking up the network through mechanisms such as sharding (e.g. L2s such as Polygon).

The first approach is vulnerable to platform risk and cartel-like behavior. The second approach outsources the challenges of scaling the platform, effectively handing off the complexities of bridging the different strongly-federated ecosystems to application developers.

Flow offers a new path: pipelining applied to blockchain networks.

Pipelining is a well-established technique across various fields, from manufacturing to CPU design, for significantly increasing productivity. Flow leverages this concept by distributing the tasks typically handled by a full node in a monolithic blockchain architecture across four specialized roles: Collection, Consensus, Execution, and Verification. This division of labor between nodes occurs within the different validation stages for each transaction, rather than distributing transactions across different nodes as is done with sharding.

In other words, every Flow node still participates in the validation of every transaction, but they do so only at one of the stages of validation.

They can therefore specialize—and greatly increase the efficiency—for their particular stage of focus.

Flow node roles and what they do

Responsibility	Node type
What do the nodes of this role do?	
----- :-----: -----	
----- ----- -----	
----- ----- -----	
!collection Collection Collection nodes act as a censorship-resistant data availability layer, which caches transactions for subsequent execution. Collection nodes order transactions into batches known as collection.	
!consensus Consensus The consensus committee serves as the security authority in the network and orchestrates Flow's transaction processing pipeline. Consensus nodes order collections into blocks and commit execution results after verification.	
!execution Execution Execution nodes provide the computational resources for executing transactions and maintaining the state. Execution nodes execute the transaction and record state changes.	
!verification Verification Verification nodes ensure that transactions are truthfully executed.	
Verification nodes verify the work of the execution nodes. They either approve or disagree with their results, reporting their findings to the consensus nodes.	
!access Access Access Nodes route transactions into the network and replicate (parts of) the state and transaction results for external clients to query. Access node serve the API calls to send and read data from the chain.	

Further reading

1. Primer on multi-role architecture
2. Technical papers
3. Core protocol vision
4. Medium article from Jan which deep dives into the Flow architecture

In the next section, lets look at how Flow multi-role architecture solves those three big problems with blockchains.

solving-blockchain-trilemma.md:

```
---
title: Solving the blockchain trilemma
sidebarlabel: Solving the blockchain trilemma
sidebarposition: 2
---
```

Solving the blockchain trilemma

In a monolithic architecture, all nodes perform every task. As network usage grows, the transaction processing capacity of the individual nodes becomes a limiting factor, restricting the network's throughput and latency. The amount of data that can be stored on-chain is limited since nodes have a finite storage capacity. The only way to scale monolithic blockchains is by increasing the capacity of each node by adding more CPU, memory, and storage (i.e. vertical scaling, an approach taken by Solana). However, this solution comes at the cost of decentralization. As nodes scale vertically, they become more expensive to run, and eventually, only a few operators can afford to run such high-performance, high-capacity nodes. Worse, energy consumption for every node in the network increases over time, making the chain environmentally unsustainable.

Through its multi-role architecture, Flow implements a modular pipeline for processing transactions. This design allows the network to scale by tuning the level of decentralization at each specific step without sharding the state and fragmenting the network into smaller security zones.

The modular pipeline is composed of Collection, Consensus, Execution and Verification Nodes.

!pipeline

Separating Consensus from Compute

At a high level, the pipeline essentially separates consensus from transaction computation. Non-deterministic (or "subjective") processes such as determining the inclusion and order of transactions are decided by the broadly decentralized consensus committee. The deterministic (or "objective") task of computing the result of those ordered transactions is done independently by a small number of specialized execution nodes.

Collection and consensus are highly decentralized and achieve high levels of redundancy through a large number of lightweight, cost-effective nodes, numbering in the thousands, operated by several hundred different operators. These steps guarantee resilient transaction ordering (assuming that a malicious actor can only compromise a limited number of nodes).

In comparison, transaction execution has low decentralization and redundancy (10 or less) with more powerful and expensive nodes. To accommodate for the anticipated growth of on-chain state without sharding, only the execution nodes have to be scaled vertically. All

other node types can continue to run low-cost hardware. The execution nodes may eventually be scaled up to small data centers.

!scalingflow

Low decentralization for transaction execution might appear to compromise decentralization of the whole network, as it is conceivable that a malicious actor might compromise a dominant fraction of nodes participating in execution. However, correctness of the transaction results is still guaranteed by the verification step, which also requires reasonably high redundancy, again with a large number of lighter and less expensive verification nodes to withstand compromisation attempts.

Every node in Flow makes the protocol stronger, and the network can grow as needed to achieve different objectives:

- More censorship resistance? Add more collection nodes
- More decentralized block production? Add more consensus nodes
- Need to accommodate higher transaction throughput and state storage? Scale up execution nodes
- Do node operators want to reinforce network security with modest node hardware and low stake? Add more verification nodes.
- Need access to chain data locally? Add access nodes.

In contrast, when traditional Layer 1 blockchains add more nodes to increase decentralization, they do so without providing any additional benefits.

!veryingredundancy

> Flow's architectural goals are to provide a throughput of at least 1M TPS, ingest at least $\frac{1}{2}$ GB of transaction data per second and store and serve a very large state of one Petabyte and beyond.

Thus, Flow's multi-role architecture solves the blockchain trilemma:

1. Scalability: Scale to thousands of times higher throughput and on-chain storage capacity.
2. Decentralization: Except for the execution nodes, all nodes are light weight and low cost, lowering the barrier to entry and ensuring participation from a diverse set of node operators—big and small
3. Security: Maintain a shared non-sharded execution environment for all operations on the network and use a secure in-built platform to build on.

!trilemmmasolved

sustainability.md:

```
---
```

title: Sustainability
sidebarlabel: Sustainability
sidebarposition: 3

```
--
```

Sustainability with Flow

It's no secret that Proof of Stake blockchains are better for the environment.

As Web3 becomes more widely adopted, we engaged with Deloitte Canada to validate how much energy it uses.

And the results are astounding: Flow uses just 0.18 GWh annually, based on 2021 usage - or in simpler terms, minting an NFT on Flow takes less energy than a Google search or Instagram post.

In addition to operating on a Proof of Stake consensus system, Flow's multi-role node architecture securely divides the processing between specialized node types, making the network significantly more efficient than other blockchain architectures.

As network usage grows, vertical scaling is only needed for the execution nodes (as they execute transactions and persist all the chain state).

Because the increase in energy and hardware consumption over time is restricted to a small subset of the nodes in the network, this drastically limits the environmental footprint of the chain.

The overall energy use of the network won't increase significantly even if the activity increases by 100x or more, making the per-transaction energy footprint decrease over time.

Read more about it [here](#).

```
# user-safety.md:
```

```
---
```

```
title: User safety
sidebarlabel: User safety
sidebarposition: 4
---
```

User Safety with Flow

The monolithic node design of common L1s such as Bitcoin and Ethereum overly privileges operator control over block production.

This makes the chain vulnerable to censorship and MEV attacks. This problem is exacerbated by L2s with centralized sequencers. ERC-4337 is also susceptible to MEV on the user operations via bundlers.

```
!mev
```

Flow's multi-role architecture provides censorship & MEV resistance by design:

- Transactions are randomly assigned to collection nodes for inclusion in collections and eventually in blocks. Each collection node only sees a subset of transactions.

- There is already a distinct separation between the proposers (represented by the collection nodes) and the builders (represented by the consensus nodes). This separation essentially provides an inherent implementation of "proposer-builder separation," a concept currently being explored by Ethereum. With this separation, even if the collection nodes were to reorder the transactions, there is no incentive for the consensus nodes to prefer one collection node's proposal over another.

!mevprotection

index.md:

```
---
```

title: Node Operations
sidebarposition: 1

Hello Node Operator!

Flow nodes are vital components of the Flow blockchain. These nodes are responsible for a variety of network operations to maintain the distributed ledger.

Why Run a Node?

By running your own node, you have direct access to the evolving state of the network, without having to rely on third parties.

This increases privacy and security, reduces reliance on external servers, and helps balance load distribution.

By running a node, you also directly contribute to the security and decentralization of the whole network.

Flow multirole architecture makes it more scalable and provides several node types that you as a node operator can pick and choose from.

Which Node Should You Run?

The different types of nodes are described here. As node operator, you can choose to run any of the different types of node that best fits your needs.

The nodes are classified as follows,

!Flownodesdiagram.png

Light Node A.K.A. Observer Node

The light node is one of the easiest nodes to spin up and can be run by Dapp developers who need the latest block data available locally, e.g. a wallet application that needs to track the latest block ID and height. In addition to supporting dapps, an observer node can also be run by access node operators who want to scale their access nodes' endpoints. Access node operators can spin up geographically dispersed observer nodes which can talk to their staked access nodes and to each other.

The observer node is not staked but still provides the same API as the access node.

:::info

To run a light node, follow this guide

:::

Full Node

In a nutshell, Full Nodes are staked network participants that drive network progress, e.g. by creating and executing new blocks. They are the primary contributors to network safety (all of them validate the correctness of the consensus process and secure the network additionally through their role-specific tasks). In comparison, Light Nodes don't contribute to the networks progress. Though, they help to secure the network by also validating the integrity of the consensus process.

- The Access node is a full node that serves as an RPC node and acts as a gateway node for the network.
- The Validator node (Collection, Consensus, Verification and Execution) is a full node that plays a role in block generation.

Access Node

If you want local access to the protocol state data (blocks, collections, transactions) and do not want to use one of the community access nodes you can run an access node.

Dapp developers, chain explorers, chain analytics and others who want exclusive access to chain data and not be subject to the rate-limits on the community access node can choose to run an access node.

An access node is minimally staked for network security. The central goal for Access Nodes is to provide RPC functionality to its node operator.

In comparison, contributing to protocol progress (e.g. routing transactions to collector clusters, relaying blocks to the unstaked peer-to-peer network, etc.) should only take up a marginal fraction an Access Node's computational resources.

Furthermore, Access Node operators can freely rate-limit the amount of resources their Access Node dedicates to supporting the broader ecosystem. Therefore, Access Nodes do not receive staking rewards.

:::info

Launch an access node using QuickNode

<https://www.quicknode.com/chains/flow>

:::

:::info

To run a self-hosted access node, follow this guide

:::

:::tip

Alternately, instead of running an access node, you can use the Flow community access nodes or the ones run by any of the other node operators.

:::

Validator Node

You can also be a core participant in running the Flow network and contribute to securing it. Depending on your preference, you could run one or any combination of the following node roles:

- Collection Nodes collaboratively create batches of transactions (in Flow terminology collections).
- Consensus Nodes create blocks, schedule them for asynchronous execution, and commit execution results once they are verified (so called sealing). In addition, they orchestrate the Flow protocol and enforce protocol compliance.
- Execution Nodes asynchronously execute blocks. They are the power-houses in the protocol, providing the vast computational resources available to Flow transactions.
- Verification Nodes check the execution results in a distributed manner.

Nodes with these roles are staked and also receive staking rewards.

Running a Staked Node

To run a staked node (node type access, collection, consensus, verification or execution) the node must:

be registered with sufficient stake

be authorized by the governance working group

Before proceeding, ensure you have the stake required for your new node and that your node will be authorized by the governance working group (apply here).

To set up a new staked node after it has been authorized by the Flow governance working group, you will need to complete the following steps:

1. Provision the machine on which your node will run.
2. Generate and register your node identity.
3. Start your node!

```
# access-node-configuration-options.md:
```

```
---
```

```
title: Serving execution data
sidebarlabel: Execution Data
sidebarposition: 3
---
```

Flow chain data comprises of two parts,

1. Protocol state data - This refers to the blocks, collection, transaction that are being continuously added to the chain.
2. Execution state data - This refers to what makes up the execution state and includes transaction events and account balances.

The access node by default syncs the protocol state data and has been now updated to also sync the execution state data.

This guide provides an overview of how to use the execution data sync feature of the Access node.

```
<aside>
```

⚠ Nodes MUST be running v0.32.10+ or newer to enable execution data indexing.

```
</aside>
```

Setup node's directory

The access node typically has the following directory structure:

```
bash
$ tree flowaccess
flowaccess/
    ├── bootstrap
    │   ├── private-root-information (with corresponding AN data)
    │   ├── execution-state
    │   └── public-root-information
            └── node-id
```

```
    └── node-info.pub.NODEID.json
        └── root-protocol-state-snapshot.json (the genesis data)
    └── data (directory used by the node to store block data)
        └── execution-data
            └── execution-state
```

Setup execution data indexing

First, your node needs to download and index the execution data. There are 3 steps:

1. Enable Execution Data Sync
2. Download the root checkpoint file
3. Configure the node to run the indexer
4. Use the indexed data in the Access API.

As of mainnet24 / devnet49, Access nodes can be configured to index execution data to support local script execution, and serving all of the Access API endpoints using local data. There are different setup procedures depending on if you are enabling indexing immediately after a network upgrade, or at some point between upgrades.

Enable Execution Data Sync

This is enabled by default, so as long as you didn't explicitly disable it, the data should already be available.

1. Make sure that either `--execution-data-sync-enabled` is not set, or is set to true
2. Make sure that you have a path configured for `--execution-data-dir`, otherwise the data will be written to the running user's home directory, which is most likely inside the container's volume. For example, you can create a folder within the node's data directory `/data/execution-data/`.

There are some additional flags available, but you most likely do not need to change them.

Option 1: Enabling Indexing at the Beginning of a Spork

Download the root protocol state snapshot

The `root-protocol-state-snapshot.json` is generated for each spork and contains the genesis data for that spork. It is published and made available after each spork. The download location is specified here under `rootProtocolStateSnapshot`.

Store the `root-protocol-state-snapshot.json` into the `/bootstrap/public-root-information/` folder.

Download the root checkpoint

The root checkpoint for the network is used by Execution nodes and Access nodes to bootstrap their local execution state database with a known trusted snapshot. The checkpoint contains 18 files that make up the merkle trie used to store the blockchain's state.

The root checkpoint for each spork is hosted in GCP. You can find the link for the specific network in the sporks.json file. Here's the URL for mainnet24:

```
https://github.com/onflow/flow/blob/52ee94b830c2d413f0e86c1e346154f84c2643a4/sporks.json#L15
```

The URL in that file will point to a file named root.checkpoint. This is the base file and is fairly small. There are 17 additional files that make up the actual data, named root.checkpoint.000, root.checkpoint.001, ..., root.checkpoint.016. If you have gsutil installed, you can download them all easily with the following command.

```
bash  
gsutil -m cp "gs://flow-genesis-bootstrap/[network]-execution/public-root-information/root.checkpoint" .
```

Where [network] is the network you are downloading for. For example, mainnet-24 or testnet-49.

Once the files are downloaded, you can either move them to /bootstrap/execution-state/ within the node's bootstrap directory or put them in any mounted directory and reference the location with this cli flag: --execution-state-checkpoint=/path/to/root.checkpoint. The naming of files should be root.checkpoint..

Option 2: Enabling Indexing Mid-Spork

Identify the root checkpoint

The root checkpoint for the network is used by Execution and Access nodes to bootstrap their local execution state database with a known trusted snapshot. The checkpoint contains 18 files that make up the merkle trie used to store the blockchain's state.

Root checkpoints are periodically generated on Flow Foundation execution nodes and uploaded to a GCP bucket. You can see a list of available checkpoints here, or list them using the gsutil command

```
bash  
gsutil ls "gs://flow-genesis-bootstrap/checkpoints/"
```

The checkpoint paths are in the format flow-genesis-bootstrap/checkpoints/[network]/[epoch number]-[block height]/. Where

[network] is the network the checkpoint is from. For example, mainnet or testnet.

[epoch number] is the epoch number when the checkpoint was taken. You can find the current epoch number on the flowdiver home page.

[block height] is the block height at which the checkpoint was taken. Make sure that the checkpoint you select is from an epoch when your node was part of the network.

Download the root checkpoint

Once you have selected the checkpoint to download, you can download the files. If you have gsutil installed, you can download them all easily with the following command.

```
bash
gsutil -m cp "gs://flow-genesis-bootstrap/checkpoints/[network]/[epoch
number]-[block height]/root.checkpoint" .
```

Once the files are downloaded, you can either move them to /bootstrap/execution-state/ within the node's bootstrap directory or put them in any mounted directory and reference the location with this cli flag: --execution-state-checkpoint=/path/to/root.checkpoint. The naming of files should be root.checkpoint.

Download the root protocol state snapshot

Access nodes require that the data in the root checkpoint corresponds to the root block in the root-protocol-state-snapshot.json file. It's important to download the snapshot for the correct height, otherwise bootstrapping will fail with an error described in the Troubleshooting section.

You can download the root-protocol-state-snapshot.json file generated by the Execution from the same GCP bucket.

```
bash
gsutil cp "gs://flow-genesis-bootstrap/checkpoints/[network]/[epoch
number]-[block height]/root-protocol-state-snapshot.json" .
```

Alternatively, you can download it directly from a trusted Access node using the GetProtocolStateSnapshotByHeight gRPC endpoint with the corresponding height. You will get a base64 encoded snapshot which decodes into a json object. At this time, this endpoint is only support using the grpc API.

Store the root-protocol-state-snapshot.json into the /bootstrap/public-root-information/ folder.

Configure the node to run the indexer

Now you have the execution sync setup and the root checkpoint in place, it's time to configure the node to index all of the data so it can be used for script execution.

There are 2 cli flags that you will need to add:

- --execution-data-indexing-enabled=true This will enable the indexer.
- --execution-state-dir This defines the path where the registers db will be stored. A good default is on the same drive as the protocol db. e.g. /data/execution-state

Start your node

Now that all of the settings to enable indexing are in place, you can start your node.

At a minimum, you will need the following flags:

```
--execution-data-indexing-enabled=true  
--execution-state-dir=/data/execution-state  
--execution-data-sync-enabled=true  
--execution-data-dir=/data/execution-data
```

For better visibility of the process, you can also add

-p 8080:8080 - export port 8080 from your docker container, so you could inspect the metrics

--loglevel=info - for checking logs.

Notes on what to expect:

- On startup, the node will load the checkpoint into the execution-state db. For devnet48, this takes 20-30 min depending on the node's specs. For mainnet24, it takes >45 min. The loading time will increase over time. You can follow along with the process by grepping your logs for registerbootstrap.
- After the checkpoint is loaded, the indexer will begin ingesting the downloaded execution data. This will take several hours to days depending on if the data was already downloaded and the hardware specs of the node.
- If your node already had all the data, it will index all of it as quickly as possible. This will likely cause the node to run with a high CPU.

When you restart the node for the first time with syncing enabled, it will sync execution data for all blocks from the network.

Use the indexed data in the Access API

Setup Local Script Execution

Local execution is controlled with the `--script-execution-mode` flag, which can have one of the following values:

- `execution-nodes-only` (default): Requests are executed using an upstream execution node.
- `failover` (recommended): Requests are executed locally first. If the execution fails for any reason besides a script error, it is retried on an upstream execution node. If data for the block is not available yet locally, the script is also retried on the EN.
- `compare`: Requests are executed both locally and on an execution node, and a comparison of the results and errors are logged.
- `local-only`: Requests are executed locally and the result is returned directly.

There are a few other flags available to configure some limits used while executing scripts:

- `--script-execution-computation-limit`: Controls the maximum computation that can be used by a script. The default is 100,000 which is the same as used on ENs.
- `--script-execution-timeout`: Controls the maximum runtime for a script before it times out. Default is 10s.
- `--script-execution-max-error-length`: Controls the maximum number of characters to include in script error messages. Default is 1000.
- `--script-execution-log-time-threshold`: Controls the run time after which a log message is emitted about the script. Default is 1s.
- `--script-execution-min-height`: Controls the lowest block height to allow for script execution. Default: no limit.
- `--script-execution-max-height`: Controls the highest block height to allow for script execution. Default: no limit.
- `--register-cache-size`: Controls the number of registers to cache for script execution. Default: 0 (no cache).

Setup Using Local Data with Transaction Results and Events

Local data usage for transaction results and events are controlled with the `--tx-result-query-mode` and `--event-query-mode` corresponding flags, which can have one of the following values:

- `execution-nodes-only` (default): Requests are forwarded to an upstream execution node.
- `failover` (recommended): Requests are handled locally first. If the processing fails for any reason, it is retried on an upstream execution node. If data for the block is not available yet locally, the script is also retried on the EN.
- `local-only`: Requests are handled locally and the result is returned directly.

Troubleshooting

- If the root checkpoint file is missing or invalid, the node will crash. It must be taken from the same block as the `root-protocol-state-snapshot.json` used to start your node.

- If you don't set one the --execution-data-dir and --execution-state-dir flags, the data will be written to the home directory inside the container (likely /root). This may cause your container to run out of disk space and crash, or lose all data each time the container is restarted.
- If your node crashes or restarts before the checkpoint finishes loading, you will need to stop the node, delete the execution-state directory, and start it again. Resuming is currently not supported.
- If you see the following message then your checkpoint and root-protocol-state-snapshot are not for the same height.

```
json
{
  "level": "error",
  ...
  "module": "executionindexer",
  "submodule": "jobqueue",
  "error": "could not query processable jobs: could not read job at index 75792641, failed to get execution data for height 75792641: blob QmSZRu2SHN32d9SCkz9KXEtX3M3PozhzksMuYgNdMgmBwH not found",
  "message": "failed to check processables"
}
```

- You can check if the execution sync and index heights are increasing by querying the metrics endpoint:

```
curl localhost:8080/metrics | grep highestdownloadheight
curl -s localhost:8080/metrics | grep highestindexedheight
```

Execution Data Sync

The Execution Sync protocol is enabled by default on Access nodes, and uses the bitswap protocol developed by Protocol Labs to share data trustlessly over a peer-to-peer network. When enabled, nodes will download execution data for each block as it is sealed, and contribute to sharing the data with its peers. The data is also made available to systems within the node, such as the `ExecutionDataAPI`.

Below is a list of the available CLI flags to control the behavior of Execution Sync requester engine.

Flag	Type	Description
--	--	--
execution-data-sync-enabled	bool	Whether to enable the execution data sync protocol. Default is true
execution-data-dir	string	Directory to use for Execution Data database. Default is in the user's home directory.
execution-data-start-height	uint64	Height of first block to sync execution data from when starting with an empty Execution Data database. Default is the node's root block.
execution-data-max-search-ahead	uint64	Max number of heights to search ahead of the lowest outstanding execution data height. This limits

```
the number non-consecutive objects that will be downloaded if an earlier  
block is unavailable. Default is 5000. |  
| execution-data-fetch-timeout | duration | Initial timeout to use when  
fetching execution data from the network. timeout increases using an  
incremental backoff until execution-data-max-fetch-timeout. Default is  
10m. |  
| execution-data-max-fetch-timeout | duration | Maximum timeout to use  
when fetching execution data from the network. Default is 10s |  
| execution-data-retry-delay | duration | Initial delay for exponential  
backoff when fetching execution data fails. Default is 1s |  
| execution-data-max-retry-delay | duration | Maximum delay for  
exponential backoff when fetching execution data fails. Default is 5m |
```

<aside>

i Note: By default, execution data is written to the home directory of the application user. If your node is running in docker, this is most likely in the container's volume. Depending on how you configure your node, this may cause the node's boot disk to fill up.

As a best practice, specify a path with --execution-data-dir. A sensible default is to put it within the same directory as --datadir. e.g. --execution-data-dir=/data/executiondata.

</aside>

Execution Data Indexer

Below is a list of the available CLI flags to control the behavior of Execution Data Indexer.

Flag	Type	Description
--	--	--
execution-data-indexing-enabled	bool	Whether to enable the execution data indexing. Default is false
execution-state-dir	string	Directory to use for execution-state database. Default is in the user's home directory.
execution-state-checkpoint	string	Location of execution-state checkpoint (root.checkpoint.) files.
event-query-mode	string	Mode to use when querying events. one of [local-only, execution-nodes-only(default), failover]
tx-result-query-mode	string	Mode to use when querying transaction results. one of [local-only, execution-nodes-only(default), failover]

Below is a list of the available CLI flags to control the behavior of Script Execution.

Flag	Type	Description
--	--	--
script-execution-mode	string	Mode to use when executing scripts. one of [local-only, execution-nodes-only, failover, compare]
script-execution-computation-limit	uint64	Maximum number of computation units a locally executed script can use. Default: 100000

```
| script-execution-max-error-length | int | Maximum number characters to
include in error message strings. additional characters are truncated.
Default: 1000 |
| script-execution-log-time-threshold | duration | Emit a log for any
scripts that take over this threshold. Default: 1s |
| script-execution-timeout | duration | The timeout value for locally
executed scripts. Default: 10s |
| script-execution-min-height | uint64 | Lowest block height to allow for
script execution. Default: no limit |
| script-execution-max-height | uint64 | Highest block height to allow
for script execution. default: no limit |
| register-cache-type | string | Type of backend cache to use for
registers [lru, arc, 2q] |
| register-cache-size | uint | Number of registers to cache for script
execution. Default: 0 (no cache) |
| program-cache-size | uint | [experimental] number of blocks to cache
for cadence programs. use 0 to disable cache. Default: 0. Note: this is
an experimental feature and may cause nodes to become unstable under
certain workloads. Use with caution. |
```

Resources

FLIP: <https://github.com/onflow/flips/blob/main/protocol/20230309-accessnode-event-streaming-api.md>

Protobuf:

<https://github.com/onflow/flow/blob/master/protobuf/flow/executiondata/executiondata.proto>

access-node-setup.md:

```
---
title: Setting Up a Flow Access Node
sidebarlabel: Access Node Setup
sidebarposition: 2
---
```

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
```

This guide is for running a permissionless Access node on Flow. If you are planning to run a different type of staked node then see node bootstrap.

Permissionless Access nodes allow any operator to run a Flow Access node. Unlike the other staked nodes, a permissionless access node does not have to be approved by the service account before it can join the network, hence the term "permissionless". The goal is to make all node types permissionless and this is the first step towards achieving that goal.

Who Should Run a Permissionless Access Node?

dApp developers can choose to run their own private permissionless access node and move away from using the community access nodes. This will also allow them to not be subjected to the API rate limits of the public access nodes.

Node operators can also run their own permissionless access node and provide access to that node as a service.

Chain analytics, audit and exploration applications can run such an access node and do not have to rely on third parties for the state of the network.

Timing

New nodes are able to join the network each time a new epoch begins. An epoch is a period of time (approximately one week) when the node operators in the network are constant.

At epoch boundaries, newly staked node operators are able to join the network and existing node operators which have unstaked may exit the network.

You can read more about epochs [here](#).

In order to join the network at epoch N+1, the access node must be registered with at least 100 FLOW staked prior to the end of epoch N's Staking Auction Phase.

Currently on mainnet, the staking auction starts every Wednesday at around 20:00 UTC and ends on the next Wednesday at around 12:00 UTC. Since this deadline may shift slightly from epoch to epoch, we recommend the node be staked by Wednesday, 8:00 UTC to be able to join the network in the next epoch.

Confirmation of a new node's inclusion in epoch N+1 is included in the EpochSetup event.

!Flow Epoch Schedule

Limitations

There are five open slots for access nodes every epoch.

You can view the exact epoch phase transition time here under Epoch Phase.

To summarize,

Epoch	Epoch Phase	
:-----: :-----: :-----:		
N Staking auction starts Three new access node slots are opened. Anyone can register their access nodes		
N Staking auction ends Three of the nodes registered during this epoch are randomly selected to be a part of the network in the next epoch. No more nodes can register until the next epoch starts.		

```
| N+1      | Epoch N+1 starts      | The newly selected nodes can now
participate in the network. Three new slots are opened. |
```

How To Run a Permissionless Access Node?

:::note

To run an access node you will need to provision a machine or virtual machine to run your node software. Please follow the node-provisioning guide for it.

You can provision the machine before or after your node has been chosen.

:::

At a high level, to run a permissionless Access node, you will have to do the following steps:

1. Generate the node identity (private and public keys, node ID etc.).
2. Stake the node with 100 FLOW by the end of the staking phase of the current epoch (see timing) by providing the node information generated in step 1.
3. You can verify if your node ID was selected by the on-chain random selection process on Wednesday at around 20:00 UTC when the next epoch starts.
4. If your node ID was selected, you can provision and start running the node. If your node wasn't selected, your tokens will have been refunded to your unstaked bucket in the staking smart contract. When the next epoch begins, you can try committing tokens again in a future epoch to get a new spot.

Following is a detail explanation of these four steps.

If you want to run multiple access nodes, you will have to run through these steps for each node.

Step 1 - Generate Node Information

Download the Bootstrapping Kit

```
shell
curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-tools.tar
tar -xvf boot-tools.tar
```

```
shell CheckSHA256
sha256sum ./boot-tools/bootstrap
460cfefeb52b40d8b8b0c4641bc4e423bcc90f82068e95f4267803ed32c26d60 ./boot-
tools/bootstrap
```

> If you have downloaded the bootstrapping kit previously, ensure the SHA256 hash for it still matches. If not, re-download to ensure you are using the most up-to-date version.

Generate Your Node Identity

```

shell
#####
Generate Keys
$ mkdir ./bootstrap
YOURNODEADDRESS: FQDN associated to your instance
$ ./boot-tools/bootstrap key --address "<YOURNODEADDRESSGOESHERE>:3569" --
-role access -o ./bootstrap

shell Example
$./boot-tools/bootstrap key --address "flowaccess.mycompany.com:3569" --
role access -o ./bootstrap
<n1l> DBG will generate networking key
<n1l> INF generated networking key
<n1l> DBG will generate staking key
<n1l> INF generated staking key
<n1l> DBG will generate db encryption key
<n1l> INF generated db encryption key
<n1l> DBG assembling node information
address=flowaccess.mycompany.com:3569
<n1l> DBG encoded public staking and network keys
networkPubKey=f493a74704f6961ae7903e062ecd58d990672858eff99aece7fbccf3aa
02de8f1a624ecbf21a01e8b2f4a5854c231fbe218edd7762a34fea81f3958a215305
stakingPubKey=ae8dcf81f3a70d72036b7ba2c586ed37ed0eb82b9c0a4aab998a8420f98
894f94c14f84fa716e93654d3940fc0c8ff4d19b504c90a5b4918b28f421e9d3659dc2b7e
246025ebeffea0d83ccefe315d7ed346dbe412fdac51b64997d97d29f7e
<n1l> INF wrote file bootstrap/public-root-information/node-id
<n1l> INF wrote file bootstrap/private-root-information/private-node-
infoe737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5/node
-info.priv.json
<n1l> INF wrote file bootstrap/private-root-information/private-node-
infoe737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5/secr
etsdb-key
<n1l> INF wrote file bootstrap/public-root-information/node-
info.pub.e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5
.json

$tree ./bootstrap/
./bootstrap/
└── private-root-information
    └── private-node-
        infoe737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5
            ├── node-info.priv.json
            └── secretsdb-key
└── public-root-information
    ├── node-id
    └── node-
        info.pub.e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5
        .json

3 directories, 4 files

```

:::warning

Use a fully qualified domain name for the network address. Please also include the port number in the network address e.g.
flowaccess.mycompany.com:3569

:::

:::warning

Do not include the prefix http:// in the network address.

:::

:::tip

If you would like to stake multiple access nodes, please ensure you generate a unique identity for each node.

:::

Your node identity has now been generated. Your node ID can be found in the file ./bootstrap/public-root-information/node-id.

```
shell Example
$cat ./bootstrap/public-root-information/node-id
e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5
```

:::info

All your private keys should be in the bootstrap folder created earlier. Please take a back up of the entire folder.

:::

Step 2 - Stake the Node

You need to now register the node on chain by staking the node via Flow Port.

Here is a guide on how to use Flow port if you are not familiar with it. If you are staking via a custody provider or would like to directly submit a staking transaction instead follow this guide.

Fund your Flow account with at least 100.01 FLOW tokens, which covers the required stake plus the storage deposit.

On Flow port, choose Stake and Delegate -> Start Staking or Stake Again and then choose Access node as the option.

!chooseaccessflowport

On the next screen, provide the node details of your node.

Those node details (Node ID, Network Address, Networking Key and Staking Key) can be found in the file: ./bootstrap/public-root-information/node-info.pub.<node-id>.json.

```
shell Example
$cat ./bootstrap/public-root-information/node-info.pub.
e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5.json
{
  "Role": "access",
  "Address": "flowaccess.mycompany.com:3569",
  "NodeID":
"e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5",
  "Weight": 0,
  "NetworkPubKey":
"f493a74704f6961ae7903e062ecd58d990672858eff99aece7bfbccf3aa02de8f1a624ec
bf21a01e8b2f4a5854c231fbe218edd7762a34fea881f3958a215305",
  "StakingPubKey":
"ae8dcf81f3a70d72036b7ba2c586ed37ed0eb82b9c0a4aab998a8420f98894f94c14f84f
a716e93654d3940fc0c8ff4d19b504c90a5b4918b28f421e9d3659dc2b7e246025ebeffea
0d83cceeffe315d7ed346dbe412fdac51b64997d97d29f7e"
}
```

Example

```
!nodedetailspermissionlessan
```

On the next screen, ensure that you stake 100 FLOW token.

Example

```
!transactionregisternodepermissionlessan
```

Submit the Transaction.

Step 3 - Verify That Your Node ID Was Selected

On Wednesday at around 12:00 UTC, the staking auction for the current epoch will end and five nodes from candidate list of nodes will be chosen at random by the staking contract to be part of the next epoch.

:::note

If all 5 slots have been taken from the previous epoch, then no new access nodes will be chosen (see #limitations)

:::

There are several ways to verify whether your node was chosen as explained below.

When you stake the node, the tokens will show up under the tokensCommitted bucket. After the staking auction ends, if the node is

selected, the tokens remain in the tokensCommitted bucket and are moved to the tokensStaked bucket at the end of the epoch.
If the node is not selected, the tokens are moved to the tokensUnstaked bucket.

Check Using Flow Port

You can check these balances on Flow Port before and after the epoch transition that will occur on Wednesday (see timing).

When you stake the node, you should see the following on Flow Port under Stake & Delegate

!Stakednode

After the epoch transition, if you see your token balance under the Staked Amount then your node got chosen.

!Stakednode

Instead, if you see that your token balance is under the Unstaked Amount, then your node did not get chosen.

!Unstakednode

Check Using Flow CLI

You can also check these balance using Flow Cli. Once you have downloaded and installed Flow Cli, you can query the account balance using the command,
shell
flow accounts staking-info <your account address> -n mainnet

For Example, the following node was chosen as Tokens staked is 100.

```
shell Example
$ flow accounts staking-info 0xefdfb20806315bfa -n testnet

Account staking info:
  ID: "e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5"
    Initial Weight: 100
    Networking Address: "flowaccess.mycompany.com:3569"
    Networking Key: "f493a74704f6961ae7903e062ecd58d990672858eff99aece7bfbccf3aa02de8f1a624ec
bf21a01e8b2f4a5854c231fbe218edd7762a34fea881f3958a215305"
      Role: 5
      Staking Key: "ae8dcf81f3a70d72036b7ba2c586ed37ed0eb82b9c0a4aab998a8420f98894f94c14f84f
a716e93654d3940fc0c8ff4d19b504c90a5b4918b28f421e9d3659dc2b7e246025ebeffea
0d83cceeffe315d7ed346dbe412fdac51b64997d97d29f7e"
      Tokens Committed: 0.00000000
      Tokens To Unstake: 100.00000000
```

Tokens Rewarded:	0.00000000
Tokens Staked:	100.00000000
Tokens Unstaked:	0.00000000
Tokens Unstaking:	0.00000000
Node Total Stake (including delegators):	0.00000000

Epoch Setup Event

Alternatively, if you can monitor events, look for the epoch setup event that gets emitted by the epoch contract. That event is emitted at the end of epoch N's staking auction and contains a list of node IDs that are confirmed for the next epoch.

Step 4 - Start Your Node

If your node was selected as part of Step 3, you can now start your node.

First you'll need to provision a machine or virtual machine to run your node software. Please see follow the node-provisioning guide for it.

The access node can be run as a Docker container with the following command.

Be sure to set \$VERSION below to the version tag (e.g. v1.2.3) corresponding to the latest released version here for version releases). Set \$NODEID to your node's ID (see Generate Your Node Identity section above).

```
<Tabs>
<TabItem value="mainnet" label="Mainnet">

shell
docker run --rm \
-v $PWD/bootstrap:/bootstrap:ro \
-v $PWD/data:/data:rw \
--name flow-go \
--network host \
gcr.io/flow-container-registry/access:$VERSION \
--nodeid=$NODEID \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--secretsdir=/data/secrets \
--rpc-addr=0.0.0.0:9000 \
--http-addr=0.0.0.0:8000 \
--rest-addr=0.0.0.0:80 \
--rpc-metrics-enabled=true \
--bind 0.0.0.0:3569 \
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \
--dynamic-startup-access-
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \
--dynamic-startup-epoch-phase=EpochPhaseStaking \
--loglevel=error
```

```

</TabItem>
<TabItem value="testnet" label="Testnet">

    shell
    docker run --rm \
    -v $PWD/bootstrap:/bootstrap:ro \
    -v $PWD/data:/data:rw \
    --name flow-go \
    --network host \
    gcr.io/flow-container-registry/access:$VERSION \
    --nodeid=$NODEID \
    --bootstrapdir=/bootstrap \
    --datadir=/data/protocol \
    --secretsdir=/data/secrets \
    --rpc-addr=0.0.0.0:9000 \
    --http-addr=0.0.0.0:8000 \
    --rest-addr=0.0.0.0:80 \
    --rpc-metrics-enabled=true \
    --bind 0.0.0.0:3569 \
    --dynamic-startup-access-address=secure.testnet.nodes.onflow.org:9001 \
    --dynamic-startup-access-
    publickey=ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46d
    aeebf836f7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d \
    --dynamic-startup-epoch-phase=EpochPhaseStaking \
    --loglevel=error

</TabItem>
</Tabs>
```

For example, if your Node ID is e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5 and the software version is v1.2.3, the Docker command would be the following:

```

shell Example
docker run --rm \
    -v $PWD/bootstrap:/bootstrap:ro \
    -v $PWD/data:/data:rw \
    --name flow-go \
    --network host \
    gcr.io/flow-container-registry/access:v1.2.3 \
    --
    nodeid=e737ec6efbd26ef43bf676911cdc5a11ba15fc6562d05413e6589fccdd6c06d5 \
    --bootstrapdir=/bootstrap \
    --datadir=/data/protocol \
    --secretsdir=/data/secrets \
    --rpc-addr=0.0.0.0:9000 \
    --http-addr=0.0.0.0:8000 \
    --rest-addr=0.0.0.0:80 \
    --rpc-metrics-enabled=true \
    --bind 0.0.0.0:3569 \
    --dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \
```

```
--dynamic-startup-access-
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \
--dynamic-startup-epoch-phase=EpochPhaseStaking \
--loglevel=error
```

> If you would like your node to sync from the start of the last network upgrade, then please see the instructions here

Alternatively, you can build a binary for the access node to run it without using Docker.

To build the access node binary, see the instructions here.

Please make sure to git checkout the latest release tag before building the binary.

```
<Tabs>
<TabItem value="mainnet" label="Mainnet">

shell
$PWD/flow-go/flowaccessnode \
--nodeid=e1a8b231156ab6f2a5c6f862c933baf5e5c2e7cf019b509c7c91f4ddb0a13398
\
--bootstrapdir=$PWD/bootstrap \
--datadir=$PWD/data/protocol \
--secretsdir=$PWD/data/secrets \
--execution-data-dir=$PWD/data/executiondata \
--rpc-addr=0.0.0.0:9000 \
--secure-rpc-addr=0.0.0.0:9001 \
--http-addr=0.0.0.0:8000 \
--rest-addr=0.0.0.0:8070 \
--admin-addr=localhost:9002 \
--bind=0.0.0.0:3569 \
--dht-enabled=false \
--grpc-compressor=gzip \
--profiler-dir=$PWD/data/profiler \
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \
--dynamic-startup-access-
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \
--dynamic-startup-epoch-phase=EpochPhaseStaking
```

```
</TabItem>
<TabItem value="testnet" label="Testnet">

shell
$PWD/flow-go/flowaccessnode \
--nodeid=e1a8b231156ab6f2a5c6f862c933baf5e5c2e7cf019b509c7c91f4ddb0a13398
\
--bootstrapdir=$PWD/bootstrap \
--datadir=$PWD/data/protocol \
--secretsdir=$PWD/data/secrets \
--execution-data-dir=$PWD/data/executiondata \
```

```
--rpc-addr=0.0.0.0:9000 \
--secure-rpc-addr=0.0.0.0:9001 \
--http-addr=0.0.0.0:8000 \
--rest-addr=0.0.0.0:8070 \
--admin-addr=localhost:9002 \
--bind=0.0.0.0:3569 \
--dht-enabled=false \
--grpc-compressor=gzip \
--profiler-dir=$PWD/data/profiler \
--dynamic-startup-access-address=secure.testnet.nodes.onflow.org:9001 \
--dynamic-startup-access-
publickey=ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46d
aebfb836f7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d \
--dynamic-startup-epoch-phase=EpochPhaseStaking

</TabItem>
</Tabs>
```

For a more mature setup, it is recommended that you run the container using systemd as described here

>  The access node should now be up and running, and you should be able to query the node using Flow CLI or curl,

```
shell Example
flow blocks get latest --host localhost:9000
```

```
shell Example
curl http://localhost/v1/blocks?height=sealed
```

Monitoring and Metrics

The node publishes several Prometheus metrics. See Monitoring Node Health to setup node monitoring.

Node Status

The metrics for the node should be able to provide a good overview of the status of the node. If we want to get a quick snapshot of the status of the node, and if it's properly participating in the network, you can check the consensuscompliancefinalizedheight or consensuscompliancesealedheight metric, and ensure that it is not zero and strictly increasing.

```
shell
curl localhost:8080/metrics | grep consensuscompliancesealedheight

HELP consensuscompliancesealedheight the last sealed height
TYPE consensuscompliancesealedheight gauge
consensuscompliancesealedheight 1.132054e+06
```

FAQs

Will the access node receive rewards?

No, the access nodes do not receive any rewards.

Why is there a 100 FLOW token minimum?

As mentioned in the FLIP, the minimum is required to prevent certain vulnerabilities in the smart contract that are a result of having a zero minimum stake requirement.

Can the Access node be unstaked?

Yes, like any other staked node, the Access node can be unstaked. The staked tokens will be moved to the unstaked bucket in the subsequent epoch.

How to see all the access nodes that have staked?

When the nodes are initially staked, they are all added to the candidate list of nodes before the end of the epoch staking phase. The list can be retrieved from the chain by executing the getcandidatenodes script which returns the candidate list for the current epoch.

```
shell
$ flow scripts execute
./transactions/idTableStaking/scripts/getcandidatenodes.cdc -n mainnet
```

How to check the availability of open access nodes slots for the next epoch?

The limits for the open slots are defined in the staking contract and can be queried from the chain by executing the getslotlimits script.

Node types are defined here

```
shell
$ flow scripts execute
./transactions/idTableStaking/scripts/getslotlimits.cdc --args-json '[{"type": "UInt8", "value": "5"}]' -n mainnet
Result: 118
```

Example: there are 115 access nodes already part of the network. Hence, the total number of new nodes that can join are $118 - 115 = 3$.

```
# access-api.md:
```

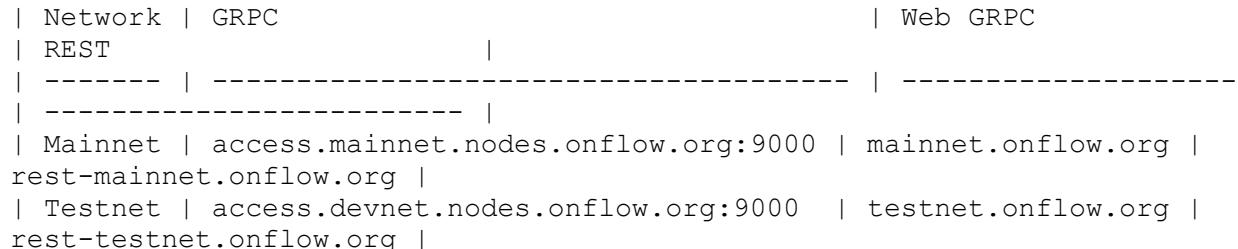
```
---
title: Flow Access API Specification
sidebarlabel: Access API
sidebarposition: 1
---
```

The Access API is implemented as a gRPC service.

A language-agnostic specification for this API is defined using Protocol Buffers, which can be used to generate client libraries in a variety of programming languages.

- Flow Access API protobuf source files

Flow Access Node Endpoints



```
---
```

Ping

Ping will return a successful response if the Access API is ready and available.

```
proto
rpc Ping(PingRequest) returns (PingResponse)
```

If a ping request returns an error or times out, it can be assumed that the Access API is unavailable.

Request

```
proto
message PingRequest {}
```

Response

```
proto
message PingResponse {}
```

```
---
```

Block Headers

The following methods query information about block headers.

GetLatestBlockHeader

GetLatestBlockHeader gets the latest sealed or unsealed block header.

```
proto
rpc GetLatestBlockHeader (GetLatestBlockHeaderRequest) returns
(BlockHeaderResponse)
```

Request

```
proto
message GetLatestBlockHeaderRequest {
    bool issealed = 1;
}
```

Response

```
proto
message BlockHeaderResponse {
    entities.BlockHeader block = 1;
    entities.BlockStatus blockstatus = 2;
    entities.Metadata metadata = 3;
}
```

GetBlockHeaderByID

GetBlockHeaderByID gets a block header by ID.

```
proto
rpc GetBlockHeaderByID (GetBlockHeaderByIDRequest) returns
(BlockHeaderResponse)
```

Request

```
proto
message GetBlockHeaderByIDRequest {
    bytes id = 1;
}
```

Response

```
proto
message BlockHeaderResponse {
    entities.BlockHeader block = 1;
    entities.BlockStatus blockstatus = 2;
}
```

```
    entities.Metadata metadata = 3;
}
```

GetBlockHeaderByHeight

GetBlockHeaderByHeight gets a block header by height.

```
proto
rpc GetBlockHeaderByHeight (GetBlockHeaderByHeightRequest) returns
(BlockHeaderResponse)
```

Request

```
proto
message GetBlockHeaderByHeightRequest {
    uint64 height = 1;
}
```

Response

```
proto
message BlockHeaderResponse {
    entities.BlockHeader block = 1;
    entities.BlockStatus blockstatus = 2;
    entities.Metadata metadata = 3;
}
```

Blocks

The following methods query information about full blocks.

GetLatestBlock

GetLatestBlock gets the full payload of the latest sealed or unsealed block.

```
proto
rpc GetLatestBlock (GetLatestBlockRequest) returns (BlockResponse)
```

Request

```
proto
message GetLatestBlockRequest {
    bool issealed = 1;
    bool fullblockresponse = 2;
}
```

Response

```
proto
message BlockResponse {
    entities.Block block = 1;
    entities.BlockStatus blockstatus = 2;
    entities.Metadata metadata = 3;
}
```

GetBlockByID

GetBlockByID gets a full block by ID.

```
proto
rpc GetBlockByID (GetBlockByIDRequest) returns (BlockResponse)
```

Request

```
proto
message GetBlockByIDRequest {
    bytes id = 1;
    bool fullblockresponse = 2;
}
```

Response

```
proto
message BlockResponse {
    entities.Block block = 1;
    entities.BlockStatus blockstatus = 2;
    entities.Metadata metadata = 3;
}
```

GetBlockByHeight

GetBlockByHeight gets a full block by height.

```
proto
rpc GetBlockByHeight (GetBlockByHeightRequest) returns (BlockResponse)
```

Request

```
proto
message GetBlockByHeightRequest {
    uint64 height = 1;
    bool fullblockresponse = 2;
}
```

Response

```
proto
message BlockResponse {
    entities.Block block = 1;
    entities.BlockStatus blockstatus = 2;
    entities.Metadata metadata = 3;
}
```

Collections

The following methods query information about collections.

GetCollectionByID

GetCollectionByID gets a collection by ID.

```
proto
rpc GetCollectionByID (GetCollectionByIDRequest) returns
(CollectionResponse)
```

Request

```
proto
message GetCollectionByIDRequest {
    bytes id = 1;
}
```

Response

```
proto
message CollectionResponse {
    entities.Collection collection = 1;
    entities.Metadata metadata = 2;
}
```

GetFullCollectionByID

GetFullCollectionByID gets a collection by ID, which contains a set of transactions.

```
proto
rpc GetFullCollectionByID(GetFullCollectionByIDRequest) returns
(FullCollectionResponse);
```

Request

```
proto
message GetFullCollectionByIDRequest {
    bytes id = 1;
}
```

Response

```
proto
message FullCollectionResponse {
    repeated entities.Transaction transactions = 1;
    entities.Metadata metadata = 2;
}
```

Transactions

The following methods can be used to submit transactions and fetch their results.

SendTransaction

SendTransaction submits a transaction to the network.

```
proto
rpc SendTransaction (SendTransactionRequest) returns
(SendTransactionResponse)
```

SendTransaction determines the correct cluster of collection nodes that is responsible for collecting the transaction based on the hash of the transaction and forwards the transaction to that cluster.

Request

SendTransactionRequest message contains the transaction that is being request to be executed.

```
proto
message SendTransactionRequest {
    entities.Transaction transaction = 1;
}
```

Response

SendTransactionResponse message contains the ID of the submitted transaction.

```
proto
message SendTransactionResponse {
    bytes id = 1;
    entities.Metadata metadata = 2;
}
```

GetTransaction

GetTransaction gets a transaction by ID.

If the transaction is not found in the access node cache, the request is forwarded to a collection node.

Currently, only transactions within the current epoch can be queried.

```
proto
rpc GetTransaction (GetTransactionRequest) returns (TransactionResponse)
```

Request

GetTransactionRequest contains the ID of the transaction that is being queried.

```
proto
message GetTransactionRequest {
    bytes id = 1;
    bytes blockid = 2;
    bytes collectionid = 3;
    entities.EventEncodingVersion eventencodingversion = 4;
}
```

Response

TransactionResponse contains the basic information about a transaction, but does not include post-execution results.

```
proto
message TransactionResponse {
    entities.Transaction transaction = 1;
    entities.Metadata metadata = 2;
}
```

GetTransactionsByBlockID

GetTransactionsByBlockID gets all the transactions for a specified block.

```
proto
rpc GetTransactionsByBlockID(GetTransactionsByBlockIDRequest) returns
(TransactionsResponse);
```

Request

```
proto
message GetTransactionsByBlockIDRequest {
    bytes blockid = 1;
    entities.EventEncodingVersion eventencodingversion = 2;
}
```

Response

```
proto
message TransactionsResponse {
    repeated entities.Transaction transactions = 1;
    entities.Metadata metadata = 2;
}
```

GetTransactionResult

GetTransactionResult gets the execution result of a transaction.

```
proto
rpc GetTransactionResult (GetTransactionRequest) returns
(TransactionResultResponse)
```

Request

```
proto
message GetTransactionRequest {
    bytes id = 1;
    bytes blockid = 2;
    bytes collectionid = 3;
    entities.EventEncodingVersion eventencodingversion = 4;
}
```

Response

```
proto
message TransactionResultResponse {
    entities.TransactionStatus status = 1;
    uint32 statuscode = 2;
    string errormessage = 3;
    repeated entities.Event events = 4;
    bytes blockid = 5;
    bytes transactionid = 6;
    bytes collectionid = 7;
    uint64 blockheight = 8;
    entities.Metadata metadata = 9;
    uint64 computationusage = 10;
}
```

GetTransactionResultByIndex

GetTransactionResultByIndex gets a transaction's result at a specified block and index.

```
proto
rpc GetTransactionResultByIndex(GetTransactionByIndexRequest) returns
(TransactionResultResponse);
```

Request

```
proto
message GetTransactionByIndexRequest {
    bytes blockid = 1;
    uint32 index = 2;
    entities.EventEncodingVersion eventencodingversion = 3;
}
```

Response

```
proto
message TransactionResultResponse {
    entities.TransactionStatus status = 1;
    uint32 statuscode = 2;
    string errormessage = 3;
    repeated entities.Event events = 4;
    bytes blockid = 5;
    bytes transactionid = 6;
    bytes collectionid = 7;
    uint64 blockheight = 8;
    entities.Metadata metadata = 9;
    uint64 computationusage = 10;
}
```

GetTransactionResultsByBlockID

GetTransactionResultsByBlockID gets all the transaction results for a specified block.

```
proto
rpc GetTransactionResultsByBlockID(GetTransactionsByBlockIDRequest)
returns (TransactionResultsResponse);
```

Request

```
proto
message GetTransactionsByBlockIDRequest {
    bytes blockid = 1;
    entities.EventEncodingVersion eventencodingversion = 2;
```

```
}
```

Response

```
proto
message TransactionResultsResponse {
    repeated TransactionResultResponse transactionresults = 1;
    entities.Metadata metadata = 2;
}
```

GetSystemTransaction

GetSystemTransaction gets the system transaction for a block.

```
proto
rpc GetSystemTransaction(GetSystemTransactionRequest) returns
(TransactionResponse);
```

Request

```
proto
message GetSystemTransactionRequest {
    bytes blockid = 1;
}
```

Response

```
proto
message TransactionResponse {
    entities.Transaction transaction = 1;
    entities.Metadata metadata = 2;
}
```

GetSystemTransactionResult

GetSystemTransactionResult gets the system transaction result for a block.

```
proto
rpc GetSystemTransactionResult(GetSystemTransactionResultRequest) returns
(TransactionResultResponse);
```

Request

```
proto
message GetSystemTransactionResultRequest {
    bytes blockid = 1;
    entities.EventEncodingVersion eventencodingversion = 2;
```

```
}
```

Response

```
proto
message TransactionResultResponse {
    entities.TransactionStatus status = 1;
    uint32 statuscode = 2;
    string errormessage = 3;
    repeated entities.Event events = 4;
    bytes blockid = 5;
    bytes transactionid = 6;
    bytes collectionid = 7;
    uint64 blockheight = 8;
    entities.Metadata metadata = 9;
    uint64 computationusage = 10;
}
```

```
---
```

Accounts

GetAccount

GetAccount gets an account by address at the latest sealed block.

⚠ Warning: this function is deprecated. It behaves identically to GetAccountAtLatestBlock and will be removed in a future version.

```
proto
rpc GetAccount(GetAccountRequest) returns (GetAccountResponse)
```

Request

```
proto
message GetAccountRequest {
    bytes address = 1;
}
```

Response

```
proto
message GetAccountResponse {
    entities.Account account = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountAtLatestBlock

GetAccountAtLatestBlock gets an account by address.

The access node queries an execution node for the account details, which are stored as part of the sealed execution state.

```
proto
rpc GetAccountAtLatestBlock(GetAccountAtLatestBlockRequest) returns
(AccountResponse)
```

Request

```
proto
message GetAccountAtLatestBlockRequest {
    bytes address = 1;
}
```

Response

```
proto
message AccountResponse {
    entities.Account account = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountAtBlockHeight

GetAccountAtBlockHeight gets an account by address at the given block height.

The access node queries an execution node for the account details, which are stored as part of the execution state.

```
proto
rpc GetAccountAtBlockHeight(GetAccountAtBlockHeightRequest) returns
(AccountResponse)
```

Request

```
proto
message GetAccountAtBlockHeightRequest {
    bytes address = 1;
    uint64 blockheight = 2;
}
```

Response

```
proto
message AccountResponse {
    entities.Account account = 1;
```

```
    entities.Metadata metadata = 2;
}
```

GetAccountBalanceAtLatestBlock

GetAccountBalanceAtLatestBlock gets an account's balance by address from the latest sealed block.

```
proto
rpc GetAccountBalanceAtLatestBlock(GetAccountBalanceAtLatestBlockRequest)
returns (AccountBalanceResponse);
```

Request

```
proto
message GetAccountBalanceAtLatestBlockRequest {
    bytes address = 1
}
```

Response

```
proto
message AccountBalanceResponse {
    uint64 balance = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountBalanceAtBlockHeight

GetAccountBalanceAtBlockHeight gets an account's balance by address at the given block height.

```
proto
rpc GetAccountBalanceAtBlockHeight(GetAccountBalanceAtBlockHeightRequest)
returns (AccountBalanceResponse);
```

Request

```
proto
message GetAccountBalanceAtBlockHeightRequest {
    bytes address = 1;
    uint64 blockheight = 2;
}
```

Response

```
proto
message AccountBalanceResponse {
```

```
    uint64 balance = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountKeyAtLatestBlock

GetAccountKeyAtLatestBlock gets an account's public key by address and key index from the latest sealed block.

```
proto
rpc GetAccountKeyAtLatestBlock(GetAccountKeyAtLatestBlockRequest) returns
(AccountKeyResponse);
```

Request

```
proto
message GetAccountKeyAtLatestBlockRequest {
    // address of account
    bytes address = 1;
    // index of key to return
    uint32 index = 2;
}
```

Response

```
proto
message AccountKeyResponse {
    entities.AccountKey accountkey = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountKeyAtBlockHeight

GetAccountKeyAtBlockHeight gets an account's public key by address and key index at the given block height.

```
proto
rpc GetAccountKeyAtBlockHeight(GetAccountKeyAtBlockHeightRequest) returns
(AccountKeyResponse);
```

Request

```
proto
message GetAccountKeyAtBlockHeightRequest {
    // address of account
    bytes address = 1;
    // height of the block
    uint64 blockheight = 2;
    // index of key to return
}
```

```
    uint32 index = 3;
}
```

Response

```
proto
message AccountKeyResponse {
    entities.AccountKey accountkey = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountKeysAtLatestBlock

GetAccountKeysAtLatestBlock gets an account's public keys by address from the latest sealed block.

```
proto
rpc GetAccountKeysAtLatestBlock (GetAccountKeysAtLatestBlockRequest)
returns (AccountKeysResponse);
```

Request

```
proto
message GetAccountKeysAtLatestBlockRequest {
    // address of account
    bytes address = 1;
}
```

Response

```
proto
message AccountKeysResponse {
    repeated entities.AccountKey accountkeys = 1;
    entities.Metadata metadata = 2;
}
```

GetAccountKeysAtBlockHeight

GetAccountKeysAtBlockHeight gets an account's public keys by address at the given block height.

```
proto
rpc GetAccountKeysAtBlockHeight (GetAccountKeysAtBlockHeightRequest)
returns (AccountKeysResponse);
```

Request

```
proto
```

```
message GetAccountKeysAtBlockHeightRequest {
    // address of account
    bytes address = 1;
    uint64 blockheight = 2;
}
```

Response

```
proto
message AccountKeysResponse {
    repeated entities.AccountKey accountkeys = 1;
    entities.Metadata metadata = 2;
}
```

##

Scripts

ExecuteScriptAtLatestBlock

ExecuteScriptAtLatestBlock executes a read-only Cadence script against the latest sealed execution state.

This method can be used to read execution state from the blockchain. The script is executed on an execution node and the return value is encoded using the JSON-Cadence data interchange format.

```
proto
rpc ExecuteScriptAtLatestBlock (ExecuteScriptAtLatestBlockRequest)
returns (ExecuteScriptResponse)
```

This method is a shortcut for the following:

```
header = GetLatestBlockHeader()
value = ExecuteScriptAtBlockID(header.ID, script)
```

Request

```
proto
message ExecuteScriptAtLatestBlockRequest {
    bytes script = 1;
    repeated bytes arguments = 2;
}
```

Response

```
proto
message ExecuteScriptResponse {
```

```
    bytes value = 1;
    entities.Metadata metadata = 2;
    uint64 computationusage = 3;
}
```

ExecuteScriptAtBlockID

`ExecuteScriptAtBlockID` executes a ready-only Cadence script against the execution state at the block with the given ID.

This method can be used to read account state from the blockchain. The script is executed on an execution node and the return value is encoded using the JSON-Cadence data interchange format.

```
proto
rpc ExecuteScriptAtBlockID (ExecuteScriptAtBlockIDRequest) returns
(ExecuteScriptResponse)
```

Request

```
proto
message ExecuteScriptAtBlockIDRequest {
    bytes blockid = 1;
    bytes script = 2;
    repeated bytes arguments = 3;
}
```

Response

```
proto
message ExecuteScriptResponse {
    bytes value = 1;
    entities.Metadata metadata = 2;
    uint64 computationusage = 3;
}
```

ExecuteScriptAtBlockHeight

`ExecuteScriptAtBlockHeight` executes a ready-only Cadence script against the execution state at the given block height.

This method can be used to read account state from the blockchain. The script is executed on an execution node and the return value is encoded using the JSON-Cadence data interchange format.

```
proto
rpc ExecuteScriptAtBlockHeight (ExecuteScriptAtBlockHeightRequest)
returns (ExecuteScriptResponse)
```

Request

```
proto
message ExecuteScriptAtBlockHeightRequest {
    uint64 blockheight = 1;
    bytes script = 2;
    repeated bytes arguments = 3;
}
```

Response

```
proto
message ExecuteScriptResponse {
    bytes value = 1;
    entities.Metadata metadata = 2;
    uint64 computationusage = 3;
}
```

Events

The following methods can be used to query for on-chain events.

`GetEventsForHeightRange`

`GetEventsForHeightRange` retrieves events emitted within the specified block range.

```
proto
rpc GetEventsForHeightRange (GetEventsForHeightRangeRequest) returns
(GetEventsForHeightRangeResponse)
```

Events can be requested for a specific sealed block range via the `startheight` and `endheight` (inclusive) fields and further filtered by event type via the `type` field.

If `startheight` is greater than the current sealed chain height, then this method will return an error.

If `endheight` is greater than the current sealed chain height, then this method will return events up to and including the latest sealed block.

The event results are grouped by block, with each group specifying a block ID, height and block timestamp.

Event types are name-spaced with the address of the account and contract in which they are declared.

Request

```
proto
message GetEventsForHeightRangeRequest {
    string type
    uint64 startheight = 2;
    uint64 endheight = 3;
    entities.EventEncodingVersion eventencodingversion = 4;
}
```

Response

```
proto
message EventsResponse {
    message Result {
        bytes blockid = 1;
        uint64 blockheight = 2;
        repeated entities.Event events = 3;
        google.protobuf.Timestamp blocktimestamp = 4;
    }
    repeated Result results = 1;
    entities.Metadata metadata = 2;
}
```

GetEventsForBlockIDs

GetEventsForBlockIDs retrieves events for the specified block IDs and event type.

```
proto
rpc GetEventsForBlockIDs(GetEventsForBlockIDsRequest) returns
(GetEventsForBlockIDsResponse)
```

Events can be requested for a list of block IDs via the blockids field and further filtered by event type via the type field.

The event results are grouped by block, with each group specifying a block ID, height and block timestamp.

Request

```
proto
message GetEventsForBlockIDsRequest {
    string type = 1;
    repeated bytes blockids = 2;
    entities.EventEncodingVersion eventencodingversion = 3;
}
```

Response

```
proto
message EventsResponse {
```

```
message Result {
    bytes blockid = 1;
    uint64 blockheight = 2;
    repeated entities.Event events = 3;
    google.protobuf.Timestamp blocktimestamp = 4;
}
repeated Result results = 1;
entities.Metadata metadata = 2;
}
```

Network Parameters

Network parameters provide information about the Flow network. Currently, it only includes the chain ID.
The following method can be used to query for network parameters.

GetNetworkParameters

GetNetworkParameters retrieves the network parameters.

```
proto
rpc GetNetworkParameters (GetNetworkParametersRequest) returns
(GetNetworkParametersResponse)
```

Request

```
proto
message GetNetworkParametersRequest {}
```

Response

```
proto
message GetNetworkParametersResponse {
    string chainid = 1;
}
```

Field	Description
----- -----	-----
chainid Chain ID helps identify the Flow network. It can be one of flow-mainnet, flow-testnet or flow-emulator	

GetNodeVersionInfo

GetNodeVersionInfo gets information about a node's current versions.

```
proto
rpc GetNodeVersionInfo (GetNodeVersionInfoRequest) returns
(GetNodeVersionInfoResponse);
```

Request

```
proto
message GetNodeVersionInfoRequest {}
```

Response

```
proto
message GetNodeVersionInfoResponse {
    entities.NodeVersionInfo info = 1;
}
```

Protocol state snapshot

The following method can be used to query the latest protocol state snapshot.

GetLatestProtocolStateSnapshot

GetLatestProtocolStateSnapshot retrieves the latest Protocol state snapshot serialized as a byte array.

It is used by Flow nodes joining the network to bootstrap a space-efficient local state.

```
proto
rpc GetLatestProtocolStateSnapshot
(GetLatestProtocolStateSnapshotRequest) returns
(ProtocolStateSnapshotResponse);
```

Request

```
proto
message GetLatestProtocolStateSnapshotRequest {}
```

Response

```
proto
message ProtocolStateSnapshotResponse {
    bytes serializedSnapshot = 1;
    entities.Metadata metadata = 2;
}
```

GetProtocolStateSnapshotByBlockID

GetProtocolStateSnapshotByBlockID retrieves the latest sealed protocol state snapshot by block ID.

Used by Flow nodes joining the network to bootstrap a space-efficient local state.

```
proto
rpc
GetProtocolStateSnapshotByBlockID(GetProtocolStateSnapshotByBlockIDRequest) returns (ProtocolStateSnapshotResponse);
```

Request

```
proto
message GetProtocolStateSnapshotByBlockIDRequest {
    bytes blockid = 1;
}
```

Response

```
proto
message ProtocolStateSnapshotResponse {
    bytes serializedSnapshot = 1;
    entities.Metadata metadata = 2;
}
```

GetProtocolStateSnapshotByHeight

GetProtocolStateSnapshotByHeight retrieves the latest sealed protocol state snapshot by block height.

Used by Flow nodes joining the network to bootstrap a space-efficient local state.

```
proto
rpc
GetProtocolStateSnapshotByHeight(GetProtocolStateSnapshotByHeightRequest) returns (ProtocolStateSnapshotResponse);
```

Request

```
proto
message GetProtocolStateSnapshotByHeightRequest {
    uint64 blockheight = 1;
}
```

Response

```
proto
message ProtocolStateSnapshotResponse {
    bytes serializedSnapshot = 1;
    entities.Metadata metadata = 2;
}
```

Execution results

The following method can be used to query the for execution results for a given block.

GetExecutionResultForBlockID

GetExecutionResultForBlockID retrieves execution result for given block. It is different from Transaction Results, and contain data about chunks/collection level execution results rather than particular transactions. Particularly, it contains EventsCollection hash for every chunk which can be used to verify the events for a block.

```
proto
rpc GetExecutionResultForBlockID(GetExecutionResultForBlockIDRequest)
returns (ExecutionResultForBlockIDResponse);
```

Request

```
proto
message GetExecutionResultForBlockIDRequest {
    bytes blockid = 1;
}
```

Response

```
proto
message ExecutionResultForBlockIDResponse {
    flow.ExecutionResult executionresult = 1;
    entities.Metadata metadata = 2;
}
```

GetExecutionResultByID

GetExecutionResultByID returns Execution Result by its ID. It is different from Transaction Results, and contain data about chunks/collection level execution results rather than particular transactions. Particularly, it contains EventsCollection hash for every chunk which can be used to verify the events for a block.

```
proto
```

```
rpc GetExecutionResultByID(GetExecutionResultByIDRequest) returns  
(ExecutionResultByIDResponse);
```

Request

```
proto  
message GetExecutionResultByIDRequest {  
    bytes id = 1;  
}
```

Response

```
proto  
message ExecutionResultByIDResponse {  
    flow.ExecutionResult executionresult = 1;  
    entities.Metadata metadata = 2;  
}
```

Entities

Below are in-depth descriptions of each of the data entities returned or accepted by the Access API.

Block

```
proto  
message Block {  
    bytes id = 1;  
    bytes parentid = 2;  
    uint64 height = 3;  
    google.protobuf.Timestamp timestamp = 4;  
    repeated CollectionGuarantee collectionguarantees = 5;  
    repeated BlockSeal blockseals = 6;  
    repeated bytes signatures = 7;  
    repeated ExecutionReceiptMeta executionreceiptmetaList = 8;  
    repeated ExecutionResult executionresultlist = 9;  
    BlockHeader blockheader = 10;  
    bytes protocolstateid = 11;  
}
```

Field	Description
id	SHA3-256 hash of the entire block payload
parentid	Parent block's ID
height	Block's height in the chain
timestamp	Timestamp of the block creation
collectionguarantees	Collection guarantees associated with the block
blockseals	Block seals (e.g., Merkle root)
signatures	Signatures of the block payload
executionreceiptmetaList	Execution receipt meta data for the block
executionresultlist	Execution results for the block
blockheader	Block header information
protocolstateid	Protocol state ID associated with the block

height	Height of the block in the chain
parentid	ID of the previous block in the chain
timestamp	Timestamp of when the proposer claims it constructed the block. NOTE: It is included by the proposer, there are no guarantees on how much the time stamp can deviate from the true time the block was published. Consider observing blocks' status changes yourself to get a more reliable value
collectionguarantees	List of collection guarantees
blockseals	List of block seals
signatures	BLS signatures of consensus nodes
executionreceiptmetaList	List of execution-receipt-meta
executionresultlist	List of execution results
blockheader	A summary of a block
protocolstateid	The root hash of protocol state.

The detailed semantics of block formation are covered in the block formation guide.

Block Header

A block header is a summary of a block and contains only the block ID, height, and parent block ID.

```
proto
message BlockHeader {
    bytes id = 1;
    bytes parentid = 2;
    uint64 height = 3;
    google.protobuf.Timestamp timestamp = 4;
    bytes payloadhash = 5;
    uint64 view = 6;
    repeated bytes parentvoterids = 7;
    bytes parentvotersigdata = 8;
    bytes proposerid = 9;
    bytes proposersigdata = 10;
    string chainid = 11;
    bytes parentvoterindices = 12;
    TimeoutCertificate lastviewtc = 13;
    uint64 parentview = 14;
}
```

Field	Description
-------	-------------

----- -----	
id	SHA3-256 hash of the entire block payload
parentid	ID of the previous block in the chain
height	Height of the block in the chain
timestamp	The time at which this block was proposed
payloadhash	A hash of the payload of this block
view	View number during which this block was proposed.
parentvoterids	An array that represents all the voters ids for the parent block
parentvotersigdata	An aggregated signature over the parent block
chainid	Chain ID helps identify the Flow network. It can be one of flow-mainnet, flow-testnet or flow-emulator
parentvoterindices	A bitvector that represents all the voters for the parent block
lastviewtc	A timeout certificate for previous view, it can be nil. It has to be present if previous round ended with timeout
parentview	A number at which parent block was proposed

Block Seal

A block seal is an attestation that the execution result of a specific block has been verified and approved by a quorum of verification nodes.

```
proto
message BlockSeal {
    bytes blockid = 1;
    bytes executionreceiptid = 2;
    repeated bytes executionreceiptsignatures = 3;
    repeated bytes resultapprovalssignatures = 4;
}
```

Field	Description
----- -----	
blockid	ID of the block being sealed
executionreceiptid	ID execution receipt being sealed
executionreceiptsignatures	BLS signatures of verification nodes on the execution receipt contents
resultapprovalssignatures	BLS signatures of verification nodes on the result approval contents

Block Status

```
proto
enum BlockStatus {
    UNKNOWN = 0;
    FINALIZED = 1;
    SEALED = 2;
}
```

Value	Description
UNKNOWN	The block status is not known
FINALIZED	The consensus nodes have finalized the block
SEALED	The verification nodes have verified the block

Collection

A collection is a batch of transactions that have been included in a block. Collections are used to improve consensus throughput by increasing the number of transactions per block.

```
proto
message Collection {
    bytes id = 1;
    repeated bytes transactionids = 2;
}
```

Field	Description
id	SHA3-256 hash of the collection contents
transactionids	Ordered list of transaction IDs in the collection

Collection Guarantee

A collection guarantee is a signed attestation that specifies the collection nodes that have guaranteed to store and respond to queries about a collection.

```
proto
message CollectionGuarantee {
    bytes collectionid = 1;
    repeated bytes signatures = 2;
    bytes referenceblockid = 3;
    bytes signature = 4;
    repeated bytes signerids = 5; // deprecated!! value will be empty.
    replaced by signerindices
    bytes signerindices = 6;
}
```

Field	Description
collectionid	SHA3-256 hash of the collection contents
signatures	BLS signatures of the collection nodes guaranteeing the collection
referenceblockid	Defines expiry of the collection
signature	Guarantor signatures
signerids	An array that represents all the signer ids
signerindices	Encoded indices of the signers

Transaction

A transaction represents a unit of computation that is submitted to the Flow network.

```
proto
message Transaction {
    bytes script = 1;
    repeated bytes arguments = 2;
    bytes referenceblockid = 3;
    uint64 gaslimit = 4;
    ProposalKey proposalkey = 5;
    bytes payer = 6;
    repeated bytes authorizers = 7;
    repeated Signature payloadsignatures = 8;
    repeated Signature envelopesignatures = 9;
}

message TransactionProposalKey {
    bytes address = 1;
    uint32 keyid = 2;
    uint64 sequencenumber = 3;
}

message TransactionSignature {
    bytes address = 1;
    uint32 keyid = 2;
    bytes signature = 3;
}
```

Field	Description

script	Raw source code for a Cadence script, encoded as UTF-8 bytes
arguments	Arguments passed to the Cadence script, encoded as JSON-Cadence bytes
referenceblockid	Block ID used to determine transaction expiry
proposalkey	Account key used to propose the transaction
payer	Address of the payer account
authorizers	Addresses of the transaction authorizers
signatures	Signatures from all signer accounts

The detailed semantics of transaction creation, signing and submission are covered in the [transaction submission guide](#).

Proposal Key

The proposal key is used to specify a sequence number for the transaction. Sequence numbers are covered in more detail [here](#).

Field	Description
----- -----	
address	Address of proposer account
keyid	ID of proposal key on the proposal account
sequencenumber	Sequence number for the proposal key

Transaction Signature

Field	Description
----- -----	
address	Address of the account for this signature
keyid	ID of the account key
signature	Raw signature byte data

Transaction Status

```
proto
enum TransactionStatus {
    UNKNOWN = 0;
    PENDING = 1;
    FINALIZED = 2;
    EXECUTED = 3;
    SEALED = 4;
    EXPIRED = 5;
```

```
}
```

Value	Description
UNKNOWN	The transaction status is not known.
PENDING	The transaction has been received by a collector but not yet finalized in a block.
FINALIZED	The consensus nodes have finalized the block that the transaction is included in
EXECUTED	The execution nodes have produced a result for the transaction
SEALED	The verification nodes have verified the transaction (the block in which the transaction is) and the seal is included in the latest block
EXPIRED	The transaction was submitted past its expiration block height.

Account

An account is a user's identity on Flow. It contains a unique address, a balance, a list of public keys and the code that has been deployed to the account.

```
proto
message Account {
    bytes address = 1;
    uint64 balance = 2;
    bytes code = 3;
    repeated AccountKey keys = 4;
    map<string, bytes> contracts = 5;
}
```

Field	Description
address	A unique account identifier
balance	The account balance
code	The code deployed to this account (deprecated, use contracts instead)
keys	A list of keys configured on this account

contracts A map of contracts or contract interfaces deployed on this account	
--------------------------------------------------------------------------------	--

The code and contracts fields contain the raw Cadence source code, encoded as UTF-8 bytes.

More information on accounts can be found [here](#).

Account Key

An account key is a reference to a public key associated with a Flow account. Accounts can be configured with zero or more public keys, each of which can be used for signature verification when authorizing a transaction.

```
proto
message AccountKey {
    uint32 index = 1;
    bytes publickey = 2;
    uint32 signalgo = 3;
    uint32 hashalgo = 4;
    uint32 weight = 5;
    uint32 sequencenumber = 6;
    bool revoked = 7;
}
```

Field	Description
----- -----	-----
id	Index of the key within the account, used as a unique identifier
publickey	Public key encoded as bytes
signalgo	Signature algorithm
hashalgo	Hash algorithm
weight	Weight assigned to the key
sequencenumber	Sequence number for the key
revoked	Flag indicating whether or not the key has been revoked

More information on account keys, key weights and sequence numbers can be found [here](#).

Event

An event is emitted as the result of a transaction execution. Events are either user-defined events originating from a Cadence smart contract, or built-in Flow system events.

```
proto
message Event {
    string type = 1;
```

```

    bytes transactionid = 2;
    uint32 transactionindex = 3;
    uint32 eventindex = 4;
    bytes payload = 5;
}

```

Field	Description
type	Fully-qualified unique type identifier for the event
transactionid	ID of the transaction the event was emitted from
transactionindex	Zero-based index of the transaction within the block
eventindex	Zero-based index of the event within the transaction
payload	Event fields encoded as JSON-Cadence values

Execution Result

Execution result for a particular block.

```

proto
message ExecutionResult {
    bytes previousresultid = 1;
    bytes blockid = 2;
    repeated Chunk chunks = 3;
    repeated ServiceEvent serviceevents = 4;
}

```

Field	Description
previousresultid	Identifier of parent block execution result
blockid	ID of the block this execution result corresponds to
chunks	Zero or more chunks
serviceevents	Zero or more service events

Execution Receipt Meta

ExecutionReceiptMeta contains the fields from the Execution Receipts that vary from one executor to another

```

proto
message ExecutionReceiptMeta {

```

```

    bytes executorid = 1;
    bytes resultid = 2;
    repeated bytes spocks = 3;
    bytes executorsignature = 4;
}

```

Field	Description
executorid	Identifier of the executor node
resultid	Identifier of block execution result
spocks	SPOCK
executorsignature	Signature of the executor

Chunk

Chunk described execution information for given collection in a block

```

proto
message Chunk {
    uint32 CollectionIndex = 1;
    bytes startstate = 2;
    bytes eventcollection = 3;
    bytes blockid = 4;
    uint64 totalcomputationused = 5;
    uint32 numberoftxns = 6;
    uint64 index = 7;
    bytes endstate = 8;
    bytes executiondataid = 9;
    bytes statedeltacommitment = 10;
}

```

Field	Description
CollectionIndex	Identifier of a collection
startstate	State commitment at start of the chunk
eventcollection	Hash of events emitted by transactions in this chunk
blockid	Identifier of a block
totalcomputationused	Total computation used by transactions in this chunk
numberoftxns	Number of transactions in a chunk
index	Index of chunk inside a block (zero-based)
endstate	State commitment after executing chunk

executiondataid	Identifier of a execution data
statedeltacommitment	A commitment over sorted list of register changes

Service Event

Special type of events emitted in system chunk used for controlling Flow system.

```
proto
message ServiceEvent {
    string type = 1;
    bytes payload = 2;
}
```

Field	Description
----- -----	-----
type	Type of an event
payload	JSON-serialized content of an event

Subscriptions

SubscribeEvents

SubscribeEvents streams events for all blocks starting at the requested start block, up until the latest available block. Once the latest is reached, the stream will remain open and responses are sent for each new block as it becomes available.

Events within each block are filtered by the provided EventFilter, and only those events that match the filter are returned. If no filter is provided, all events are returned.

Responses are returned for each block containing at least one event that matches the filter. Additionally, heartbeat responses (SubscribeEventsResponse with no events) are returned periodically to allow clients to track which blocks were searched. Clients can use this information to determine which block to start from when reconnecting.

```
proto
rpc SubscribeEvents(SubscribeEventsRequest) returns (stream
SubscribeEventsResponse)
```

Request

```
proto
message SubscribeEventsRequest {
    bytes startblockid = 1;
    uint64 startblockheight = 2;
```

```

EventFilter filter = 3;
uint64 heartbeatinterval = 4;
entities.EventEncodingVersion eventencodingversion = 5;
}

```

Field	Description
----- -----	-----
----- -----	-----
startblockid The first block to search for events. Only one of startblockid and startblockheight may be provided, otherwise an InvalidArgument error is returned. If neither are provided, the latest sealed block is used	
startblockheight Block height of the first block to search for events. Only one of startblockid and startblockheight may be provided, otherwise an InvalidArgument error is returned. If neither are provided, the latest sealed block is used	
filter Filter to apply to events for each block searched. If no filter is provided, all events are returned	
heartbeatinterval Interval in block heights at which the server should return a heartbeat message to the client	
eventencodingversion Preferred event encoding version of the block events payload. Possible variants: CCF, JSON-CDC	

Response

```

proto
message SubscribeEventsResponse {
    bytes blockid = 1;
    uint64 blockheight = 2;
    repeated entities.Event events = 3;
    google.protobuf.Timestamp blocktimestamp = 4;
    uint64 messageindex = 5;
}

```

SubscribeExecutionData

SubscribeExecutionData streams execution data for all blocks starting at the requested start block, up until the latest available block. Once the latest is reached, the stream will remain open and responses are sent for each new execution data as it becomes available.

```

proto
rpc SubscribeExecutionData(SubscribeExecutionDataRequest) returns (stream
SubscribeExecutionDataResponse)

```

Request

```
proto
message SubscribeExecutionDataRequest {
    bytes startblockid = 1;
    uint64 startblockheight = 2;
    entities.EventEncodingVersion eventencodingversion = 3;
}
```

Field	Description
startblockid	The first block to get execution data for. Only one of startblockid and startblockheight may be provided, otherwise an InvalidArgument error is returned. If neither are provided, the latest sealed block is used
startblockheight	Block height of the first block to get execution data for. Only one of startblockid and startblockheight may be provided, otherwise an InvalidArgument error is returned. If neither are provided, the latest sealed block is used
eventencodingversion	Preferred event encoding version of the block events payload. Possible variants: CCF, JSON-CDC

Response

```
proto
message SubscribeExecutionDataResponse {
    uint64 blockheight = 1;
    entities.BlockExecutionData blockexecutiondata = 2;
    google.protobuf.Timestamp blocktimestamp = 3;
}
```

Execution data

EventFilter

EventFilter defines the filter to apply to block events. Filters are applied as an OR operation, i.e. any event matching any of the filters is returned.

If no filters are provided, all events are returned. If there are any invalid filters, the API will return an InvalidArgument error.

```
proto
message EventFilter {
    repeated string eventtype = 1;
    repeated string contract = 2;
    repeated string address = 3;
}
```

Field	Description
eventtype	A list of full event types to include. Event types have 2 formats: Protocol events: flow.[event name] Smart contract events: A.[contract address].[contract name].[event name]
contract	A list of contracts who's events should be included. Contracts have the following name formats: Protocol events: flow Smart contract events: A.[contract address].[contract name] This filter matches on the full contract including its address, not just the contract's name
address	A list of addresses who's events should be included. Addresses must be Flow account addresses in hex format and valid for the network the node is connected to. i.e. only a mainnet address is valid for a mainnet node. Addresses may optionally include the 0x prefix

Execution data streaming API

Execution Data API

The ExecutionDataAPI provides access to block execution data over gRPC, including transactions, events, and register data (account state). It's an optional API, which makes use of the Execution Sync protocol to trustlessly download data from peers on the network.

execution data protobuf file

> The API is disabled by default. To enable it, specify a listener address with the cli flag --state-stream-addr.

<aside>

i Currently, the api must be started on a separate port from the regular gRPC endpoint. There is work underway to add support for using the same port.

</aside>

Below is a list of the available CLI flags to control the behavior of the API

Flag	Type	Description

```
|-----|-----|-----|
|-----|-----|-----|
|-----|-----|-----|
| state-stream-addr           | string    | Listener address for API.
e.g. 0.0.0.0:9003. If no value is provided, the API is disabled. Default
is disabled.
|
| execution-data-cache-size   | uint32    | Number of block execution
data objects to store in the cache. Default is 100.
|
| state-stream-global-max-streams | uint32    | Global maximum number of
concurrent streams. Default is 1000.
|
| state-stream-max-message-size | uint       | Maximum size for a gRPC
response message containing block execution data. Default is 2010241024
(20MB).
|
| state-stream-event-filter-limits | string    | Event filter limits for
ExecutionData SubscribeEvents API. These define the max number of filters
for each type. e.g. EventTypes=100, Addresses=20, Contracts=50. Default is
1000 for each.
|
| state-stream-send-timeout     | duration   | Maximum wait before
timing out while sending a response to a streaming client. Default is
30s.
|
| state-stream-send-buffer-size | uint       | Maximum number of unsent
responses to buffer for a stream. Default is 10.
|
| state-stream-response-limit   | float64   | Max number of responses
per second to send over streaming endpoints. This effectively applies a
rate limit to responses to help manage resources consumed by each client.
This is mostly used when clients are querying data past data. e.g. 3 or
0.5. Default is 0 which means no limit. |
```

<aside>

i This API provides access to Execution Data, which can be very large (100s of MB) for a given block. Given the large amount of data, operators should consider their expected usage patterns and tune the available settings to limit the resources a single client can use. It may also be useful to use other means of managing traffic, such as reverse proxies or QoS tools.

</aside>

```
# access-http-api.md:
```

```
---  
redirect: /http-api  
title: Access HTTP API ↗  
sidebarposition: 2  
---
```

Access HTTP API

Go to HTTP API

```
<meta http-equiv="refresh" content="0; url=/http-api" />
```

```
# observer-node.md:
```

```
---
```

```
title: Light Node a.k.a Observer Node
```

```
sidebarlabel: Light Node Setup
```

```
sidebarposition: 1
```

```
---
```

A light node also known as the observer node is similar to an access node and provides a locally accessible, continuously updated, verified copy of the block data. It serves the gRPC Access API but unlike an access node, an light node does not need to be staked, and anyone can run it without being added to the approved list of nodes.

The light node bootstraps by connecting to an access node and becoming part of the public network comprised of access nodes and other light nodes. It then continuously receives blocks, which the consensus nodes are adding to the chain, either directly from access nodes or from other light nodes that are part of the public network. However, it makes no trust assumption of the upstream access node or the light node which is providing the block and locally verifies that the blocks that are received are the correct extension of the chain e.g. after receiving valid blocks A, B and C when it receives block D, it verifies that block D is indeed signed by the consensus nodes and is a valid next block. The received block data is indexed and made available via the Access API. For Collection, Transactions and Account queries, it delegates those requests to the upstream access node. Similarly, transactions and scripts sent to a light node are also forwarded to the upstream access node. Future versions of the light node will be able to serve this data locally as well.

Since the light node is not staked, it does not produce or execute blocks but instead serves as an unstaked access node that can be easily run on any consumer-grade computer which has enough disk space.

!Observer nodes

Who should run a light node?

The light node provides an alternative to running an access node. Hence, it is ideal for Dapps that need access to the latest block data locally on a machine they control. Examples include a wallet application that needs to track the latest block ID and height. Alternatively, access node operators that want to scale their access node endpoints geographically can spin up light nodes in different regions, which can talk to their staked access node and to each other.

Running an light node

Hardware

In general, any consumer-grade computer with a decent network connection and sufficient disk space should be able to run a light node.

Minimum requirements

- CPU with 2+ cores
- 4 GB RAM minimum
- 300 GB SSD disk
- 10Mbps network connection

Steps to run a light node

> Here is video walk-though of these 4 steps.

Step 1 - Generate the node directory structure

The light node requires the following directory structure,
shell

```
$ tree flowobserver
flowobserver/
└── bootstrap
    ├── network.key (file containing the node private network key)
    └── public-root-information
        └── root-protocol-state-snapshot.json (the genesis data of
            the current spork)
    └── data (directory used by the light node to store block data)
```

Create the parent and the sub-directories

e.g.

shell

```
mkdir -p flowobserver/bootstrap/public-root-information
mkdir flowobserver/data
```

Step 2 - Generate the network key

Like any other Flow node, the light node also needs a networking ECDSA key to talk to the network.

Download the Bootstrapping kit, and generate the networking key.

```
shell
curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-tools.tar
tar -xvf boot-tools.tar
./boot-tools/bootstrap observer-network-key --output-file
./flowobserver/bootstrap/network.key
```

If you are running on a mac, download the boot-tools for mac to generate the key

```
shell
For M1
curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-tools-
m1.tar
For Intel Mac
curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-tools-
intel-mac.tar
```

Step 3 - Download the root-protocol-state-snapshot.json file for the current spork

The root-protocol-state-snapshot.json is generated for each spork and contains the genesis data for that spork. It is published and made available after each spork. The download location is specified here under rootProtocolStateSnapshot and can be downloaded as follows,

For mainnet find the latest spork version from sporks.json and then download the root-protocol-state-snapshot.json and the signature file for it.

```
shell
wget -P ./flowobserver/bootstrap/public-root-information
https://storage.googleapis.com/flow-genesis-bootstrap/mainnet-<spork
version>-execution/public-root-information/root-protocol-state-
snapshot.json
wget -P ./flowobserver/bootstrap/public-root-information
https://storage.googleapis.com/flow-genesis-bootstrap/mainnet-<spork
version>-execution/public-root-information/root-protocol-state-
snapshot.json.asc
```

Similarly, for testnet find the latest spork version from sporks.json and then download the root-protocol-state-snapshot.json and the signature file for it.

```
shell
wget -P ./flowobserver/bootstrap/public-root-information
https://storage.googleapis.com/flow-genesis-bootstrap/testnet-<spork
version>/public-root-information/root-protocol-state-snapshot.json
wget -P ./flowobserver/bootstrap/public-root-information
https://storage.googleapis.com/flow-genesis-bootstrap/testnet-<spork
version>/public-root-information/root-protocol-state-snapshot.json.asc
```

Verify the PGP signature

```
Add the flow-signer@onflow.org public key
shell
gpg --keyserver keys.openpgp.org --search-keys flow-signer@onflow.org

gpg: data source: http://keys.openpgp.org:11371
(1)  Flow Team (Flow Full Observer node snapshot verification master
key) <
```

```
256 bit ECDSA key CB5264F7FD4CDD27, created: 2021-09-15
Keys 1-1 of 1 for "flow-signer@onflow.org". Enter number(s), N)ext, or
Q)uit > 1
```

```
Verify the root-snapshot file
shell
gpg --verify ./flowobserver/bootstrap/public-root-information/root-
protocol-state-snapshot.json.asc

gpg: assuming signed data in 'bootstrap/public-root-information/root-
protocol-state-snapshot.json'
gpg: Signature made Wed Sep 15 11:34:33 2021 PDT
gpg:           using ECDSA key
40CD95717AC463E61EE3B285B718CA310EDB542F
gpg: Good signature from "Flow Team (Flow Full Observer node snapshot
verification master key) <flow-signer@onflow.org>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:           There is no indication that the signature belongs to the
owner.
Primary key fingerprint: 7D23 8D1A E6D3 2A71 8ECD 8611 CB52 64F7 FD4C
DD27
Subkey fingerprint: 40CD 9571 7AC4 63E6 1EE3 B285 B718 CA31 0EDB
542F
```

Alternately, if you don't care about the blocks before the current block, you can request the current root-snapshot file via the Flow CLI.

```
For mainnet
shell
flow snapshot save ./flowobserver/bootstrap/public-root-
information/root-protocol-state-snapshot.json --host
secure.mainnet.nodes.onflow.org:9001 --network-key
28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295144692a58e
be7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae
```

```
For testnet
shell
flow snapshot save ./flowobserver/bootstrap/public-root-information/root-
protocol-state-snapshot.json --host secure.testnet.nodes.onflow.org:9001
--network-key
ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daebf836f7
445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d
```

Step 4 - Start the node

The light node can be run as a docker container

Observer for Flow Mainnet

```
shell
```

```

docker run --rm \
-v $PWD/flowobserver/bootstrap:/bootstrap:ro \
-v $PWD/flowobserver/data:/data:rw \
--name flowobserver \
-p 80:80 \
-p 3569:3569 \
-p 9000:9000 \
-p 9001:9001 \
gcr.io/flow-container-registry/observer:v0.27.2 \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--bind 0.0.0.0:3569 \
--rest-addr=:80 \
--loglevel=error \
--secretsdir=/data/secrets \
--upstream-node-addresses=secure.mainnet.nodes.onflow.org:9001 \
--upstream-node-public-
keys=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e29514469
2a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \
--bootstrap-node-addresses=secure.mainnet.nodes.onflow.org:3570 \
--bootstrap-node-public-
keys=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e29514469
2a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \
--observer-networking-key-path=/bootstrap/network.key

```

Observer for Flow Testnet

```

shell
docker run --rm \
-v $PWD/flowobserver/bootstrap:/bootstrap:ro \
-v $PWD/flowobserver/data:/data:rw \
--name flowobserver \
-p 80:80 \
-p 3569:3569 \
-p 9000:9000 \
-p 9001:9001 \
gcr.io/flow-container-registry/observer:v0.27.2 \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--bind 0.0.0.0:3569 \
--rest-addr=:80 \
--loglevel=error \
--secretsdir=/data/secrets \
--upstream-node-addresses=secure.devnet.nodes.onflow.org:9001 \
--upstream-node-public-
keys=ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daeebf
836f7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d \
--bootstrap-node-addresses=secure.devnet.nodes.onflow.org:3570 \
--bootstrap-node-public-
keys=ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daeebf
836f7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d \
--observer-networking-key-path=/bootstrap/network.key

```

The light node acts as a DHT client and bootstraps from upstream access nodes which run the DHT server.

The upstream bootstrap server is specified using the bootstrap-node-addresses which is the comma-separated list of hostnames of the access nodes.

The bootstrap-node-public-keys is the list of the corresponding networking public key of those nodes.

The light node delegates many of the API calls to the upstream access nodes.

The upstream-node-addresses is the list of access node hostnames to which this light node can delegate to. The list can be different from the bootstrap node list.

The bootstrap-node-public-key is the list of the corresponding networking public key of those nodes.

> In the above docker commands, the Flow community access nodes are being used as the upstream access nodes. However, any other Flow access node that supports a light node can be used

All parameters and their explanation can be found here

☞ The node should now be up and running

You can now query the node for blocks, transaction etc. similar to how you would query an access node.

```
e.g. querying the gRPC API endpoint using Flow CLI  
shell  
flow blocks get latest --host localhost:9000
```

```
e.g. querying the REST API endpoint using curl  
shell  
curl "http://localhost/v1/blocks?height=sealed"
```

The light node, like the other type of Flow nodes, also produces Prometheus metrics that can be used to monitor node health. More on that here

FAQs

Does the light node need to be staked?

No, the light node is not a staked node.

Can any access node be used to bootstrap a light node?

No, only Access nodes which have explicitly turned ON support for light nodes can be used to bootstrap a light node.

The public access nodes that support light nodes are listed below. Apart from these, other public access nodes run by node operators other than the Flow foundation team may choose to support light nodes.

How can an access node turn ON support for light node?

An access node can support a light node by passing in the following two parameters when starting the access node

```
shell  
  --supports-observer=true --public-network-address=0.0.0.0:3570
```

public-network-address is the address the light nodes will connect to.

Are light nodes subject to rate limits?

The light node serves all the Block related queries from its local database. These are not subjected to any rate limits.

However, it proxies all the other requests to the access node and those will be rate limited as per the rate limits defined on that access node.

Flow community access nodes that support connections from light nodes

For mainnet

Access-007:

```
Host: access-007.[current mainnet spork].nodes.onflow.org
```

Public Key:

```
28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295144692a58e  
be7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae
```

Access-008:

```
Host: access-008.[current mainnet spork].nodes.onflow.org
```

Public Key:

```
11742552d21ac93da37ccda09661792977e2ca548a3b26d05f22a51ae1d99b9b75c8a9b3b  
40b38206b38951e98e4d145f0010f8942fd82ddf0fb1d670202264a
```

For testnet

Access-001:

```
Host: access-001.[current devnet spork].nodes.onflow.org
```

Public Key:

```
ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daebf836f7  
445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d
```

Access-003:

```
Host: access-003.[current devnet spork].nodes.onflow.org
```

Public Key:

```
b662102f4184fc1caeb2933cf87bba75cdd37758926584c0ce8a90549bb12ee0f9115111b  
bbb6acc2b889461208533369a91e8321eaf6bcb871a788ddd6bfbf7
```

While the public keys remain the same, the hostnames change each spork to include the spork name. Substitute [current mainnet spork] and [current devnet spork] with the appropriate spork name (e.g. mainnet20). See Past Sporks for the current spork for each network.

```
# byzantine-node-attack-response.md:
```

```
---
```

title: Byzantine Node Attack Response
sidebarlabel: Byzantine Attack Response
description: How to respond to a byzantine node attack on the network
sidebarposition: 3

Flow, like most blockchains, forms an open decentralized peer-to-peer network between all of the nodes on the network. Due to its decentralized nature, there is a potential for nodes to behave maliciously (byzantine) and intentionally try to harm the network. There are a variety of protections within the node software to deal with invalid messages - message signatures, sender authorization, payload validation, etc. These protections guard the network against many types of attacks. However, there could still be a byzantine node that spams other nodes in the network with invalid messages at volumes that are intended to impact node performance. While this will not compromise the security of the network it could impact network liveness.

This guide explains how to detect such a node and what actions you should take as a node operator to deal with such byzantine nodes.

Responding to an attack from a byzantine node requires the following:

1. Immediate action to block network traffic originating from the byzantine node to your node.
2. Raising a governance FLIP to remove the node from the network as described in this FLIP.
3. A service account transaction to set the node weight to 0.

This guide focuses on the first action.

Admin Server

Flow nodes have an admin server which exposes a simple REST API for interacting with the node.
See the README for some useful examples.
It is disabled by default.

Enable the Admin Server

To enable to admin server,

1. Add the following option to the node's CLI flags.

```
--admin-addr=localhost:9002
```

> Note: The port does not have to be 9002. You can choose any free port.

> **⚠** Do NOT expose the port outside the machine and always use
localhost:port

2. Reboot the node to apply the new setting. You can then verify it's working by logging into the machine via ssh and running,

```
curl localhost:9002
```

This should return a json response message as below.

```
{"code":5,"message":"Not Found","details":[]}
```

If you instead get a connection rejected message then it's not configured correctly.

Detecting a Byzantine Node

There are 2 general categories of byzantine attacks:

1. Safety attacks - are attacks where a node attempts to corrupt or modify the state of the blockchain outside of normal protocol rules.
2. Liveness attacks - sometimes called spamming attacks, are when a node attempts to disrupt the network by abusing their access to waste network and node resources. This generally results in degraded performance.

Flow nodes are protected against safety attacks, but liveness attacks are extremely difficult to completely prevent. To close the gap, we rely on coordination between node operators to detect and block abusive nodes.

Metrics

Flow nodes generate a variety of metrics that can be used to measure the node's performance and identify abnormal behavior. Most metrics are only useful in the context of "normal" operation, so it is a good idea to regularly review them to build an understanding of what is "normal".

Metrics to watch:

CPU, memory, network connections, network I/O, file descriptors
networkauthorization - counts the number of unauthorized/invalid
messages received
networkqueue - measures the number of incoming messages
waiting to be processed
networkenginetotal - measures the number of messages
received from the network

There are many other metrics, but these are a good starting point. If you notice any anomalous trends, review the logs for additional context.

Logs

Log events related to suspicious activity are logged with the label "suspicious":true. This is helpful to identify the most relevant logs, but there are legitimate cases when these logs are emitted, so they cannot be used as a definitive indicator of malicious activity. Two examples of expected log messages are:

rejected inbound connection - You may see this error if an operator unstaked their node between sporks, but never shut it down. The node will continue to operate as usual, but peers will not have it in their identity table and will (correctly) reject incoming connections.

middleware does not have subscription for the channel ID indicated in the unicast message received -

This is commonly logged during node startup when receiving messages before all of the components have finished registering their channels with the network layer. It is NOT expected after startup.

The following is an example of a log message indicating an Access node attempted to send a message it is not authorized to send:

```
{  
  "level": "error",  
  "noderole": "collection",  
  "nodeid":  
  "4a6f7264616e20536368616c6d00a875801849f2b5bea9e9d2c9603f00e5d533",  
  "module": "networkslashingconsumer",  
  "peerid": "QmY2kby3xt3ugu2QqJP5w24rP4HSakYgDFpAJy1ifSRkF7",  
  "networkingoffense": "unauthorizedsender",  
  "messagetype": "messages.BlockProposal",  
  "channel": "sync-committee",  
  "protocol": "publish",  
  "suspicious": true,  
  "role": "access",
```

```
    "senderid":  
    "f9237c896507b8d654165c36b61c9a3080e6dd042dea562a4a494fb73133634",  
    "time": "2023-01-24T21:10:32.74684667Z",  
    "message": "potential slashable offense: sender role not authorized  
to send message on channel"  
}
```

Identifying the Source of Malicious Traffic

Most log messages include either the node ID or peer ID. Peer ID is the ID used to identify nodes on by the libp2p library. Peer IDs are derived from the node's networking public key, so there is a 1:1 mapping between node ID and peer ID.

The two simplest ways to match a node ID to a peer ID:

1. inbound connection established and outbound connection established log messages contain both the node and peer IDs
2. The following admin command will return the node info for a given peer ID:

```
curl localhost:9002/admin/runcommand \  
-H 'Content-Type: application/json' \  
-d '{"commandName": "get-latest-identity", "data": { "peerid":  
"QmY2kby3xt3ugu2QqJP5w24rP4HSakYgDFpAJylifSRkF7" }}'
```

If you cannot find any log messages at the current log level, you may need to enable debug logging.

See the admin server's README for an example.

Reporting the Byzantine Node

Report the suspicious node on Discord in the #flowValidators-alerts channel along with all the evidence you have collected (log messages, other networking related metrics, etc). This will alert other node operators who can review their nodes to corroborate the report. Using evidence from multiple operators, a consensus can be reached about the suspicious node, and appropriate action can be taken.

Blocking a Byzantine Node

Once a consensus is reached about the suspicious node on Discord among the node operators, the suspicious node can be blocked using the admin command.

```
curl localhost: 9002/admin/runcommand \  
&gt;
```

```
-H 'Content-Type: application/json' \
-d '{"commandName": "set-config", "data": {"network-id-provider-
blocklist": ["<suspicious node id>"]}}
```

After blocking the node, all traffic coming from the node will be rejected and you should only see logs about reject messages and connections for that node ID.

Unblocking a Node

If you need to unblock a node, you can use the same command to remove the node ID from the blocklist.

Simply run it again with an empty list to remove all blocked nodes, or an existing list with the specific node ID you want to unblock removed.

The following command returns a list of the currently blocked nodes.

```
curl localhost: 9002/admin/runcommand \
-H 'Content-Type: application/json' \
-d '{"commandName": "get-config", "data": "network-id-provider-
blocklist"}}
```

After unblocking the node, connections and traffic coming from the node should resume.

db-encryption-existing-operator.md:

```
---
title: Database Encryption for Existing Node Operators
sidebarlabel: Database Encryption for Existing Node Operators
description: Instructions for existing Node Operators to follow to create a machine account for their collection or consensus nodes.
sidebarposition: 4
---
```

In Mainnet14, the DKG (distributed key generation) is turned on, requiring storage of dynamically generated confidential data (random beacon keys). These are stored in a separate database which is new with the Mainnet14 release.

All node operators joining after Mainnet14 will generate encryption keys for this database through the node bootstrapping and staking process. We strongly recommend all node operators (especially consensus node operators) generate an encryption key for this database. This guide demonstrates how to enable encryption for this database for existing operators.

Downloading Bootstrap Utility

```
<Callout type="warning">
  If you have downloaded the bootstrapping kit previously, ensure that
  you do
    this step again to get the latest copy of the bootstrapping kit since
    there
    have been significant changes to it.
</Callout>
```

Follow the instructions here to download the latest version of the bootstrapping kit, then return to this page.

Generate Database Encryption Key

You will need to generate an encryption key for the database using the bootstrap utility.

```
<Callout type="warning">
  Ensure you run the following commands on the machine you use to run
  your node software.
  The bootstrap directory passed to the -o flag must be the same
  bootstrap directory used by your node.
  The default location is /var/flow/bootstrap, but double-check your
  setup before continuing.
</Callout>
```

```
shell GenerateEncryptionKey
$./boot-tools/bootstrap db-encryption-key -o ./bootstrap
<nill> INF generated db encryption key
<nill> INF wrote file bootstrap/private-root-information/private-node-
infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9secre
tsdb-key

$tree ./bootstrap/
./bootstrap
└── private-root-information
    └── private-node-
        infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
            ├── node-info.priv.json
            └── secretsdb-key
    └── public-root-information
        ├── node-id
        └── node-
            info.pub.ab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
            .json

3 directories, 4 files
```

```
# faq.md:

---
title: Operator FAQ
sidebarposition: 1
---



## Operator FAQ



Can anybody run a node? What is the approval process?



Anyone can run an observer node.



Anyone can run an Access Node after registering and staking. See Access Node Setup for detailed instructions.



For the other node roles, individuals can go through an application process that involves asking about their background and experience contributing to decentralized projects. To pursue an application, please visit the Flow website here to apply.



Pending approval, new node operators will be onboarded and invited to join a webinar to meet the team and share more about how they'll grow the community. Node Operators are invited to join and participate in Flow's Node Validator Discord channel for setup questions and network announcements.



In the long-term, anyone can run a node validator on Flow.



How do I generate keys?



Please follow the instructions provided here: Generate Your Node Keys



How do I check on the status of my node?



Please follow the instructions provided here: Monitoring nodes



Can I bootstrap and run a node at any time?



Flow allows nodes to join/leave the network each time a new epoch begins (roughly once per week).  
See Staking & Epochs for general information and Node Setup for a guide to running a new node.



Would it hurt the network to have a node that constantly spins up and down?



All staked nodes except access nodes, have to be online at all time. A staked node, other than an access node, which is not online can cause severe degradation of network performance and will be subjected to slashing of rewards.  
A way to prevent this is to check your equipment meets Flow's recommended requirements, periodically checking for updates and announcements in


```

Discord but also using a node monitoring system for when your node does go offline.

Does Flow has a regular schedule for Sporks?

Yes, see Upcoming Sporks for the latest schedule. Currently, Flow has a Mainnet Spork and a Testnet Spork roughly every two months.

How do I update the Node Software?

One of the reasons for a spork is to make sure all nodes update to the latest software version. Hence, you should have the latest software update as long as you are participating in each spork.
However, if we do release any software update in between a Spork (e.g. an emergency patch) we will announce it on Discord.

Is there any way to know if a node is currently online?

To verify if a node is online, please setup metrics for the node.

Can I migrate a node to a new machine?

Yes, as long as you retain the bootstrap information which includes the node staking key, networking key, IP address and port from the old node to the new.

More on this here

Where can I find how many nodes are currently running Flow?

If you are running a node, then you most definitely have this information on your node in the file <your bootstrap dir>/public-root-information/node-infos.pub.json. If you are not running a node, you can find this information by using a Cadence script to query the Staking Smart Contract (or check Flowdiver)

Why do I need to update my node's ulimit?

Flow nodes create network connections to other nodes on the network to participate in the protocol. The node's operating system represents these connections as file descriptors, and uses soft and hard limits to control the number of open files. The node software uses these limits to manage how many connections it will open and accept from other nodes. If the limit is too low, the node will not be able to communicate with its peers, preventing it from functioning properly.

```
# hcu.md:  
---  
title: Height coordinated upgrade (HCU)  
sidebarlabel: Height coordinated upgrade  
---
```

Overview

To enable rapid development of the Flow Protocol, the height coordinated upgrade method is used to roll out non-breaking changes such as bug fixes, feature implementations and security fixes.

HCU versus Spork

A spork requires a coordinated network upgrade process where node operators upgrade their node software and re-initialize with a consolidated representation of the previous spork's state.

It is used to roll out changes which may be non-backward compatible with respect to the protocol and the execution state.

Spork entails a network downtime as all nodes in the system are upgraded and brought back online.

Sporks are only executed once every quarter.

A height coordinated upgrade (HCU) on the other hand allows the execution and the verification nodes to be upgraded without stopping the network. There is no network downtime during an HCU but the transaction execution will stop for few minutes while the execution nodes restart. Currently, an HCU is only used to update the execution and the verification nodes.

For other node types, a simple rolling upgrade is used where operators are asked to upgrade their nodes async.

HCU process

The HCU is executed in two parts.

The first part is executed by the service committee. In this, the version boundary at which the execution nodes and verification nodes should stop is set on chain by submitting the setversionboundary transaction. The version boundary includes the block height at which the two node types should stop and the new node software version that the nodes should compare after a restart.

The second part is executed by the node operator. In this the node operator, monitors the execution and verification node that they are running. When the nodes reach the height set on chain, they stop if their version is lower than the version specified in the version boundary. At this point, the operator should update the node version to the new node software version and start the node again. The node will continue from where it left off.

The block height and the node version will be announced by the Flow team on Discord as well as the forum page.

It can also be directly queried from the chain using the following script.

TODO: insert flow cli command here to query the block version details.

```
# machine-existing-operator.md:  
  
---  
title: Machine Accounts for Existing Node Operators  
sidebarlabel: Machine Accounts for Existing Node Operators  
description: Instructions for existing Node Operators to follow to create  
a machine account for their collection or consensus nodes.  
sidebarposition: 6  
---
```

The Flow Epoch Preparation Protocol requires that collection and consensus nodes use an automated machine account to participate in important processes required to start the next epoch. (QC and DKG, respectively)

Starting on Thursday, August 26th 2021, all collector and consensus nodes who register with Flow Port will automatically create and initialize this machine account as part of their node registration.

If you have an existing consensus or collection node that you registered with Flow Port before Thursday August 26th, you will need to create this Machine Account manually in order to participate in epochs. You will need to create one Machine Account for each consensus or collection node that you operate.

This guide will walk you through creating a Machine Account and getting it set up.

```
<Callout type="warning">  
  During this process you will generate a new private key which will have sole control over your machine account.  
  This private key will be stored on the machine you use to run your node, alongside your staking and networking keys.  
  Loss of any of these keys (staking, networking, or machine account) will require you to un-stake your tokens, start a completely new node, and register the new node to continue participating in the Flow network, which takes multiple weeks.  
</Callout>
```

Downloading Bootstrap Utility

```
<Callout type="warning">  
  If you have downloaded the bootstrapping kit previously, ensure that you do this step again to get the latest copy of the bootstrapping kit since there have been significant changes to it.  
</Callout>
```

Follow the instructions here to download the latest version of the bootstrapping kit, then return to this page.

Generate Machine Account key

You will need to generate a Machine account private key using the bootstrap utility.

```
<Callout type="warning">
```

Ensure you run the following commands on the machine you use to run your node software.

The bootstrap directory passed to the -o flag must be the same bootstrap directory used by your node.

The default location is /var/flow/bootstrap, but double-check your setup before continuing.

```
</Callout>
```

```
shell GenerateMachineAccountKey
$./boot-tools/bootstrap machine-account-key -o ./bootstrap
<n1> INF generated machine account private key
<n1> INF encoded machine account public key for entry to Flow Port
machineAccountPubKey=f847b84031d9f47b88435e4ea828310529d2c60e806395da50d3
dd0dd2f32e2de336fb44eb06488645673850897d7cc017701d7e6272a1ab7f2f125aede46
363e973444a02038203e8
<n1> INF wrote file bootstrap/private-root-information/private-node-
info6f6e98c983dbd9aa69320452949b81abeab2ac591a247f55f19f4dbf0b477d26/node
-machine-account-key.priv.json

$tree ./bootstrap/
./bootstrap
└── private-root-information
    └── private-node-
        infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
            ├── node-info.priv.json
            ├── node-machine-account-key.priv.json
            └── secretsdb-key
        └── public-root-information
            ├── node-id
            └── node-
                info.pub.ab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
                .json

3 directories, 4 files
```

Create Machine Account

You will now need to copy the Machine account public key displayed in the terminal output and head over to Flow Port to submit a transaction to create a Machine Account.

For example, from the example above, we would copy f847... from this line:

```
shell Example
<n1l> INF encoded machine account public key for entry to Flow Port
machineAccountPubKey=f847b84031d9f47b88435e4ea828310529d2c60e806395da50d3
dd0dd2f32e2de336fb44eb06488645673850897d7cc017701d7e6272a1ab7f2f125aede46
363e973444a02038203e8
```

This process will create your machine account for you and show you your machine account's address, which you will need to save for the next step.

Finalize Machine Account setup

You will now need to use the bootstrap utility to run machine-account with the created address to finalize the set up of your Machine account.

```
shell
$ ./boot-tools/bootstrap machine-account --address
${YOURMACHINEACCOUNTADDRESS} -o ./bootstrap
```

```
shell Example
$./boot-tools/bootstrap machine-account --address 0x1de23de44985c7e7 -o
./bootstrap
<n1l> INF read machine account private key json
<n1l> DBG encoded public machine account key
machineAccountPubKey=2743786d1ff1bf7d7026d693a774210eaa54728343859baab62e
2df7f71a370651f4c7fd239d07af170e484eedd4f3c2df47103f6c39baf2eb2a50f67bbcb
a6a
<n1l> INF wrote file bootstrap/private-root-information/private-node-
info6f6e98c983dbd9aa69320452949b81abeab2ac591a247f55f19f4dbf0b477d26/node
-machine-account-info.priv.json
```

```
$tree ./bootstrap/
./bootstrap
└── private-root-information
    └── private-node-
        infod60bd55ee616c5c297cae1d5cfb7f65e7e04014d9c4abe595af2fd83f3cf160
            ├── node-info.priv.json
            ├── node-machine-account-info.priv.json
            ├── node-machine-account-key.priv.json
            └── secretsdb-key
        └── public-root-information
            └── node-id
                └── node-
                    info.pub.d60bd55ee616c5c297cae1d5cfb7f65e7e04014d9c4abe595af2fd83f3cf160
                        .json
```

3 directories, 5 files

After running this step, you should see the node-machine-account-info.priv.json file in your bootstrap directory as shown above.

Verify Machine Account Setup

After finalizing your machine account setup, you should verify its correctness with the check-machine-account command:

```
shell CheckMachineAccount
$ ./boot-tools/bootstrap check-machine-account --access-address
access.mainnet.nodes.onflow.org:9000 -o ./bootstrap
<nil> DBG read machine account info from disk hashalgo=SHA3256 keyindex=0
machineaccountaddress=0x284463aa6e25877c
machineaccountpubkey=f847b84051bad4512101640772bf5e05e8a49868d92eaf9ebcd4
1030881d95485769af28653c5c53216cdcd4554384bb3ff6396a2ac04842422d55f0562
496ad8d952802038203e8 signingalgo=ECDSAP256
<nil> DBG checking machine account configuration...
machineaccountaddress=0x284463aa6e25877c role=consensus
<nil> DBG machine account balance: 0.10000000
<nil> INF □ machine account is configured correctly
```

This command will detect and provide information about common misconfigurations, or confirm that the machine account is configured correctly.

```
# monitoring-nodes.md:
```

```
---
title: Monitoring Node Health
sidebarlabel: Node Monitoring
sidebarposition: 7
---
```

A Flow node generates logs and publishes metrics as it runs. These logs and metrics can be used to gain insights into the health of the node.

Logs

Logs are emitted to stdout as JSON formed strings. Where these logs are available on your system depends on how you launch your containers. On systemd systems for example, the logs will be sent to the system journal daemon journald. Other systems may log to /var/log.

Metrics

Flow nodes produce health metrics in the form of Prometheus metrics, exposed from the node software on /metrics.

If you wish to make use of these metrics, you'll need to set up a Prometheus server to scrape your Nodes. Alternatively, you can deploy the Prometheus Server on top of your current Flow node to see the metrics without creating an additional server.

> The flow-go application doesn't expose any metrics from the underlying host such as CPU, network, or disk usages. It is recommended you collect these metrics in addition to the ones provided by flow using a tool like node exporter (<https://github.com/prometheus/nodeexporter>)

1. Copy the following Prometheus configuration into your current flow node

```
yaml
global:
  scrapeinterval: 15s By default, scrape targets every 15 seconds.

  scrapeconfigs:
    The job name is added as a label job=<jobname> to any timeseries
scraped from this config.
    - jobname: 'prometheus'

      Override the global default and scrape targets from this job
every 5 seconds.
      scrapeinterval: 5s

  staticconfigs:
    - targets: ['localhost:8080']
```

- ## 2. Start Prometheus server

```
shell
  docker run \
    --network=host \
    -p 9090:9090 \
    -v /path/to/prometheus.yml:/etc/prometheus/prometheus.yml \
    prom/prometheus"
```

3. (optional) Port forward to the node if you are not able to access port 9090 directly via the browser

```
ssh -L 9090:127.0.0.1:9090 YOURNODE
```

4. Open your browser and go to the URL <http://localhost:9090/graph> to load the Prometheus Dashboard

Key Metric Overview

The following are some important metrics produced by the node.

Metric Name	Description
go\\	Go runtime metrics

```

| consensuscompliancefinalizedheight | Latest height finalized by this
node; should increase at a constant rate.
|
| consensuscompliancesealedheight    | Latest height sealed by this node;
should increase at a constant rate.
|
| consensushotstuffcurview          | Current view of the HotStuff
consensus algoritm; Consensus/Collection only; should increase at a
constant rate.
|
| consensushotstufftimeoutseconds   | How long it takes to timeout
failed rounds; Consensus/Collection only; values consistently larger than
5s are abnormal.

```

Machine Account

Collection and consensus nodes use a machine account that must be kept funded. See here for details.

Nodes check their machine account's configuration and funding and produce metrics.

Metric Name	Description
-----	-----

machineaccountbalance	The current
balance (FLOW)	
machineaccountrecommendedminbalance	The recommended
minimum balance (FLOW)	
machineaccountismisconfigured	0 if the node is
configured correctly; 1 if the node is misconfigured	

To be notified when your node's machine account needs to be refilled or has a configuration error, you can set up alerts.

When the machine account balance needs to be refilled:

```
machineaccountbalance < machineaccountrecommendedminbalance
```

When the machine account has a configuration error:

```
machineaccountismisconfigured > 0
```

The metrics include the account address of the machine account (acctaddress label) for convenience:

```
HELP machineaccountbalance the last observed balance of this node's
machine account, in units of FLOW
TYPE machineaccountbalance gauge
machineaccountbalance{acctaddress="7b16b57ae0a3c6aa"} 9.99464935
```

```
# node-bootstrap.md:

---
title: Node Bootstrap
sidebarlabel: Node Bootstrapping
description: How to get started running a node on Flow
sidebarposition: 8
---

This guide is for getting a new node staked and running on Flow other than a permissionless Access node. For running a permissionless Access node see Access node setup. For sporking documentation for existing node operators, see Spork Process.
```

Timing

New nodes are able to join the network each time a new epoch begins. In order to join the network at epoch N+1, the node must be registered with sufficient stake and authorized by the service account prior to the end of epoch N's Staking Auction Phase. Confirmation of a new node's inclusion in epoch N+1 is included in the EpochSetup event.

Nodes registered for epoch N+1 are able to participate in network communication on a limited basis starting in the Epoch Setup Phase of epoch N.

!Flow Epoch Schedule

Once registered and confirmed to join the network at epoch N+1, the node must start up before epoch N+1 begins.

Verification & Access nodes may start up any time during the Epoch Setup Phase.

Consensus & Collection nodes must start up within the first 1000 views (30mins) of the Epoch Setup Phase to participate in the Epoch Preparation Protocol.

Step 1 - Run Genesis Bootstrap

:::info

You will need to run this process for each node that you are operating

:::

Download the Bootstrapping Kit

:::warning

If you have downloaded the bootstrapping kit previously, ensure you check the hash below still matches. If not, re-download to ensure you are using the most up-to-date version.

:::

```
shell
curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-tools.tar
tar -xvf boot-tools.tar
chmod +x ./boot-tools/bootstrap
chmod +x ./boot-tools/transit

shell CheckSHA256
sha256sum ./boot-tools/bootstrapcmd
460cfefeb52b40d8b8b0c4641bc4e423bcc90f82068e95f4267803ed32c26d60 ./boot-
tools/bootstrap

sha256sum ./boot-tools/transit
f146bdc82ce0cce73c0fb9de284b2e2639e851120f8b89a1dd9368e8442123b4 ./boot-
tools/transit
```

Generate Your Node Keys

Network Address

:::info

Use a fully qualified domain name for the network address. Please also include the port number in the network address e.g. example.com:3569

:::

:::warning

Do not include in http:// format.

:::

:::info

If you are running multiple nodes, please ensure you have different addresses for each node.

:::

:::warning

All your current keys and Flow genesis files should be in the bootstrap folder created earlier. Please take a back up of the entire folder.

:::

```

shell
Skip this section if this is your first time ##
If you joined our network previously, make sure to take a backup of your
previously generated keys!
cp -r /path/to/bootstrap /path/to/bootstrap.bak
#####
Generate Keys
$ mkdir ./bootstrap
YOURNODEADDRESS: FQDN associated to your instance (do NOT use an IP
address, use a hostname)
YOURNODEROLE: The Flow nodes that you wish to run, it should be ONE of
the following - [ access, collection, consensus, execution, verification
]
$ ./boot-tools/bootstrap key --address \"YOURNODEADDRESSGOESHERE:3569\" -
-role YOURNODEROLEGOESHERE -o ./bootstrap

```

```

shell Example
$./boot-tools/bootstrap key --address "consensus-001.nodes.flow.com:3569"
--role consensus -o ./bootstrap
<nil> DBG will generate networking key
<nil> INF generated networking key
<nil> DBG will generate staking key
<nil> INF generated staking key
<nil> DBG will generate db encryption key
<nil> INF generated db encryption key
<nil> DBG assembling node information address=consensus-
001.nodes.flow.com:3569
<nil> DBG encoded public staking and network keys
networkPubKey=7f31ae79017a2a58a5e59af9184f440d08885a16614b2c4e361019fa72a
9a1a42bf85b4e3f9674782f12ca06afd9782e9ccf19496baed069139385b82f8f40f6
stakingPubKey=829d086b292d84de8e7938fd2fafaf8f51a6e025f429291835c20e59d9e2
5665febfb24fa59de12a4df08be7e82c5413180cc7b1c73e01f26e05344506aaca4fa9cc00
9dc1c33f8ba3d7c7509e86d3d3e7341b43b9bf80bb9fba56ae0b3135dd72
<nil> INF wrote file bootstrap/public-root-information/node-id
<nil> INF wrote file bootstrap/private-root-information/private-node-
infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9/node
-info.priv.json
<nil> INF wrote file bootstrap/private-root-information/private-node-
info5e44ad5598bb0acb44784f629e84000ffea34d5552427247d9008ccf147fb87f/secretsdb-key
<nil> INF wrote file bootstrap/public-root-information/node-
info.pub.ab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
.json
<nil> DBG will generate machine account key
<nil> INF generated machine account key
<nil> DBG assembling machine account information address=consensus-
001.nodes.flow.com:3569
<nil> INF encoded machine account public key for entry to Flow Port
machineAccountPubKey=f847b8406e8969b869014cd1684770a8db02d01621dd1846cdf4
2fc2bca3444d2d55fe7abf740c548639cc8451bcae0cd6a489e6ff59bb6b38c2cfb83e095
e81035e507b02038203e8

```

```
<nil> INF wrote file bootstrap/private-root-information/private-node-
infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9/node
-machine-account-key.priv.json
```

```
$tree ./bootstrap/
./bootstrap
└── private-root-information
    └── private-node-
        infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
            ├── node-info.priv.json
            ├── node-machine-account-key.priv.json
            └── secretsdb-key
        └── public-root-information
            └── node-id
                └── node-
                    info.pub.ab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9
                        .json
```

3 directories, 4 files

:::info

For consensus and collection node types an additional key will be created for the Machine Account.

For all other node types this will not be needed.

:::

Machine Account Creation

If you are running a collection and consensus node, you will have an additional private key file (`node-machine-account-key.priv.json`) which contains the private key for your node's machine account. You can learn more about machine accounts here.

In Step 2 of this guide, when you submit a transaction to stake your node, you will need to provide the machine account public key, which can be found in the output of the previous bootstrap key command.

```
shell MachineAccountPublicKey
$./boot-tools/bootstrap key --address YOURNODEADDRESSGOESHHERE --role
YOURNODEROLEGOESHERE -o ./bootstrap
...
<nil> DBG encoded public machine account key
machineAccountPubKey=1b9c00e6f0930792c5738d3397169f8a592416f334cf11e84e63
27b98691f2b72158b40886a4c3663696f96cd15bfb5a08730e529f62a00c78e2405013a60
16d
<nil> INF wrote file bootstrap/private-root-information/private-node-
infoab6e0b15837de7e5261777cb65665b318cf3f94492dde27c1ea13830e989bbf9/node
-machine-account-key.priv.json
```

```
:::warning  
Copy the machine account public key somewhere safe. You will need it in a  
later step.
```

```
:::
```

Step 2 - Stake Your Node

Stake your node via Flow Port

The node details (Node ID, Network Address, Networking Key and Staking Key) that need to be submitted when staking the node on Flow Port, can be found in the file: ./bootstrap/public-root-information/node-info.pub.<node-id>.json.

```
shell Example  
$cat ./bootstrap/public-root-information/node-  
info.pub.39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bfbf30799a  
.json  
{  
  "Role": "consensus",  
  "Address": "consensus-001.nodes.flow.com:3569",  
  "NodeID":  
"39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bfbf30799a",  
  "Weight": 0,  
  "NetworkPubKey":  
"d92e3d5880abe233cf9fe9104db34bbb31251468a541454722b3870c04156a1b0504aef4  
43bcaad124b997384b8fe7052847ce1e6189af1392d865e6be69835b",  
  "StakingPubKey":  
"917826e018f056a00b778a58ae83054906957ffd4b6f1b7da083551f7a9f35e02b76ace5  
0424ed7d2c9fc69207a59f0f08a031048f5641db94e77d0648b24d150dedd54bab7cd44b4  
aa60cf54be418647b0b3965f8ae54c0bcb48ae9d705162"  
}
```

If you are running a collection or consensus node, you will need to provide an additional field Machine Account Public Key. This value is found in the output of the bootstrap key command from Step 1.

Staking a collection or consensus node will also create a machine account for the node. The machine account will be mentioned in the output of the staking transaction displayed by Flow Port. Please save the machine account for the next step.

```
:::info
```

Please let us know your node id via discord or email.

```
:::
```

Finalize Machine Account Setup

```
:::warning
```

If you are not running a collection or consensus node, you can skip this step.

```
:::
```

You will now need to use the bootstrap utility to run machine-account with the created address to finalize the set up of your Machine account.

```
shell
$ ./boot-tools/bootstrap machine-account --address
YOURMACHINEACCOUNTADDRESSGOESHHERE -o ./bootstrap
```

```
shell Example
$ ./boot-tools/bootstrap machine-account --address 0x1de23de44985c7e7 -o
./bootstrap
<nil> INF read machine account private key json
<nil> DBG encoded public machine account key
machineAccountPubKey=2743786d1ff1bf7d7026d693a774210eaa54728343859baab62e
2df7f71a370651f4c7fd239d07af170e484eedd4f3c2df47103f6c39baf2eb2a50f67bbcb
a6a
<nil> INF wrote file bootstrap/private-root-information/private-node-
info6f6e98c983dbd9aa69320452949b81abeab2ac591a247f55f19f4dbf0b477d26/node
-machine-account-info.priv.json
```

```
$tree ./bootstrap/
./bootstrap
└── private-root-information
    └── private-node-
        infod60bd55ee616c5c297cae1d5cfb7f65e7e04014d9c4abe595af2fd83f3cf160
            ├── node-info.priv.json
            ├── node-machine-account-info.priv.json
            ├── node-machine-account-key.priv.json
            └── secretsdb-key
        └── public-root-information
            └── node-id
                └── node-
                    info.pub.d60bd55ee616c5c297cae1d5cfb7f65e7e04014d9c4abe595af2fd83f3cf160
                        .json
3 directories, 5 files
```

After running this step, you should see the node-machine-account-info.priv.json file in your bootstrap directory as shown above.

Verify Machine Account Setup

After finalizing your machine account setup, you should verify its correctness with the check-machine-account command:

```
shell CheckMachineAccount
$ ./boot-tools/bootstrap check-machine-account --access-address
access.mainnet.nodes.onflow.org:9000 -o ./bootstrap
<nil> DBG read machine account info from disk hashalgo=SHA3256 keyindex=0
machineaccountaddress=0x284463aa6e25877c
machineaccountpubkey=f847b84051bad4512101640772bf5e05e8a49868d92eaf9ebcd4
1030881d95485769af28653c5c53216cdcd4554384bb3ff6396a2ac04842422d55f0562
496ad8d952802038203e8 signingalgo=ECDSAP256
<nil> DBG checking machine account configuration...
machineaccountaddress=0x284463aa6e25877c role=consensus
<nil> DBG machine account balance: 0.10000000
<nil> INF ✅ machine account is configured correctly
```

This command will detect and provide information about common misconfigurations, or confirm that the machine account is configured correctly.

Push transit keys (consensus node only)

If you are running a consensus node, run the following command to generate the transit keys.

```
shell transit
$ ./boot-tools/transit prepare -b ./bootstrap -r consensus
<nil> INF running prepare
<nil> INF generating key pair
<nil> INF completed preparation role=consensus
```

This will generate the public and private transit keys under the bootstrap folder.

The transit keys are used to transfer the DKG keys after a network upgrade.

Please share the public transit key with the Flow Foundation via discord or email.

Step 3 - Start Your Flow Node

Ensure you have configured your node using the Node Setup guide.

Confirming authorization

You can confirm your node's successful registration and authorization by executing a Cadence script to query the Staking Contract.

At the end of the Staking Auction Phase, the members of the Proposed Identity Table are confirmed as authorized participants in the next epoch.

Therefore, if your node ID appears in the Proposed Identity Table during the Staking Auction Phase, your node will be a participant in the next epoch.

You can read the current Proposed Identity Table using the `getProposedTable` script.

You can read the current epoch phase using the `getEpochPhase` script. (A return value of 0 indicates the Staking Auction Phase.)

Trusted Root Snapshot

Once your node has been registered and authorized by the service account, it will be able to participate in the next epoch.

!Flow Epoch Schedule

A new node must bootstrap with a trusted root snapshot of the protocol state, where the node is a confirmed participant.

Since new nodes are confirmed at the end of the Staking Auction Phase, this means that, if the node is registered to join at epoch N+1, it must use a root snapshot from within the Epoch Setup Phase of epoch N.

Dynamic Startup

Flow provides a mechanism called Dynamic Startup to simplify the process of obtaining the root snapshot.

When using Dynamic Startup, the node can be started at any time during the Staking Auction Phase.

The node will wait for the Epoch Setup Phase to begin, retrieve a valid root snapshot from a trusted Access Node, then bootstrap its state and join the network.

This is the recommended way to start your node for the first time.

1. Remove any `root-protocol-state-snapshot.json` file from your bootstrap folder. (If this file is present the node will attempt to bootstrap with it rather than Dynamic Startup.)
2. Select a trusted Access Node to provide the root snapshot. You will need this node's secure GRPC server address and Networking Public Key.
3. Configure Dynamic Startup by adding flags:

```
shell ExampleDynamicStartupFlags
... \
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \
--dynamic-startup-access-
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae
```

4. Start your node (see guide)

```
:::info
```

Once the node has bootstrapped, these flags will be ignored and may be removed.

```
:::
```

Manually Provisioned Root Snapshot

You can also provision the root snapshot file manually, then start the node without configuring Dynamic Startup.
See here for the available options to provision a Root Snapshot.

:::warning

The snapshot must be within the Epoch Setup Phase.

:::

:::warning

Since Collection and Consensus Nodes must start up in the first 30mins of the Epoch Setup Phase (see Timing), the snapshot must be provisioned within this time window.

:::

Once a valid root snapshot file is downloaded to the node's bootstrap folder, it can be started (see guide)

node-economics.md:

```
---
title: Node Economics
sidebarlabel: Node Economics
description: Node Operator Economics - An illustration
sidebarposition: 8
---
```

Node operators play a crucial role in securing the Flow network. Here's a simple example to illustrate what node operators can expect in terms of node economics.

Node Operator Economics: An illustration

:::warning

This illustration is strictly to serve as an example. Actual numbers will vary based on several factors.

For real-time numbers, please refer to the block explorer.

:::

	Parameter	Value	Explanation
	-----	-----	-----
A	Node Operator's Stake	500,000 FLOW	Assuming minimum staking requirements for a consensus node. Remember there's no upper cap on how much FLOW can be staked to a Flow node.

B	Delegation to node	1,000,000 FLOW	Funds that individual/institutional delegators delegate to your node. Assuming 1M FLOW for this example.	
C	APY	10%	Subject to change based on total ecosystem stake in each epoch. Remember APY = R / S, where S = Total FLOW Staked / Total FLOW Supply and R = 5% ("reward rate")	
D	Delegation Rate	8%	Fee taken by the node operator from delegator rewards to cover their operating expenses, currently set at 8% of the rewards received by delegators. Note that the 8% fee is only applied to the staking reward, not to the tokens delegated.	
E	Annual Staking Rewards	50,000 FLOW	Product of A x C; the number shown is annualized but is paid each epoch (week).	
F	Annual Delegator Fee	8,000 FLOW	Product of B x C x D; ; the number shown is annualized but is paid each epoch (week).	
G	Annual (Gross) Rewards	58,000 FLOW	Sum of E and F	
H	COGS	4,190 FLOW	Assumed costs of running a consensus node in FLOW assuming 1US\$/FLOW. The actual cost will vary depending on several factors such as self-hosted vs cloud, bare metal vs VM, the type of node, the FLOW exchange rate.	
J	Net Annual Rewards	53,810 FLOW	G less H	

Note

1. Each year, 5% of the total Flow supply is distributed as rewards to incentivize validators and delegators. While the total rewards for each epoch are fixed, the rewards for individual stakers vary depending on the amount they stake and the total funds delegated to their node.
2. All Flow node types follow the same economic principles, with the only difference being their minimum staking requirements. For details on the minimum stakes needed for each node type, see [here](#).

node-migration.md:

```
---
title: Node Migration
description: How to migrate a Flow node from one machine to another
sidebarposition: 9
---
```

There are a few different methods to migrate a running Flow node from one machine to the other.

Choose the method depending upon what part of the staking data of the node is changing.

Method 1 - No change to the node staking data

If there is no change to the network address or the staking and networking keys and only the hardware the node is running needs to be changed then do the following:

1. Stop the Flow node.

2. Copy over the bootstrap data (typically under /var/flow/bootstrap) which contains the node private key to the new machine.
3. Copy over the data folder (typically under /var/flow/data) which contains the state data.
4. Start the new node on the same network address as the old one.

:::warning

Please ensure that there is minimal downtime during this migration.

:::

:::warning

The network address is currently part of the staking data that was submitted for the node. It is how other nodes in the network discover this node.

Hence, the network address of the node must stay the same between epochs otherwise the node will become unreachable for the other nodes and stop functioning.

:::

Method 2 - Network address change

A change to the node network address (IP or a hostname) can only be done during the spork process.

To change the networking address:

1. A day before the upcoming mainnet spork, change the network address for the nodes in Flow Port (using the update network address feature). The change will not take effect till an epoch transition happens.
2. Change the addresses in the /var/flow/bootstrap/private-root-information/private-node-info<nodeid>/node-info.priv.json json file on the node.
3. A spork also causes an epoch transition, and the new addresses will take effect after the spork immediately.

Method 3 - Staking or networking key change

If the node after migration will be using new staking or networking keys then it needs to be unstaked and then re-staked with the new keys.

1. Unstake the node via Flow Port.
2. Register the new node via Flow Port with the new staking information.
3. Run the new node with the new keys and network address. It should be able to join the network at the next epoch (see timing)

:::warning

Unstaking a node will result in the node not earning rewards for the next epoch.

Delegators to the old node will have their tokens unstaked automatically. They will also stop earning rewards unless they withdraw their unstaked tokens and delegate them to a different node.

:::

```
# node-providers.md:
```

```
---
```

sidebarposition: 18
description: |
 Easy access to Flow's blockchain. Providers handle the technical work,
 letting you use Flow's features without managing nodes yourself.
sidebarcustomprops:
 icon: 🛡

```
---
```

Node Providers

Quick Node

QuickNode offers convenient access to Flow's blockchain infrastructure, allowing developers and businesses to utilize Flow's capabilities without the complexities of managing nodes themselves. It offers reliable and fast connectivity to blockchain networks, sparing users from the resource-intensive task of running their own full nodes.

Supported Networks

- Testnet
- Mainnet

Tatum

Tatum provides a comprehensive platform that simplifies the process of building, testing, and deploying blockchain applications. With Tatum, users can access infrastructure, an SDK, and a unified API to develop blockchain apps without the need to handle individual blockchain node configuration or maintenance.

Supported Networks

- Testnet
- Mainnet

```
# node-provisioning.md:
```

```
---
```

title: Provisioning a Flow node
sidebarlabel: Node Provisioning
description: Hardware, networking and Operating system setup for a Flow node
sidebarposition: 10

Hardware Requirements

The hardware your Node will need varies depending on the role your Node will play in the Flow network. For an overview of the differences see the [Node Roles Overview](#).

Node Type	CPU	Memory	Disk	Example GCP Instance	Example AWS Instance
Collection	4 cores	32 GB	200 GB	n2-highmem-4	r6i.xlarge
Consensus	2 cores	16 GB	200 GB	n2-standard-4	m6a.xlarge
Execution	128 cores	864 GB	9 TB (with maintenance see: pruning chunk data pack or 30 TB without maintenance)	n2-highmem-128	
Verification	2 cores	16 GB	200 GB	n2-highmem-2	r6a.large
Access	16 cores	64 GB	750 GB	n2-standard-16	m6i.4xlarge
Observer	2 cores	4 GB	300 GB	n2-standard-4	m6i.xlarge
EVM Gateway	2 cores	32 GB	30 GB	n2-highmem-4	r6i.xlarge

Note: The above numbers represent our current best estimate for the state of the network. These will be actively updated as we continue benchmarking the network's performance.

Networking Requirements

Most of the load on your nodes will be messages sent back and forth between other nodes on the network. Make sure you have a sufficiently fast connection; we recommend at least 1Gbps, and 5Gbps is better.

Each node will require a fixed DNS name and we will refer to this more generally as your 'Node Address' from here on out.

```
<Callout type="info" title="Node Address Requirements">
  Your Node Address must be a publicly routable valid DNS name
  that points to your node. This is how other nodes in the network will
  communicate with you.
</Callout>
```

Your firewalls must expose TCP/3569 for both, ingress and egress.

If you are running an Access Node, you must also expose the GRPC port 9000 to your internal network traffic. Port 9000 is not required for external ingress/egress.

Flow Architecture

Operating System Requirements

The Flow node code is distributed as a Linux container image, so your node must be running an OS with a container runtime like docker or containerd.

The bootstrapping scripts we'll use later are compiled binaries targeting an amd64 architecture, so your system must be 64-bit. Some of these scripts are bash based hence a shell interpreter that is bash compatible will also be needed.

Flow also provides systemd service and unit files as a template for installation, though systemd is not required to run Flow.

```
<Callout type="info" title="Choose Your Own Adventure">
    Flow is distributed in such a way that makes it very system agnostic.
    You are
        free to build your own orchestration around how you run your nodes
        and manage
        your keys.
```

For the remainder of this guide, we cover the most simple case, a single node being hand deployed. This should give you a good sense of what's needed, and you can modify to suit your needs from there.

```
The Flow team has tested running nodes on Ubuntu 18.04 and GCP's Container
Optimized OS, which is based on Chromium OS. If you are unsure where to start,
those are good choices.
</Callout>
```

Time synchronization

You should also ensure you run time synchronization on the machine hosting the container, to avoid clock drift. In practice, this means configuring a client for the NTP protocol, and making sure it runs as a daemon. ntpd is one recommended example. To configure it, you just have to point it to an NTP server to query periodically. A default from your Linux distribution or cloud operator may already be set, and in the interest of decentralization, our recommendation would be to use it unless you have a specific reason to do otherwise.

```
<Callout type="info" title="Time synchronization FAQ">
```

- Leap-smearing: Leap-smearing time servers and non-leap-smearing time servers are both acceptable for the magnitude of our time precision requirements - though considering very few providers offer leap smearing time servers, a "regular" time server helps ensure our pool of time providers is more diverse.

- Why not do it in the container itself? Why do we need to do this?: Without special privileges and in all major container runtimes, a container will not run with the CAPSYSTIME capability. For Flow, this means that the node software itself cannot change the time of the host machine, making the in-container use of standard time synchronization protocols ineffective.

- Why does time matter in Flow?: Time information comes up in consensus and in smart contracts. The consensus algorithm of Flow allows nodes to exit the influence of a corrupt or ineffective "leader" node by collectively deciding to switch to the next "phase" of the protocol at the right time. The smart contract language also allows developer access to block time stamps, which provide an approximation of time. To resist manipulation in each case, honest nodes must compute timing values from an aggregate of the information provided by all nodes. That approach, though resilient, is still sensitive to inaccurate time information. In other words, a node subject to clock drift but otherwise honest will not stop the consensus, but might make it slower.

</Callout>

Setup Data Directories & Disks

Flow stores protocol state on disk, as well as execution state in the case of execution nodes.

Where the data is stored is up to you. By default, the systemd files that ship with Flow use /var/flow/data.

This is where the vast majority of Flow's disk usage comes from, so you may wish to mount this directory on a separate disk from the OS.

The performance of this disk IO is also a major bottleneck for certain node types.

While all nodes need to make use of this disk, if you are running an execution node, you should make sure this is a high performing SSD.

As a rough benchmark for planning storage capacity, each Flow block will grow the data directory by 3-5KiB.

Confidential Data & Files

Flow stores dynamically generated confidential data in a separate database. We strongly recommend enabling encryption for this database - see this guide for instructions.

Confidential information used by Flow is stored in the private-root-information subtree of the bootstrap folder.

In particular:

the staking private key (node-info.priv.json)
the networking private key (node-info.priv.json)
the encryption key for the secrets database (secretsdb-key)
(if applicable) the initial random beacon private key (random-beacon.priv.json)

These files contain confidential data, and must be stored and accessed securely.

```
# node-roles.md:
```

```
---
```

```
title: Node Roles
```

```
sidebarposition: 11
```

```
---
```

Unlike most blockchains, not all Flow nodes are equal. Flow nodes all specialize and fulfill a specific role in the operation of the network. Collection, consensus, execution, verification and access nodes are all staked nodes while the observer node is not staked.

Collection

Collection nodes are bandwidth-optimized nodes divided by the protocol into several cooperating Clusters. Their first task is managing the transaction pool and collecting well-formed transactions to propose to Consensus nodes. Transactions are assigned to a cluster pseudorandomly by transaction hash. A well-formed transaction must include credentials from the guarantor of the transaction. When a Collection Node sees a well-formed transaction, it hashes the text of that transaction and signs the transaction to indicate two things: first, that it is well-formed; and second, that it will commit to storing the transaction text until the Execution nodes have finished processing it. Each cluster collects transactions, assembles them into Collections and submits a Collection Guarantee signed by a super-majority of the cluster to the Consensus nodes.

Collection nodes are required to stake a minimum of 250,000 FLOW to be a confirmed node operator.

Consensus

Consensus nodes form and propose blocks in a manner similar to traditionally-structured proof-of-stake blockchains, using the HotStuff consensus algorithm to create a globally consistent chain of blocks. Consensus nodes validate that the signed collection hashes submitted to them by Collection nodes were, in fact, signed by the required majority of Collection nodes. Thereafter, the Consensus nodes assemble the transactions into blocks and finalize them through voting. The more participants there are in this process, the more decentralized the network. However, consensus algorithms typically bottleneck the limit to the number of participants. The Flow protocol chose the HotStuff algorithm because it is flexible enough to add participants and currently supports about 100 operators. Adding more than 100 participants to the protocol by adapting HotStuff will continue to be an area of active development.

Consensus nodes act as checkpoints against other Collection nodes. They are responsible for checking that a critical number of Collection nodes reviewed and signed for the transaction. Collection nodes are held accountable by Consensus nodes. A common concern with proof-of-work- and proof-of-stake based systems is that a small subset of the population of nodes can control important resources such as the mining or stake needed to produce and vote on blocks, which is a degradation of the security of the system. By lowering the requirements to participate, Flow makes it extremely difficult and expensive to coordinate a Byzantine majority of Consensus nodes.

Consensus nodes have minimal bandwidth and computation requirements, allowing even a modest computing device (any consumer-grade hardware) to participate in the voting process and ensure the safety of the network. Many networks claim open participation, yet substantial resources – stake, computation, or otherwise – are needed to partake. Maintaining such barriers to entry undermines the security of the network. Lowering the participation requirements preserves the security of the network by providing a high degree of byzantine fault tolerance since it becomes exceedingly difficult for a subset of bad actors to subvert the network.

Consensus nodes are required to stake a minimum of 500,000 FLOW to be a confirmed node operator.

Execution

Execution nodes are the most resource-intensive nodes on the Flow network, responsible for executing transactions and maintaining the Execution State – a cryptographically-verifiable data store for all user accounts and smart contract states – as well as responding to queries related to it. Execution nodes compute the outputs of the blocks they are provided. They then ask the Collection nodes for the collections which contain transactions waiting to be executed. With this data they are able to compute the output, which is later verified by Verification nodes to ensure honesty (allocation of Verification nodes is via a sortition algorithm). The Execution nodes are primarily responsible for Flow's improvements in scale and efficiency because only a very small number of these powerful compute resources are required to compute and store the historical state.

Execution nodes give the Flow network its performance characteristics: highly scalable within a single shared state environment (i.e., no sharding). However, the significant hardware requirements make them the least accessible option for participation as a Validator. Because the revenue pool splits between relatively few nodes, the revenue per-node should more than compensate for the high capital costs of operating this node.

An Execution Node presents a hashed commitment once it has computed the output. The output is only revealed once its co-executors have also submitted their outputs. This is important to ensure nodes aren't spoofing each other's work. Once they've all submitted their answers, the output is revealed and subjected to random queries and checks run by Verification nodes. The Execution nodes have relatively low byzantine

fault tolerance. However, this does not compromise the overall security of the system because the process they perform is deterministic -- any bad actor will easily be detected and punished by Verification nodes.

This relatively small group of nodes has the most substantial technical requirements for processor speed and bandwidth because they are tasked with all the computations necessary to determine the output of the network. Allowing for this degree of specialization can reduce computation costs by at least one thousand times, and possibly much more, when compared to Ethereum.

Execution nodes are required to stake a minimum of 1,250,000 FLOW to be a confirmed node operator.

Verification

Verification nodes are responsible for confirming the correctness of the work done by Execution nodes. Individual Verification nodes only check a small amount of the total computation, but collectively they check every computation many times in parallel. Verification nodes verify Execution Receipts provided by Execution nodes and issue Result Approvals. A sortition algorithm determines which chunks of the Execution Receipt from the Execution nodes the Verification Node must query to check they were computed correctly. Ultimately, these nodes keep the Execution nodes honest; this balance of power maintains the access, security, and verifiability criteria of decentralization. It is highly byzantine fault tolerant because even if there is a substantial number of byzantine errors in the Verification Node pool, the Consensus nodes are still required to approve that transactions they signed were reviewed by a critical amount of the network.

Verification nodes are required to stake a minimum of 135,000 FLOW to be a confirmed node operator.

Access

Access nodes act as a proxy to the Flow network. The Access node routes transactions to the correct collection node and routes state queries to execution nodes (while likely caching state to answer queries in a timely manner in the future). Clients submit their transactions to any Access node or run their own if they can't find a service provider they're happy with.

Access nodes are required to stake 100 FLOW to be a confirmed node operator. However, since an access node does not participate in block production, it does not receive any staking rewards.

Observer

An observer node provides locally accessible, continuously updated, verified copy of the block data. It serves the Access API but unlike an access node, an observer node does not need to be staked, and anyone can run it without being added to the approved list of nodes.

Get started running an observer node

##

Here is a comparison of the different node roles,

Role	Staked	Recives Rewards	Permissioned
Collection	Yes	Yes	Yes
Consensus	Yes	Yes	Yes
Execution	Yes	Yes	Yes
Verification	Yes	Yes	Yes
Access	Yes	No	No <small>NEW</small>
Observer	No	No	No

node-setup.md:

```
---
title: Setting Up a Flow Node
sidebarlabel: Node Setup
description: How to run a Collection, Consensus, Verification and Execution node
sidebarposition: 12
---
```

This guide is for running a Collection, Consensus, Verification and Execution node.

If you are planning to run an Access node then refer to access node setup.

First you'll need to provision a machine or virtual machine to run your node software. Please see follow the node-provisioning guide for it.

Pull the Flow Images

The flow-go binaries are distributed as container images, and need to be pulled down to your host with your image management tool of choice.

Replace \$ROLE with the node type you are planning to run. Valid options are:

- collection
- consensus
- execution
- verification
- access

```
shell
Docker
docker pull gcr.io/flow-container-registry/${ROLE}:alpha-v0.0.1

Containerd
ctr images pull gcr.io/flow-container-registry/${ROLE}:alpha-v0.0.1",
```

Prepare Your Node to Start

Your nodes will need to boot at startup, and restart if they crash.

If you are running systemd you can use the service files provided by flow-go.

Find them in the Flow Go.

If you are using some other system besides Systemd, you need to ensure that the Flow container is started, the appropriate key directories are mounted into the container, and that the container will automatically restart following a crash.

The systemd files pull runtime settings from /etc/flow/runtime-config.env and any .env files under /etc/flow/conf.d. Examples of these files are also available in the github repo.

You will need to modify the runtime config file later.

Systemd

```
<Callout type="info">
If you are not using Systemd, you can skip this step
</Callout>
```

1. Ensure that you pulled the latest changes from flow-go repository via git

```
shell
Clone the repo if you haven't already done so
git clone https://github.com/onflow/flow-go
```

```
Get latest changes
cd flow-go
git pull origin master
```

2. Copy your respective systemd unit file to: /etc/systemd/system
3. Create directory sudo mkdir /etc/flow
4. Copy the runtime-conf.env file to: /etc/flow/
5. Enable your service sudo systemctl enable flow-\$ROLE.service (replace \$ROLE with your node role - eg. collection)

Docker Configuration

If you are not using Systemd, sample commands for running each Docker container are below.

Be sure to replace /path/to/data and /path/to/bootstrap with the appropriate paths you are using.

:::warning

Do not run your node using docker run command directly without a mechanism for the node to automatically restart following a crash.

:::

:::info

The actual Docker image tag can be found here for appropriate spork.

:::

System Configuration

Flow nodes create connections to other nodes on the network, which are represented as file descriptors by the OS. Depending on the default limits for your machine, you may need to increase the soft limit available to the node software.

Make sure the soft limit is at least 8192.

You can configure the ulimit for the node's docker container. See the Docker documentation for more details.

Admin Server

Each node can be configured with an admin server, which allows you to control some of the node's configuration, as well as view some of its internal state. You can find a few of the commands in the Admin Server README. Two commands to highlight are:

list-commands: which returns a list of all of the available commands for your node

set-log-level: which allows you to change the log level of your node at runtime

You can enable the admin server by passing the --admin-addr flag with an interface and port.

> **⚠️ IMPORANT:** The admin server can modify your node's configuration. DO NOT allow access to untrusted clients.

Access

shell

```
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
--ulimit nofile=8192 \
gcr.io/flow-container-registry/access:<applicable docker tag> \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--secretsdir=/data/secrets \
--execution-data-dir=/data/executiondata \
--rpc-addr=0.0.0.0:9000 \
--http-addr=0.0.0.0:8000 \
--admin-addr=0.0.0.0:9002 \
--collection-ingress-port=9000 \
--script-addr=${FLOWNETWORKEXECUTIONNODE} \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Collection

```
shell
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
--ulimit nofile=8192 \
gcr.io/flow-container-registry/collection:<applicable docker tag> \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--secretsdir=/data/secrets \
--ingress-addr=0.0.0.0:9000 \
--admin-addr=0.0.0.0:9002 \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Consensus

```
shell
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
--ulimit nofile=8192 \
gcr.io/flow-container-registry/consensus:<applicable docker tag> \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
```

```
--secretsdir=/data/secrets \
--admin-addr=0.0.0.0:9002 \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Execution

```
shell
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
--ulimit nofile=500000 \
gcr.io/flow-container-registry/execution:<applicable docker tag> \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--secretsdir=/data/secrets \
--triedir=/data/execution \
--execution-data-dir=/data/executiondata \
--rpc-addr=0.0.0.0:9000 \
--admin-addr=0.0.0.0:9002 \
--bind 0.0.0.0:3569 \
--loglevel=error
```

For execution nodes, it is recommend to increase the open files limit in your operating system. To do that, add the following to your /etc/security/limits.conf or the equivalent limits.conf for your distribution:

```
hard nofile 500000
soft nofile 500000
root hard nofile 500000
root soft nofile 500000
```

Restart your machine to apply these changes. To verify that the new limits have been applied, run:

```
ulimit -n
```

Verification

```
shell
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
```

```
--network host \
--ulimit nofile=8192 \
gcr.io/flow-container-registry/verification:<applicable docker tag>
\
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--secretsdir=/data/secrets \
--admin-addr=0.0.0.0:9002 \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Start the Node

Now that your node is provisioned and configured, it can be started.

<Callout type="warning">

Before starting your node, ensure it is registered and authorized.

</Callout>

Ensure you start your node at the appropriate time.

See Spork Process for when to start up a node following a spork.

See Node Bootstrap for when to start up a newly registered node.

Systemd

1. Check that your runtime-conf.env is at /etc/flow/runtime-conf.env
2. Update your environment variables: source /etc/flow/runtime-conf.env
3. Start your service: sudo systemctl start flow

Verify your Node is Running

Here are a few handy commands that you can use to check if your Flow node is up and running

Systemd

- To get Flow logs: sudo journalctl -u flow-YOURROLE
- To get the status: sudo systemctl status flow

shell

```
● flow-verification.service - Flow Access Node running with Docker
   Loaded: loaded (/etc/systemd/system/flow-verification.service; enabled;
             vendor preset: enabled)
   Active: active (running) since Wed 2020-05-20 18:18:13 UTC; 1 day 6h ago
     Process: 3207 ExecStartPre=/usr/bin/docker pull gcr.io/flow-container-
              registry/verification:${FLOWGONODEVERSION} (code=exited,
              status=0/SUCCESS)
    Main PID: 3228 (docker)
       Tasks: 10 (limit: 4915)
```

```
Memory: 33.0M
CGroup: /system.slice/flow-verification.service
└─3228 /usr/bin/docker run --rm -v
/var/flow/bootstrap:/bootstrap:ro -v /var/flow/data:/data:rw --rm --name
flow-go --network host gcr.io/flow-container-
registry/verification:candidate8 --
nodeid=489f8a4513d5bd8b8b093108fec00327b683db545b37b4ea9153f61b2c0c49dc -
--bootstrapdir=/bootstrap --datadir=/data/protocol --alpha=1 --bind
0.0.0.0:3569 --loglevel=error
```

Docker

- To get Flow logs: sudo docker logs flow-go
- To get the status: sudo docker ps

```
shell
$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1dc5d43385b6 gcr.io/flow-container-registry/verification:candidate8
"/bin/app --nodeid=4..." 30 hours ago Up 30 hours flow-go
```

Monitoring and Metrics

This is intended for operators who would like to see what their Flow nodes are currently doing. Head over to Monitoring Node Health to get setup.

Node Status

The metrics for the node should be able to provide a good overview of the status of the node. If we want to get a quick snapshot of the status of the node, and if it's properly participating in the network, you can check the consensuscompliancefinalizedheight or consensuscompliancesealedheight metric, and ensure that it is not zero and strictly increasing.

```
shell
curl localhost:8080/metrics | grep consensuscompliancesealedheight

HELP consensuscompliancesealedheight the last sealed height
TYPE consensuscompliancesealedheight gauge
consensuscompliancesealedheight 1.132054e+06
```

```
# past-sporks.md:
```

```
---
title: Past Spork Info
description: Information about all Flow Mainnet and Testnet sporks.
sidebarposition: 14
---
```

A JSON version of the information below can be found in
onflow/flow/sporks.json

Mainnet Sporks

Mainnet 26

```
| Spork Info
|
| :-----
-----|
| Access Node: access.mainnet.nodes.onflow.org:9000
|
| Date: Sep 25, 2024
|
| Root Height: 88226267
|
| Root Parent ID:
71052eb6e774ae5065b31c604a85af47fa16d5c53e54b9a992581c1ecf0ecfac      |
| Root State Commit:
3bba639062a723af1b44b1dfe07e795d158482f02f807f1df0b7c39edd6a8cca      |
| Git Commit: 25d9c2a9b89bac8fa003ca67928eb79b1427ea17
|
| Branch/Tag: v0.37.16-patch.1
|
| Docker Image Tag: v0.37.16-patch.1
|
```

Mainnet 25

```
| Spork Info
|
| :-----
-----|
| Access Node: access-001.mainnet25.nodes.onflow.org:9000
|
| Date: Sep 4, 2024
|
| Root Height: 85981135
|
| Root Parent ID:
bc16d37060cb408163a04afe406b9c9398a31663c839de35b92e3c3b10bcf834      |
| Root State Commit:
dead79e8f86d20ea3214735d4247b7fac1b4408e93e9b092fe0566cf40cecb9e      |
| Git Commit: feabd3a4f9baaa5d7770a312e5b8dc1bd94b1edb
|
| Branch/Tag: v0.37.10      |
| Docker Image Tag: v0.37.10
|
```

Mainnet 24

```
| Spork Info
|
| :-----
-----|
| Access Node: access-001.mainnet24.nodes.onflow.org:9000
|
| Date: Nov 8, 2023
|
| Root Height: 65264619
|
| Root Parent ID:
aace2d9b6e66067989d0f71c2efff38fe30d88da61e3d80946c7e7b4ee2bbc2f      |
| Root State Commit:
709530929e4968daff19c303ef1fc5f0a7649b3a1ce7d5ee5202056969524c94      |
| Git Commit: e63117642e34b215993d14d36622d45df249016c
|
| Branch/Tag: v0.32.7          |
| Docker Image Tag: v0.32.7
|
```

Mainnet 23

```
| Spork Info
|
| :-----
-----|
| Access Node: access-001.mainnet23.nodes.onflow.org:9000
|
| Date: Jun 20, 2023
|
| Root Height: 55114467
|
| Root Parent ID:
dcaa1469c6cd67154942c70b594bdff407ea28eda1fc9c4a81a814f02dc2efc9      |
| Root State Commit:
5586f6b9af7c0d8efa7b403dbd3a894d71a18faad5a1abda48d3dfb7dc4b017      |
| Git Commit: 0f6ea38efc91b7d27736b8b2c94091076c624796
|
| Branch/Tag: v0.31.9          |
| Docker Image Tag: v0.31.9
|
```

Mainnet 22

```
| Spork Info
|
| :-----
-----|
| Access Node: access-001.mainnet22.nodes.onflow.org:9000
|
| Date: Feb 22, 2023
|
| Root Height: 47169687
|
```

```
| Root Parent ID:  
60a976d4cc36d0a5150d3f81ede85809916d4af9eb791d9190af0a12c1fd8a17 |  
| Root State Commit:  
c9c9d3da3fe059a616b13768da2374275bd1a35f94d753ee8e41c538a3cc92d8 |  
| Git Commit: e644427a8e83e8cd2a307c40e4c8fd3066008cae  
|  
| Branch/Tag: v0.29.13 |  
| Docker Image Tag: v0.29.13  
|
```

Mainnet 21

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.mainnet21.nodes.onflow.org:9000  
|  
| Date: Jan 18, 2023  
|  
| Root Height: 44950207  
|  
| Root Parent ID:  
52004adfa7854b1a515d0d905dd5317dc7f77a8dbb56058e04dde01e53e80a92 |  
| Root State Commit:  
905c5e9a576ba2cbe49b5fe5f64ae84c2dee1bc26a3e81932e573e06a722d08a |  
| Git Commit: 7f02a642bb437b45326c4ace54a7f033b32832f8  
|  
| Branch/Tag: v0.29.6 |  
| Docker Image Tag: v0.29.6  
|
```

Mainnet 20

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.mainnet20.nodes.onflow.org:9000  
|  
| Date: Nov 2, 2022  
|  
| Root Height: 40171634  
|  
| Root Parent ID:  
f66302ac43623a87d29dfdeb08fce5d19e3af7be2e4283d468f74ee10468f248 |  
| Root State Commit:  
ec1e1cd34bb05b5abb3c4701a7f365d1dde46d9d908dc57420bde8b4a53d940a |  
| Git Commit: b9b941db9a3949db0e299a40264d852980d35ddd  
|  
| Branch/Tag: v0.28.6 |  
| Docker Image Tag: v0.28.6  
|
```

Mainnet 19

```
| Spork Info
|
| :
----- |
| Access Node: access-001.mainnet19.nodes.onflow.org:9000
|
| Date: Aug 24, 2022
|
| Root Height: 35858811
|
| Root Parent ID:
5c030d4125f8ace0e0ab8255880143190d58aca4ddb7c4720c28eaf497adcae1      |
| Root State Commit:
96ff0d9a2b7d8264ea8727d4a104d7372efcf18dc0c9111aff5c46b688eff04      |
| Git Commit: b6e9a988514d13e1e1ecd0802d7e02f9e9b1415b
|
| Branch/Tag: v0.27.4          |
| Docker Image Tag: v0.27.4
|
```

Mainnet 18

```
| Spork Info
|
| :
----- |
| Access Node: access-001.mainnet18.nodes.onflow.org:9000
|
| Date: Jun 15, 2022
|
| Root Height: 31735955
|
| Root Parent ID:
716d54edbb3d9b2ad290709a516f1ffe4290c7b7b33a49bd0480e0b193a45884      |
| Root State Commit:
5bdae9f5fb4cc5d63075547df5502e2bc3cb20707452389fb89ebbe71ecf7b68      |
| Git Commit: fdc732183233f1c577a9e529da6b453364431301
|
| Branch/Tag: v0.26.9          |
| Docker Image Tag: v0.26.9
|
```

Mainnet 17

```
| Spork Info
|
| :
----- |
| Access Node: access-001.mainnet17.nodes.onflow.org:9000
|
| Date: Apr 6, 2022
|
```

```
| Root Height: 27341470
|
| Root Parent ID:
dd7ed04e14559ed47ecc92896a5cb3dbb6b234065d9be7f816d99602238762aa      |
| Root State Commit:
113e752ec0619b63187630b4fae308ec5405a00f56f25c2de0b139e283a95b14      |
| Git Commit: 5226c35eb14890db024b9193793b0c49d1b5ad04
|
| Branch/Tag: v0.25.7          |
| Docker Image Tag: v0.25.7
|
```

Mainnet 16

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet16.nodes.onflow.org:9000
|
| Date: Feb 9, 2022
|
| Root Height: 23830813
|
| Root Parent ID:
b0e7a891682dce277a41e87e2cef52af344ac614bf70c82a5a3a801e63960e88      |
| Root State Commit:
8964d1f74c2bed2d0fbb4a366fff1fd3c39b71653e9de4d4512090798631e5f8      |
| Git Commit: c78cac3573e0548611f29df7cfa2db92203554c1
|
| Branch/Tag: v0.24.4          |
| Docker Image Tag: v0.24.4
|
```

Mainnet 15

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet15.nodes.onflow.org:9000
|
| Date: Dec 8, 2021
|
| Root Height: 21291692
|
| Root Parent ID:
8a28ddc6653a8991435dfbc95103d6be1a3b653cda3d664681215ee93112a203      |
| Root State Commit:
84d9b325d3f48f7f075eea0db6c246cb5adc759a2dd8561e461477a7ca1f7f22      |
| Git Commit: bfde54ae3324db5d18ebeaa22c70b5574a114f2e
|
| Branch/Tag: v0.23.4          |
```

```
| Docker Image Tag: v0.23.4
|
Mainnet 14

| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet14.nodes.onflow.org:9000
|
| Date: Oct 6, 2021
|
| Root Height: 19050753
|
| Root Parent ID:
ac4dbf344ce96e39e15081f1dc3fbff6dc80532e402de9a57af847d3b35df596      |
| Root State Commit:
641eb088e3ce1a01ff56df2d3a14372c65a7fef44c08799eb92cd7759d1d1d2a   |
| Git Commit: f019c1dbd778ce9f92dea61349ca36003678a9ad
|
| Branch/Tag: v0.22.9-patch-1-epoch-view-check-hotfix      |
| Docker Image Tag: v0.22.9-patch-1-epoch-view-check-hotfix
|
```

```
Mainnet 13

| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet13.nodes.onflow.org:9000
|
| Date: Sept 15, 2021
|
| Root Height: 18587478
|
| Root Parent ID:
2f708745fff4f66db88fac8f2f41d496edd341a2837d3e990e87679266e9bdb8      |
| Root State Commit:
51e3098d327df22fd005d960cb73167c83cb438c53e6c4363c07d8611ae44528   |
| Git Commit: 9535540110a4452231d044aceabab0e60f67708c
|
| Branch/Tag: v0.21.3      |
| Docker Image Tag: v0.21.3
|
```

```
Mainnet 12

| Spork Info
|
| :-----
----- |
```

```
| Access Node: access-001.mainnet12.nodes.onflow.org:9000
|
| Date: Aug 18, 2021
|
| Root Height: 17544523
|
| Root Parent ID:
f6ca04ba5c6fa6ba690a77202a9fad8d3ec30c67762ae065f0f0f53e8fed84d0      |
| Root State Commit:
b5cf1977b12de699d0e777af5be25095653735a153d1993f97ff07804b070917   |
| Git Commit: 4a0c10d74f1bcaadfdfec8c325efa411acd1a084
|
| Branch/Tag: v0.20.5          |
| Docker Image Tag: v0.20.5
|
```

Mainnet 11

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet11.nodes.onflow.org:9000
|
| Date: July 21, 2021
|
| Root Height: 16755602
|
| Root Parent ID:
de15461988000eddc6e507dc7b159dc192ee3aa72f3bd3b0e31ae9c6538399f      |
| Root State Commit:
4eb91bf34cb65b7537a6a95806f444f409308b2eaaa0ad28d1924b6cb8afa140   |
| Git Commit: 2644560c0562412a3c2209820be07f8f3f8b1846
|
| Branch/Tag: v0.19.2          |
| Docker Image Tag: v0.19.2
|
```

Mainnet 10

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet10.nodes.onflow.org:9000
|
| Date: June 23, 2021
|
| Root Height: 15791891
|
| Root Parent ID:
1d4109bbb364d5cdd94640546dd1c423d792d962284665233d541e4ade921726      |
| Root State Commit:
9b6a1a5ab52fd3d5a19ea22f09cb607bba63671311f157b3e604dd265efb851a   |
```

```
| Git Commit: 01f53ebd7df3101e337d9212736cff6ab1e0056d
| Branch/Tag: v0.18.4           |
| Docker Image Tag: v0.18.4   |
|
Mainnet 9

| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet9.nodes.onflow.org:9000
|
| Date: May 26, 2021
|
| Root Height: 14892104
|
| Root Parent ID:
265d10ba3e36ed6539fd4d7f4322735aad4997c0378d75783e471437dd83ef33    |
| Root State Commit:
fcc30a6664337ef534ad544ad7b17c5cc3b5470a8ef0d93f18573fddf6b25c4a  |
| Git Commit: 2d81520c49a8865fa686c32c508d2261155c86bc
|
| Branch/Tag: v0.17.4           |
| Docker Image Tag: v0.17.4   |
|
Mainnet 8

| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet8.nodes.onflow.org:9000
|
| Date: April 28, 2021
|
| Root Height: 13950742
|
| Root Parent ID:
faa2a3a996c6efcc3ef562fe03d797e4b19dbe00f6beab082d6d37a447044abd    |
| Root State Commit:
259836c7f74e6bbb803c9cfb516044bc701d99c2840e9b9f89609464867e7f0f  |
| Git Commit: 4e733ba2038512e9d80bcd955e67e88ba6e3ecf2
|
| Branch/Tag: v0.16.2
|
| Docker Image Tag: v0.16.2   |
|
Mainnet 7
```

```
| Spork Info
|
| -----
----- |
| Access Node: access-001.mainnet7.nodes.onflow.org:9000
|
| Date: April 7, 2021
|
| Root Height: 13404174
|
| Root Parent ID:
8b969d0babbb7d2043957b3d55a811f2c13344faa76565096d4ad901a466ecaa    |
| Root State Commit:
1247d74449a0252ccfe4fd0f8c6dd98e049417b3bfff3554646d92f810e11542   |
| Git Commit: b6f47fd23ffe31e2fe714c6bff0b17d901e210b4
|
| Branch/Tag: v0.15.3-patch.4
|
| Docker Image Tag: v0.15.3-patch.4
|
```

Releases compatible with Mainnet 7: No change from Mainnet 6

Mainnet 6

```
| Spork Info
|
| -----
----- |
| Access Node: access-001.mainnet6.nodes.onflow.org:9000
|
| Date: Mar 10, 2021
|
| Root Height: 12609237
|
| Root Parent ID:
c68e63ca5b6f7ff61ef2b28d7da528c5b677b0f81f2782f067679c108d77932b    |
| Root State Commit:
ddfedbfcaa2d858e6a8e3b142381a91b289f50e45622f5b5a86ac5c00ce61bf11   |
| Git Commit: c887bd343ca7db6351690007b87bec40d39d7b86
|
| Branch/Tag: v0.14.9
|
| Docker Image Tag: v0.14.9
|
```

Releases compatible with Mainnet 6:

Network Implementations

Flow Go: <https://github.com/onflow/flow-go/releases/tag/v0.14.9>

Emulator: <https://github.com/onflow/flow-emulator/releases/tag/v0.16.1>

Emulator v0.16.1 is distributed in Flow CLI v0.15.0:
<https://github.com/onflow/flow-cli/releases/tag/v0.15.0>

SDK Compatibility

Flow Go SDK

Minimum version: <https://github.com/onflow/flow-go-sdk/releases/tag/v0.16.1>

Recommended version: <https://github.com/onflow/flow-go-sdk/releases/tag/v0.16.1>

FCL (Flow Client Library)

Minimum version: v0.0.66

Recommended version: v0.0.67

While FCL v0.0.67 is not strictly necessary to use Mainnet 6, we strongly recommend upgrading in order to adapt to wallet improvements that were introduced in v0.0.67.

Mainnet 5

```
| Spork Info
|
| :-----
----- | 
| Access Node: access-001.mainnet5.nodes.onflow.org:9000
|
| Date: Feb 17, 2021
|
| Root Height: 12020337
|
| Root Parent ID:
9131733835702b0d6321088bddb4642a4964bb5c630440ccb0de47bdb371d1a
|
| Root State Commit:
54bef048a6c5574ef4eb452dd2698aeb2fe5eca6edd536aca6d0bc631c2daaa9
|
| Git Commit: 027569a9d76e41b1140b189fa1b9187c711ab241
|
| Branch/Tag: v0.14.1 for Access, Verification, Collection and v0.14.2
for Consensus and Execution |
| Docker Image Tag: v0.14.1 / v0.14.2
|
```

Mainnet 4

```
| Spork Info
|
```

```
| :-----  
----- |  
| Access Node: access-001.mainnet4.nodes.onflow.org:9000  
|  
| Date: Dec 16, 2020  
|  
| Root Height: 9992020  
|  
| Root Parent ID:  
691e35a4ac4d0d47e1be1ec81512ac9f6cdd04545b908fad1d6ceea58c76b560 |  
| Root State Commit:  
0011fda57f2f3aaa8e6bcc1e1deea9778a9543252f8b65bbd4ebca3687789420 |  
| Git Commit: f4a73c7f20109209e9e2e999cf50fcf1ec41241b  
|  
| Branch/Tag: v0.13.1  
|  
| Docker Image Tag: v0.13.1  
|
```

Mainnet 3

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.mainnet3.nodes.onflow.org:9000  
|  
| Date: Dec 9, 2020  
|  
| Root Height: 9737133  
|  
| Root Parent ID:  
116751c904a7f868cd6e8c90522fdbd70fe826db6886b830338c68c6339df3e7 |  
| Root State Commit:  
1d2c91e801d0560024848a0c981e03120efc372436ada5f7909c4d44d4600f04 |  
| Git Commit: badd5887512b955e7aa18b4e73dae980ca72fa22  
|  
| Branch/Tag: v0.12.6  
|  
| Docker Image Tag: v0.12.6  
|
```

Mainnet 2

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.mainnet2.nodes.onflow.org:9000  
|
```

```
| Date: Nov 13, 2020
|
| Root Height: 8742959
|
| Root Parent ID:
b35fdb189d21a95df7f19f941786f748d9854a8b93b1e555b51cda7d9f53a6e1    |
| Root State Commit:
d6a25be552ed93213df0ffc2e8c7f39f6401c04cbf22bac7a4b84d3c9493f005  |
| Git Commit: 4ef68efb935c0e3393ae3966752ece5e7739bab4
|
| Branch/Tag: v0.11.1
|
| Docker Image Tag: v0.11.1
|
```

Mainnet 1

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.mainnet1.nodes.onflow.org:9000
|
| Date: Oct 13, 2020
|
| Root Height: 7601063
|
| Root Parent ID:
ablee18b6e1c0ee11cc021c26a17c694c627699a576e85f7013cd743bdb7877    |
| Root State Commit:
6e1adf15689eaf5ea6859bcdd0b510f5eb4c34dac878d8577b3f65bc20c3f312  |
| Git Commit: 114d45436e7d9052e910c98a1e40f730e3fd12d7
|
| Branch/Tag: v0.10.1
|
| Docker Image Tag: v0.10.1
|
```

Candidate 9

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.candidate9.nodes.onflow.org:9000
|
| Date: Sep 25, 2020
|
| Root Height: 6483246
|
```

```
| Root Parent ID:  
9131733835702b0d6321088bddb4642a4964bb5c630440ccb0de47bdbc371d1a |  
| Root State Commit:  
90c6f406f5d21880d525ad4702cb249509b85e7f745db2de67e9fe541a56da4c |  
| Git Commit: v0.9.3  
|  
| Branch/Tag: v0.9.3  
|  
| Docker Image Tag:  
|
```

Candidate 8

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.candidate8.nodes.onflow.org:9000  
|  
| Date: Sep 9, 2020  
|  
| Root Height: 4972987  
|  
| Root Parent ID:  
5bc2b0900a5138e39d9209a8fe32e14b3e5c884bd36d2a645620f746b7c8bd47 |  
| Root State Commit:  
6b9161a225b087a461ec95e710fdf4e73f6d6c9401ebf066207a021dcfd4ce5e |  
| Git Commit:  
|  
| Branch/Tag:  
|  
| Docker Image Tag:  
|
```

Candidate 7

```
| Spork Info  
|  
| :-----  
----- |  
| Access Node: access-001.candidate7.nodes.onflow.org:9000  
|  
| Date: Aug 24, 2020  
|  
| Root Height: 4132133  
|  
| Root Parent ID:  
28f4f495aad016b519acf27fc9d9a328f6a4009807480e36e2df780eecccd99bc |  
| Root State Commit:  
001d173bf9c7f71684da89bff72b3ee582b39a69c7929360230faf73735c17 |
```

```
| Git Commit: f811f8cd49369ae2bc559e0fbb781aff129484f5
| Branch/Tag: candidate7
|
| Docker Image Tag: v0.7.2
|
```

Candidate 6

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.candidate6.nodes.onflow.org:9000
|
| Date: Aug 18, 2020
|
| Root Height: 3187931
|
| Root Parent ID:
2ff5f7424a448943a153001d2f0869d4fac330906ecb8e17b7ef7fe50e4c7b36    |
| Root State Commit:
bd7f16dc5ef5eced849ab5f437547c14c1907059e1ecf89a942d0521166c5cbb |
| Git Commit: b30c48008c0ec1cc8ecb750aeb9ff9f3d712681d
|
| Branch/Tag:
|
| Docker Image Tag:
|
```

Candidate 5

```
| Spork Info
|
| :-----
----- |
| Access Node: access-001.candidate5.nodes.onflow.org:9000
|
| Date: Jul 28, 2020
|
| Root Height: 2033592
|
| Root Parent ID:
a0efffb2beb1500419ae4f7c6e49bfbbe3a4d1d1c201bf925ccaec467ea30e91    |
| Root State Commit:
0190d417a26b9870f5bb2cf408ad31985b3aa7e57f6ababa6e543f0f90b99dcd |
| Git Commit: cd876653d20b398952af4002701a0ae2800fd5f2
|
| Branch/Tag:
|
```

```
| Docker Image Tag:  
|  
---  
  
Candidate 4  
  
| Spork Info  
|  
| :-----  
-----|  
| Access Node: access-001.candidate4.nodes.onflow.org:9000  
|  
| Date: Jul 14, 2020  
|  
| Root Height: 1065711  
|  
| Root Parent ID:  
68c2bbe68524b50f5d689bc2ac7ad2dd70e88ed7dd15ad6c3cdf6ea314cb1aa3 |  
| Root State Commit:  
c05086e4d1d428d3b9af5bd8b81d8780054f783ef4eec3ca28b491202e9ac696 |  
| Git Commit: b9b197280d6590576f1ef183bc3d04d41d6be587  
|  
| Branch/Tag:  
|  
| Docker Image Tag:  
|  
---  
  
Testnet Sporks  
  
Devnet 52  
  
| Spork Info  
|  
| :-----  
-----|  
| Date: Sept 24th, 2024  
|  
| Root Height: 218215349  
|  
| Root Parent ID:  
20dd925a750399493cf7455f199c32c952e8010a6c0b4424dba00a193fa18e44 |  
| Root State Commit:  
b0498700398cdc8c0c9368cc2f82fde62e8fe4b06e9c8af6c9bb619ab499e6c3 |  
| Git Commit: 25d9c2a9b89bac8fa003ca67928eb79b1427ea17  
|  
| Branch/Tag: v0.37.16-patch.1  
|  
| Docker Image Tag: v0.37.16-patch.1  
|
```

Devnet 51

```
| Spork Info
|
| :-----
-----|
| Date: Aug 14th, 2024
|
| Root Height: 211176670
|
| Root Parent ID:
c92e07e5d4fbb3a64e0091085a190a4a1119bfc628c71efe513e373dc0482f5a      |
| Root State Commit:
c3af77992f253f4dcfeac808912ff68e6f10923aa3fc4541a2e39eb9786c9eb3   |
| Git Commit: eeac47931cd6837ec6e29c4c0480609238959ccd
|
| Branch/Tag: v0.37.1
|
| Docker Image Tag: v0.37.1
|
```

Devnet 50

```
| Spork Info
|
| :-----
-----|
| Date: May 20th, 2024
|
| Root Height: 185185854
|
| Root Parent ID:
cc4800bf44bc07864d156f829cfda2ae1964b5e103de7b9fa1bd879f9e92c10d
|
| Root State Commit:
6a0ae7bf43660e813ee9c2d654f00476ac1bdc357ff47ad11f0e52fc1700d62f
|
| Git Commit: 0585789483c4f5ea423bb11afcfe862c9a99711e
|
| Branch/Tag: v0.33.23-failure-mode-revert-patch
|
| Docker Image Tag: v0.33.23-failure-mode-revert-patch
|
```

Devnet 49

```
| Spork Info
|
| :-----
-----|
| Date: Nov 2nd, 2023
|
| Root Height: 129578013
|
```

```
| Root Parent ID:  
91b039c1a5caf25776948270a6355017b8841bfb329c87460bfc3cf5189eba6f |  
| Root State Commit:  
e10d3c53608a1f195b7969fbc06763285281f64595be491630a1e1bdfbe69161 |  
| Git Commit: fce4ae31c8c90c6a21de9856a319c379bb797fc5  
|  
| Branch/Tag: v0.32.6-patch.1  
|  
| Docker Image Tag: v0.32.6-patch.1  
|
```

Devnet 48

```
| Spork Info  
|  
| :-----  
-----|  
| Date: Aug 4th, 2023  
|  
| Root Height: 127720466  
|  
| Root Parent ID:  
2b30e75bd857f898456dcba296bea4b8bc8001cab7062eeee9e47876411b36d76 |  
| Root State Commit:  
f2e5ebfca2fd519e49f7bd85bea81e92eeaa85a705b3376d8534e9c9649da710 |  
| Git Commit: 692969b1718b3d21f95ee7f66e5061623d99e599  
|  
| Branch/Tag: v0.32.3  
|  
| Docker Image Tag: v0.32.3  
|
```

Devnet 47

```
| Spork Info  
|  
| :-----  
-----|  
| Date: Aug 4th, 2023  
|  
| Root Height: 113167876  
|  
| Root Parent ID:  
dbd59a00503707c8bc3c5fb3bcc7bd243da4bf8e24c86b6a496505e275b85311 |  
| Root State Commit:  
42006aedbf9fa9fc949de8347b55166df77d8742bf5f273266a8dfcdf2b836b |  
| Git Commit: db1c2584d6d8359b7ccf733c16bd5e1b9385c9bc  
|  
| Branch/Tag: v0.31.13  
|  
| Docker Image Tag: v0.31.13  
|
```

Devnet 46

```
| Spork Info
|
| :-----
-----|
| Date: Jun 8th, 2023
|
| Root Height: 105155067
|
| Root Parent ID:
daced0cdeed95cf143320db50ac904ed17dabe04898e988264af2a71e0d1ca48      |
| Root State Commit:
e8c39b7a1672cb3f5f70da6f1d71a3a0322d3d6c3d7ebf8092ae2ae40d12c30b   |
| Git Commit: 3d4159e93c92cc6331e69708b8c3270d40c09c5f
|
| Branch/Tag: v0.31.4
|
| Docker Image Tag: v0.31.4
|
```

Devnet 45

```
| Spork Info
|
| :-----
-----|
| Date: Jun 7th, 2023
|
| Root Height: 105032150
|
| Root Parent ID:
71df56106ee1492c055a60e2d951a6a8d1b7d1483b903f10146cec912e793e82      |
| Root State Commit:
0e2e053ca4436a4881f65acc031e12820b5260ef616446950650a1bc8fc9be2f   |
| Git Commit: 8d32e5ea087fcceb0d1aa923c56be1d5a2d6538a
|
| Branch/Tag: v0.31.2
|
| Docker Image Tag: v0.31.2
|
```

Devnet 44

```
| Spork Info
|
| :-----
-----|
| Date: Apr 24th, 2023
|
| Root Height: 100675451
|
| Root Parent ID:
298854580771edc6ec2e2bbc8da3990ff7f746e2fc5cfe16e550d577bdd4bc5b      |
```

```
| Root State Commit:  
2a062531331b8214de4e900a26ff0f76c578481b4ce22b4b5d8e55bd9535abda |  
| Git Commit: b5b65c9d43bf6bd12766eef06e870990c3963b7c  
|  
| Branch/Tag: v0.30.6  
|  
| Docker Image Tag: v0.30.6  
|
```

Devnet 43

```
| Spork Info  
|  
| :-----  
-----|  
| Date: Apr 12th, 2023  
|  
| Root Height: 99452067  
|  
| Root Parent ID:  
0e6bae31b2f34ffb0d64d4e1a33c2fbcebd80c936d77d70d2f5dd9321dc92393 |  
| Root State Commit:  
7cd69bb0a4448566dce42808efa8b1d03f322135f0bfce6f18037fc1294984c |  
| Git Commit: d55ca8f48167e9669bdb1dc3173253936863e31e  
|  
| Branch/Tag: v0.30.3  
|  
| Docker Image Tag: v0.30.3  
|
```

Devnet 42

```
| Spork Info  
|  
| :-----  
-----|  
| Date: Apr 12th, 2023  
|  
| Root Height: 99444465  
|  
| Root Parent ID:  
a3788d5c0ff1c45db38bca2625bf18bfc562e1bfd9eb7dc83ef080be31e697 |  
| Root State Commit:  
a6e5bfc16a39109ef86caad4dab77ec3752680ef3813c3449f239d84fddc5aa1 |  
| Git Commit: d55ca8f48167e9669bdb1dc3173253936863e31e  
|  
| Branch/Tag: v0.30.3  
|  
| Docker Image Tag: v0.30.3  
|
```

Devnet 41

```
| Spork Info
|
| :-----
-----|
| Date: Jan 30th, 2023
|
| Root Height: 93156994
|
| Root Parent ID:
26ff2f7f2948a05c63e723eb42946565809b47dbef87079a8f0bae0cc36a0478      |
| Root State Commit:
08d712b4f7ed838d53b8699e2fd94d0ad010c4b1fa45735b6e80083ff8ef08ff  |
| Git Commit: a7f2cd0ddd9fc7e1e56187327a84fec9efdc3c9d
|
| Branch/Tag: v0.29.8
|
| Docker Image Tag: v0.29.8
|
```

Devnet 40

```
| Spork Info
|
| :-----
-----|
| Date: Jan 23rd, 2023
|
| Root Height: 92473965
|
| Root Parent ID:
7c9812f414d9e9795c1cd7dbd27fc45baf880391452e0e948aaa80ba86dfc77d      |
| Root State Commit:
0068a04843d2c8e007086abaaee90c8c8cae8aa78f240048cd4d43aeb0376d0b  |
| Git Commit: 7f02a642bb437b45326c4ace54a7f033b32832f8
|
| Branch/Tag: v0.29.6
|
| Docker Image Tag: v0.29.6
|
```

Devnet 39

```
| Spork Info
|
| :-----
-----|
| Date: Jan 4th, 2023
|
| Root Height: 90595736
|
| Root Parent ID:
3d09b9703019b40a065787fff3dd62e28eafa5efcfb69efbc2d713d73034cf38      |
| Root State Commit:
af47ff2a8e73efd51f0cbe7f572eefda25a66e77f1e6b9c4f3f4e7cdc46568f  |
```

```
| Git Commit: e1c172aaee7da9e33828429757b44f51e59368a2
| Branch/Tag: v0.29.3
|
| Docker Image Tag: v0.29.3
|
```

Devnet 38

```
| Spork Info
|
| :-----
-----|
| Date: Oct 19th, 2022
|
| Root Height: 83007730
|
| Root Parent ID:
1b914363c9a34c46b93974adeefb10c546578d7f3f4ac9291e01d746d2c84226 |
| Root State Commit:
79df428bc27f22d38c233777610d93d33e180f23cfbc640a95d144a508e0f080 |
| Git Commit: 3aae289f8f390f58ac481fd694254ea0e48960a8
|
| Branch/Tag: v0.28.4
|
| Docker Image Tag: v0.28.4
|
```

Devnet 37

```
| Spork Info
|
| :-----
-----|
| Date: Aug 10th, 2022
|
| Root Height: 76159167
|
| Root Parent ID:
8f379a8a86f28c7adef276890874b17786cecf5efb9e71734b0a780bd38660a0 |
| Root State Commit:
7d1fe692ea2f857568dec54ddece094a68a9ba8ff8dd0de0664e6f73abb90dd8 |
| Git Commit: 959911cabd50e1a11be45e89726952a90f1a9c22
|
| Branch/Tag: v0.27.2
|
| Docker Image Tag: v0.27.2
|
```

Devnet 36

```
| Spork Info
|
```

```
| :-----  
-----|  
| Date: July 27th, 2022  
|  
| Root Height: 74786360  
|  
| Root Parent ID:  
9d9113ca8daa4f3be72e949ea9ead2d4b11db222b2ccd0dc897186ee5f8703ab |  
| Root State Commit:  
aa2e12033129f27b5320fb973cf109f562db3ad7ddaf50dbfe9d8cc195a7ac0f |  
| Git Commit: 13970fcae812bc04487422922f73eab39f53935c  
|  
| Branch/Tag: v0.26.17  
|  
| Docker Image Tag: v0.26.17  
|
```

Devnet 35

```
| Spork Info  
|  
| :-----  
-----|  
| Date: June 9th, 2022  
|  
| Root Height: 70072575  
|  
| Root Parent ID:  
cd76fd9d5ceb1b98b5f30b0108f40836593c533ffe32eef0c6f3ed6d50bf645b |  
| Root State Commit:  
d933d73f48c9371f0a00ab7fffc1ed0daf5ba9e520d2d539e6b9494920c5ffd91 |  
| Git Commit: 07057c8d5fe2f09bbe5a9a3f8de6209346d910d0  
|  
| Branch/Tag: v0.26.6  
|  
| Docker Image Tag: v0.26.6  
|
```

Devnet 34

```
| Spork Info  
|  
| :-----  
-----|  
| Date: Apr 4th, 2022  
|  
| Root Height: 64904846  
|  
| Root Parent ID:  
0cbb13d9e7ed7092b5f20d0a20a79d1cae96fd796fb2bc569b27ce49cc97d97e |  
| Root State Commit:  
f2e77e16628543e285be8bac677db06e17b37f49e53a107af1e6bf99fbc17b30 |  
| Git Commit: 5226c35eb14890db024b9193793b0c49d1b5ad04  
|
```

```
| Branch/Tag: v0.25.7
|
| Docker Image Tag: v0.25.7
|
| Devnet 33
|
| Spork Info
|
| :-----
----- |
| Date: Feb 7th, 2022
|
| Root Height: 59558934
|
| Root Parent ID:
099fbb7b645c3441ea830746ed67059bcc1091c88ff3fd78b331137cf917d15f      |
| Root State Commit:
5b5578cb4ef6c34818ce2cf9969720c7f17681f41dfffa3a9e361935140d7c8e |  
| Git Commit: bd3dca7bf20914f5c019a325b6939bbb662aa131
|
| Branch/Tag: v0.24.3
|
| Docker Image Tag: v0.24.3
|
| Devnet 32
|
| Spork Info
|
| :-----
----- |
| Date: Dec 6th, 2021
|
| Root Height: 53376277
|
| Root Parent ID:
f900db2ccff33f3bf353c8bf28ece0a9d2650f2805b23ddb7893e296774a5457      |
| Root State Commit:
843ea0b5498342dcf960376585eeaf8ebe008df3b03325351408a100cb830c |  
| Git Commit: be20371fa8c5044a4e25e5629bbca91f1ed19731
|
| Branch/Tag: v0.23.3
|
| Docker Image Tag: v0.23.3
|
| Devnet 31
|
| Spork Info
|
| :-----
```

```
| Date: Nov 5th, 2021
|
| Root Height: 50540412
|
| Root Parent ID:
cfebaa6b8c19ec48a27eaa84c3469b23255350532b5ea4c7e4c42313386c07b6 | 
| Root State Commit:
31a2b1eb05a6acb91560970e46b8f3f3171747245eae1ad2bf40290ce773806e |
| Git Commit: 3d060b90264d59ccd38b4500ae0cd6d72036cf4
|
| Branch/Tag: v0.23-testnet
|
| Docker Image Tag: v0.23.1
|
```

Devnet 30

```
| Spork Info
|
| :-----
----- |
| Date: Oct 6th, 2021
|
| Root Height: 47330085
|
| Root Parent ID:
0ec9172a21f84cbae15761ae4d59ab4a701f9d4fd15cd0632715befc8cf205cf | 
| Root State Commit:
e280f972c72c6b379ec3d4a7173953e596704d8d72f384ef1637d2f4f01ff901 |
| Git Commit: f019c1dbd778ce9f92dea61349ca36003678a9ad
|
| Branch/Tag: v0.22.8-patch-1-scripts-and-errors
|
| Docker Image Tag: v0.22.8-patch-1-scripts-and-errors
|
```

Devnet 29

```
| Spork Info
|
| :-----
----- |
| Date: Oct 5th, 2021
|
| Root Height: 47242826
|
| Root Parent ID:
00f745576222a1e7e15ee79974b4b3eadd760fb4f56e846cfaef3cb5ea59d50 | 
| Root State Commit:
3c5bfb88a3fa184c9981e7677d1f2a2cd0a4eaa581bb2a7867b7b023ae015f38 |
| Git Commit: e1659aebc39b4a15c68112227fc8c32788a798b6
|
| Branch/Tag: v0.22.8
|
```

```
| Docker Image Tag: v0.22.8
|
| Devnet 28
|
| Spork Info
|
| :-----
----- |
| Date: Sept 22, 2021
|
| Root Height: 45889254
|
| Root Parent ID:
1518c15f7078cb8fd4c636b9fc15ee847ac231a8631ed59a0b8c9d4a182fb5b2      |
| Root State Commit:
296001ee05ce3e6c6616ad9bcd20e6d93d02c91354e57fff6054cff44c5afa3      |
| Git Commit: a979f4d25a79630581f6b350ad26730d4012cad4
|
| Branch/Tag: v0.21.6
|
| Docker Image Tag: v0.21.6
|
```

Devnet 27

```
| Spork Info
|
| :-----
----- |
| Date: Sept 14, 2021
|
| Root Height: 45051578
|
| Root Parent ID:
8525ac24717b5f42e29172f881e9a7439235e4cd443a8a59494dbecf07b9376a      |
| Root State Commit:
b3ef3b039f722130009b658e3f5faee43b3b9202fec2e976907012994a8fc9be      |
| Git Commit: 4f903f1d45e6f8c997a60de47de62a74ede3c2e4
|
| Branch/Tag: v0.21.3
|
| Docker Image Tag: v0.21.3
|
```

Devnet 26

```
| Spork Info
|
| :-----
----- |
| Date: Aug 11, 2021
|
```

```
| Root Height: 41567027
|
| Root Parent ID:
ea465f33266be26b21c82ec728c75cc0dcbb022405d83c44ed0082b0df9aa81d      |
| Root State Commit:
9459485a2a640b1bdc3066916a9a46dc20bf2528a03f06ddc541d34444c3264c      |
| Git Commit: 781ec414b892e2ccf7034aa263b5b19d97f82031
|
| Branch/Tag: v0.20.4
|
| Docker Image Tag: v0.20.4
|
```

Devnet 25

```
| Spork Info
|
| :-----
----- |
| Date: July 20, 2021
|
| Root Height: 39272449
|
| Root Parent ID:
2824828e7c87e1a89bc94daca9497625f1a35c8f9fc555f52d1f8b475179e125      |
| Root State Commit:
bd50708e7808be0d43725a9eae6039558b32b679c5a584a2e6103bb027dad7eb      |
| Git Commit: 2644560c0562412a3c2209820be07f8f3f8b1846
|
| Branch/Tag: v0.19.2
|
| Docker Image Tag: v0.19.2
|
```

Devnet 24

```
| Spork Info
|
| :-----
----- |
| Date: June 22, 2021
|
| Root Height: 36290422
|
| Root Parent ID:
8d6fad91536ec750d1bb44ebb59e030af2dca49d41c104e45f4f82433184e663      |
| Root State Commit:
fdfd7a9fff481256972095ffc2c6cba300128f41d8c6aa971985de29427eb39d      |
| Git Commit: 37d9ae7a309bf7d21063f57fce008d04828d4840
|
| Branch/Tag: v0.18.3
|
| Docker Image Tag: v0.18.3
|
```

Devnet 23

```
| Spork Info
|
| :-----
----- |
| Date: May 25, 2021
|
| Root Height: 33257098
|
| Root Parent ID:
dafa3e7a9a93de8ceb0a3acd09b2cce4ad0e5d7ea8fe4237f27c4503e5ad416c      |
| Root State Commit:
decf340a722ba136c94ece029e9333d9fbef216482cd1c81a274365e2abd6688   |
| Git Commit: fef838147fa70c94a254183e23e7e79f0d412ef6
|
| Branch/Tag: v0.17.3
|
| Docker Image Tag: v0.17.3
|
```

Devnet 22

```
| Spork Info
|
| :-----
----- |
| Date: Apr 27, 2021
|
| Root Height: 30171528
|
| Root Parent ID:
46c7c675f2ed413532009a6a6ecaa566d360a806e033693520d7add147ca89ea
|
| Root State Commit:
d88edf1b2a07dba9ef48214dbffb743c0b2e01f7c74e3e14d828ac076d30f1a6   |
| Git Commit: 8331b78d99eddea405076e3b6a7839ec5f6ea209
|
| Branch/Tag: v0.16.2
|
| Docker Image Tag: v0.16.2
|
```

Devnet 21

```
| Spork Info
|
| :-----
----- |
| Date: Mar 30, 2021
|
| Root Height: 26935025
|
```

```
| Root Parent ID:  
3f92949a68c577b6d7d17267f00fc47304d1c44052f0c9a078b63fd344f636dd |  
| Root State Commit:  
21f48748d35178bd6ad89f6324321fa9d123ee7a07b07826acbce887260b2c70 |  
| Git Commit: 8331b78d99eddea405076e3b6a7839ec5f6ea209  
|  
| Branch/Tag: v0.15.3-patch.1  
|  
| Docker Image Tag: v0.15.3-patch.1  
|
```

Devnet 20

```
| Spork Info  
|  
| :-----  
----- |  
| Date: Mar 9, 2021  
|  
| Root Height: 25450390  
|  
| Root Parent ID:  
febe212117a83f7f70ed6a5af285ff03332f81a1120ab2c306560c4cb42672f7 |  
| Root State Commit:  
828259e9cb895f7f8a7306debbe6de524500db65cb6b80e27b2db9040513b04e |  
| Git Commit: b9b197280d6590576f1ef183bc3d04d41d6be587  
|  
| Branch/Tag: v0.14.19  
|  
| Docker Image Tag: v0.14.19  
|
```

Devnet 19

```
| Spork Info  
|  
| :-----  
----- |  
| Date: Feb 3, 2021  
|  
| Root Height: 16483518  
|  
| Root Parent ID:  
4220911048404bd3a7733ab6219531a5945dc20020f86869bcd6422c3a6e3f76 |  
| Root State Commit:  
f6eea9b652a7df433b80d8d647ba414aabcf03153dd1cba33048c27ce888097 |  
| Git Commit: eb11ae095df6db6e856b1a8e824f03ce4c713b19  
|  
| Branch/Tag: v0.14.0  
|  
| Docker Image Tag: v0.14.0  
|
```

```
---  
Devnet 18  
  
| Spork Info  
|  
| :-----  
----- |  
| Date: Dec 11, 2020  
|  
| Root Height: 17756122  
|  
| Root Parent ID:  
52ace0d0c8d4e574213fe98e19a5043f215bee992659bd1ef35c5758acf54d1b |  
| Root State Commit:  
b1412c9d453b1c10d3ace6111e00ac804d564ba3d3295002bedb077685c7da73 |  
| Git Commit: 115cce9cae920ee5bc309537cf43a9b152a1cf9  
|  
| Branch/Tag: v0.13.0  
|  
| Docker Image Tag: v0.13.0  
  
---  
Devnet 17  
  
| Spork Info  
|  
| :-----  
----- |  
| Date: Nov 27, 2020  
|  
| Root Height: 16483518  
|  
| Root Parent ID:  
96d452f9d2a15fafab720f1809fc490b019b23e42c885590e62085e48b8c2b6b |  
| Root State Commit:  
0b45f6055a1f09b413d098997cc0e7c9a0ef19eac8b4d294085b0e2f436c6bff |  
| Git Commit: ef54713595d0fe1e0bb4b14e9469c4a64dfaf6e7  
|  
| Branch/Tag: v0.12.1  
|  
| Docker Image Tag: v0.12.1  
  
# protocol-state-bootstrap.md:  
  
---  
title: Protocol State Bootstrapping  
description: How to bootstrap a new or existing node  
---
```

When a node joins the network, it bootstraps its local database using a trusted initialization file, called a Root Snapshot.

Most node operators will use the Spork Root Snapshot file distributed during the spork process.

This page will explain how the bootstrapping process works and how to use it in general.

For guides covering specific bootstrapping workflows, see:

- Node Bootstrap for bootstrapping a newly joined node.
- Reclaim Disk for bootstrapping from a recent snapshot to recover disk space.

```
<Callout type="info">
```

This page covers only Protocol State bootstrapping and applies to Access, Collection, Consensus, & Verification Nodes.

Execution Nodes also need to bootstrap an Execution State database, which is not covered here.

```
</Callout>
```

Node Startup

When a node starts up, it will first check its database status.

If its local database is already bootstrapped, it will start up and begin operating.

If its local database is not already bootstrapped, it will attempt to bootstrap using a Root Snapshot.

There are two sources for a non-bootstrapped node to obtain a Root Snapshot:

1. Root Snapshot file in the bootstrap folder
2. Dynamic Startup flags, which will cause the node to download a Root Snapshot from a specified Access Node

The node software requires that only one of the above options is provided.

Using a Root Snapshot File

```
<Callout type="info">
```

If your node already has a bootstrapped database, the Root Snapshot file will be ignored. If both a Root Snapshot and Dynamic Startup flags are present, the node will not startup.

```
</Callout>
```

Using a Root Snapshot file is more flexible but more involved for operators compared to Dynamic Startup.

A file in \$BOOTDIR/public-root-information named root-protocol-state-snapshot.json will be read and used as the Root Snapshot for bootstrapping the database.

Instructions

1. Obtain a Root Snapshot file (see below for options)
2. Ensure your node is stopped and does not already have a bootstrapped database.
3. Move the Root Snapshot file to `$BOOTDIR/public-root-information/root-protocol-state-snapshot.json`, where `$BOOTDIR` is the value passed to the `--bootstrapdir` flag.
4. Start your node.

Obtain Root Snapshot File using Flow CLI

Flow CLI supports downloading the most recently sealed Root Snapshot from an Access Node using the `flow snapshot save` command.

When using this method:

- ensure you connect to an Access Node you operate or trust
- ensure you use the `--network-key` flag so the connection is encrypted

Obtain Root Snapshot File from Protocol database

If you have an existing node actively participating in the network, you can obtain a Root Snapshot using its database.

1. Obtain a copy of the Flow util tool and ensure it is in your `$PATH`. This tool is distributed during sporks, or you can build a copy from here.
2. Stop the existing node.
3. Construct a Root Snapshot using the util tool. The tool will print the JSON representation to `STDOUT`, so you can redirect the output to a file.

Replace `$DATADIR` with the value passed to the `--datadir` flag. You can specify the desired reference block for the snapshot.

Retrieve the snapshot for the latest finalized block:

```
sh
util read-protocol-state snapshot -d $DATADIR --final > latest-finalized-snapshot.json
```

Retrieve the snapshot for a specific finalized block height:

```
sh
util read-protocol-state snapshot -d $DATADIR --height 12345 > specific-height-snapshot.json
```

Using Dynamic Startup

Dynamic Startup is a startup configuration where your node will download a Root Snapshot and use it to bootstrap its local database.

Dynamic Startup is designed for nodes which are newly joining the network and need to bootstrap from within a specific epoch phase, but can be used for other use-cases.

```
<Callout type="info">
```

If your node already has a bootstrapped database, Dynamic Startup flags will be ignored. If both a Root Snapshot and Dynamic Startup flags are present, the node will not startup.

</Callout>

When using Dynamic Startup, we specify:

1. An Access Node to retrieve the snapshot from.
2. A target epoch counter and phase to wait for.

After startup, your node will periodically download a candidate Root Snapshot from the specified Access Node.

If the Root Snapshot's reference block is either within or after the specified epoch phase, the node will bootstrap using that snapshot. Otherwise the node will continue polling until it receives a valid Root Snapshot.

See the Epochs Schedule for additional context on epoch phases.

Specifying an Access Node

Two flags are used to specify which Access Node to connect to:

- --dynamic-startup-access-address - the Access Node's secure GRPC server address
- --dynamic-startup-access-publickey - the Access Node's networking public key

Select an Access Node you operate or trust to provide the Root Snapshot, and populate these two flags.

For example, to use the Access Node maintained by the Flow Foundation for Dynamic Startup, specify the following flags:

shell ExampleDynamicStartupFlags

```
... \
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \
--dynamic-startup-access-
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae
```

Specifying an Epoch Phase

Two flags are used to specify when to bootstrap:

- --dynamic-startup-epoch-phase - the epoch phase to start up in (default EpochPhaseSetup)
- --dynamic-startup-epoch - the epoch counter to start up in (default current)

> You can check the current epoch phase of the network by running this script. Alternatively, you can also check the current epoch phase here under Epoch Phase.

Bootstrapping Immediately

If you would like to bootstrap immediately, using the first Root Snapshot you receive, then specify a past epoch counter:

```
shell ExampleDynamicStartupFlags  
... \  
--dynamic-startup-epoch-phase=1
```

You may omit the --dynamic-startup-epoch-phase flag.

Instructions

Example 1

Use Dynamic Startup to bootstrap your node at the Epoch Setup Phase of the current epoch (desired behaviour for newly joining nodes):

1. Ensure your database is not already bootstrapped, and no Root Snapshot file is present in the \$BOOTSTRAPDIR folder.
2. Add necessary flags to node startup command.

For example, using the Flow Foundation Access Node:

```
sh  
... \  
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \  
--dynamic-startup-access-  
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295  
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae
```

3. Start your node.

Example 2

Use Dynamic Startup to bootstrap your node immediately, using the most recent Root Snapshot:

1. Ensure your database is not already bootstrapped, and no Root Snapshot file is present in the \$BOOTSTRAPDIR folder.
2. Add necessary flags to node startup command.

For example, using the Flow Foundation Access Node:

```
sh  
... \  
--dynamic-startup-access-address=secure.mainnet.nodes.onflow.org:9001 \  
--dynamic-startup-access-  
publickey=28a0d9edd0de3f15866dfe4aea1560c4504fe313fc6ca3f63a63e4f98d0e295  
144692a58ebe7f7894349198613f65b2d960abf99ec2625e247b1c78ba5bf2eae \  
--dynamic-startup-epoch=1
```

3. Start your node.

```
# reclaim-disk.md:
```

```
---  
title: Managing disk space  
description: How to manage the node disk space  
---
```

As the chain advances, nodes receive chain data and store it on disk. Hence, the disk usage of a node keeps increasing gradually over time.

In addition to this, currently nodes also experience an intermittent 30-35% spike in disk usage caused by the compaction process of the Badger database used by the node software.

> The spikes will be eliminated once the Badger database is replaced by the Pebble database in the future.

Hence, as a node operator, please make sure to do the following:

1. Provision enough disk space as per the node role (see: node-provisioning)

2. Setup disk usage monitoring and ensure that the node has enough room to grow and to also accommodate those intermittent spikes.

3. If needed, please add more disk space to the node from time to time.

> It highly recommended to setup alerting around disk usage to facilitate timely action and avoid any downtime and subsequent reward slashing for the node.

Reclaiming disk space

Access, Collection, Consensus and Verification node

If you are running any node other than an execution node and the node is close to running out of disk space or has already exhausted all of its disk, you can re-bootstrap the node's database. This frees up disk space by discarding historical data past a certain threshold.

1. Stop the node.

2. Back up the data folder to a tmp folder in case it is required to revert this change. The default location of the data folder is /var/flow/data unless overridden by the --datadir flag.

```
sh  
mv /var/flow/data /var/flow/databackup
```

3. Configure the node to bootstrap from a new, more recent Root Snapshot. You may use either of the two methods described here to configure your node.

4. Start the node. The node should now recreate the data folder and start fetching blocks.

5. If the node is up and running OK, delete the databackup folder created in step 2.

```
sh  
rm -rf /var/flow/databackup
```

Limitation for Access Node

Re-bootstrapping allows the node to be restarted at a particular block height by deleting all the previous state.

For an Access Node, this results in the node not being able to serve any API request before the height at which the node was re-bootstrapped.

Hence, if you require the access node to serve data from the start of the last network upgrade (spork), do not use this method of reclaiming disk space. Instead provision more disk for the node.

Execution node

For an execution node, the chunk data directory is the one that takes up most of the space. To reclaim space on an execution, do the following:

1. Stop the Execution Node.

2. Remove the Chunk Data Pack Directory. The default is /var/flow/data/chunkdatapack unless overridden by the chunk-data-pack-dir parameter.

Do not delete the bootstrap folder.

```
rm -rf /var/flow/data/chunkdatapack
```

3. Start the Execution Node.

Upon restart, the chunk data pack directory will be automatically recreated.

> Note: Always exercise caution when performing system operations, and make sure you have a backup of important data before making any changes.

```
# slashing.md:
```

```
---
```

```
title: Slashing Conditions
```

```
sidebarposition: 17
```

```
---
```

Introduction

Flow is a proof-of-stake system, which means holders of FLOW can earn inflationary rewards by staking their FLOW tokens to secure and operate the network. A node can participate in the Flow network by depositing a specific amount of stake

(based on role types) thereby making a bonded pledge to participate in the Flow protocol during the upcoming epoch.
(An epoch is a finite amount of time defined by the protocol, approximately one week, during which the nodes participate to run the protocol and are responsible for their operations.)

See the Staking and Epochs section of the documentation to learn more about the design and functionality of this part of the protocol.

Flow nodes follow the procedures defined in the protocol (based on their role) in order to receive rewards. Any deviation (see Slashing Challenges below) from the protocol can result in decreased reward payments or punishments. Severe infractions, which undermine the safety of the network, can lead to "slashing", where some or all of the staked tokens are confiscated from the offending node(s).

This reward and punishment structure is designed to guarantee the security of the protocol and optimize performance over time. This document outlines the most severe infractions against the protocol which result in some portion of a node's stake being taken from them ("slashing conditions"). Enforcing these slashing conditions is critical to ensure the cryptoeconomic security of the protocol. Future documents will describe an incentive structure that encourages system-wide efficiency and speed, by providing bonuses to the most performant nodes and withholding payments to nodes that are unresponsive.

This document assumes a working understanding of the high-level architecture of the Flow blockchain. Readers who are new to Flow or those looking for a refresher are encouraged to read the Protocol Summary here and the staking documentation.

Slashing Conditions

Any violation of the Flow protocol that could result in staked tokens being seized from the offending nodes is called **Slashable Behaviour**. In order for the tokens to be seized, the data necessary to prove the occurrence of **Slashable Behaviour** must be combined with the data necessary to attribute the behaviour to the node(s) responsible into a **Slashing Witness**. (A reduction of rewards, e.g. due to lack of active participation, is not formally included in our definition of slashing.) The Flow protocol considers only server threats to safety and liveness to be slashable conditions and as such, there are no performance related slashing penalties. The one exception is in the case of missing Collections (see the section on MCC below),

where a widespread failure to respond by a large number of nodes is presumed to be coordinated and therefore punishable with slashing.

Most Slashable Behaviour in Flow can be detected and attributed to the offender

by a single honest node observing that behaviour.

(In other words, one node can generate a Slashing Witness without coordinating with other nodes.)

However, some Slashable Behaviour can only be detected and attributed by combining information from multiple nodes. In those situations, the node that first detects the potential infraction raises a Slashing Challenge.

When a challenge is raised, other nodes are expected to provide additional information

which can be combined with the original challenge into a definitive Slashing Witness

that is used to adjudicate the challenge. Each type of Slashing Challenge depends

on different information provided from a different subset of nodes, the details of which are provided below.

Flow adheres to a number of principles in the design of its slashing rules:

- Only Consensus Nodes can perform slashing, and only by following the BFT consensus mechanism defined in the protocol. As such, a super-majority of Consensus Nodes must inspect and confirm a Slashing Witness before any punishment is levied.

- All Slashing Witnesses are objectively decidable.

Given the current protocol state (maintained by the Consensus Nodes) and a well-formed Slashing Witness, all non-Byzantine Consensus Nodes will deterministically come to the same conclusion as to which node or nodes should be slashed (if any) and the amount of stake to be seized.

- All Slashing Behaviour in Flow requires active malfeasance on the part of the offending node.

In other words, a node will only be slashed if it takes an action against the rules of the protocol, and it will not be slashed if it fails to take an action prescribed by the protocol.

("If your machine is crashed, you won't get slashed.") The one exception is in the case of missing Collections (see the section on MCC below), where a widespread failure to respond by a large number of nodes is presumed to be coordinated and therefore punishable with slashing.

- Flow makes no attempt to detect and punish liveness failures within the protocol.

A liveness failure across the network functionally slashes the stake of any participants excluded from participating in the reboot (since their stake is locked in a non-functional network). Community analysis can determine which nodes were responsible for the failure and exclude those Byzantine actors from the new instance.

- Any staked node of Flow can submit a Slashing Witness for any Slashable Behaviour, regardless of its role. (For example, a Collection Node could submit a Slashing Witness for an invalid execution receipt, even though the protocol doesn't require Collection Nodes to verify execution receipts.)
- Submitting an invalid Slashing Witness is Slashable Behaviour. We treat the invalid Slashing Witness itself as the Slashing Witness for that case.

Stages of Slashing

Transitioning to a rigorous staking protocol in which all slashable conditions are checked, enforced, and punished will take place over three phases. The Slashing Challenges section below outlines the various challenges which may be submitted against an offending node but these challenges will not be fully enforced until Phase 3 of the network.

Phase 1: Beta

- In the beta phase of the network, the expectation is that nodes are running error detection and logging but not submitting formal challenges. Any errors found may be submitted to the Flow team for additional testing and security improvements.

Phase 2: Testing Slashing Mechanics

- At this time the slashing mechanisms will be implemented and require testing. Formal challenges should be raised and the protocol will follow the complete, formal mechanics for arbitrating challenges and slashing perpetrators, but no real slashing will take place.

Phase 3: BFT

- By now, the network has been security-hardened and tested and valid challenges result in real slashing of the offending node.

Slashing Challenges

0. All Nodes

Invalid Report Witness (IRW): if any nodes report an invalid/inaccurate witness, an invalid report witness will be reported by the Consensus Nodes, and the node(s) reporting the witness get slashed.

1. Collection Nodes

1.1 Missing Collection Challenge (MCC): Collection nodes are responsible for storing collection content (all transactions) for any collection which they guarantee during the current epoch and the first 1000 blocks of the next epoch. During this time they have to respond to any collection request from staked execution, verification and Consensus Nodes and should respond in a timely manner (specific timeout). If an Execution Node or a Verification Node doesn't receive the response from any of the collection guarantors (Collection Nodes who signed a collection), they can raise a Missing Collection Challenge and broadcast it to the Consensus Nodes to evaluate.

Adjudication: Consensus nodes randomly contact some of the guarantors. If Collection Nodes don't respond, a portion of their stakes will be seized.

If the amount of their stake goes to less than half, they will be fully slashed.

Then the Consensus Nodes notify all the Execution Nodes to skip that collection.

If any of the Collection Nodes respond, Consensus Nodes redirect the collection content

to the Execution Nodes but will also set small penalties both for all the guarantors and that Execution Node (according to their revenue ratio).

1.2 Invalid Collection Witness (ICW): Collection nodes are responsible for responding to collection content queries by collection hash from any staked nodes. The collection hash is the hash of an ordered list of transaction hashes. If a collection content sent by the Collection Node turns out to be invalid, any staked node can report an Invalid Collection Witness. This includes cases where:

- the content is malformed or incomplete,
- there exists an invalid transaction inside the collection, or
- the collection hash doesn't match (inside collection guarantee).

Adjudication: Consensus nodes evaluate the content of the collection, if the collection is found invalid, the Collection Node who signed the content is slashed.

1.3 Double Collection Proposal Witness (DCPW): Collection nodes of a cluster run a mini consensus inside the cluster to decide on a collection, which requires Collection nodes to propose the collection and aggregate votes from others. During the collection consensus, if a Collection Node proposes more than one proposal, any other Collection Node inside the cluster can report a Double Collection Proposal Witness (including both proposals).

Adjudication: Consensus nodes evaluate the content and signatures of these two proposals, and if the witness turns out to be valid, the Collection Node who proposed two collections will get slashed.

1.4 Double Collection Voting Witness (DCVW): Collection nodes of a cluster run a mini consensus inside the cluster to decide on a collection, which requires Collection nodes to propose the collection and aggregate votes from others. During the collection consensus, if a Collection Node votes for more than one collection proposal with identical collection number and size, any other Collection Node inside the cluster can report a Double Collection Voting Witness (including both votes).

Adjudication: Consensus nodes evaluate the signatures of these two votes and evaluate them, and if the witness turns out to be valid, the Collection Node who voted two times will get slashed.

2. Consensus Nodes

2.1 Double Block Proposal Witness (DBPW): Consensus nodes run the consensus (HotStuff algorithm) over blocks. During these consensus steps, if a Consensus Node proposes more than one variation of a block proposal, any other Consensus Node can report a Double Block Proposal Witness (including both proposals). This report will be broadcasted to all other Consensus Nodes.

Adjudication: Consensus nodes evaluate content and signatures of both proposals. If the witness turns out to be valid, the Consensus Node who submitted both proposals will get slashed.

2.2 Double Block Voting Witness (DBVW): Consensus nodes run the consensus (HotStuff algorithm) over blocks. During the consensus steps, if a Consensus Node votes for more than one block proposal with the same height, any other Consensus Node can report a Double Block Voting Witness (including both votes). This report will be broadcasted to all other Consensus Nodes.

Adjudication: Consensus nodes evaluate content and signatures of both votes

and If the witness turns out to be valid, the Consensus Node who submitted both votes will get slashed.

2.3 Invalid Block Vote Witness (IBVW): If a Consensus Node votes for an invalid block or the content of the vote itself is invalid (e.g. vote for non-existing block), any other Consensus Nodes can report an Invalid Block Vote Witness.

Adjudication: Consensus nodes evaluate the vote content and signature. If the witness turns out to be valid, the Consensus Node who submitted the faulty vote will get slashed.

2.4 Invalid Block Proposal Witness (IBPW): If a Consensus Node proposes an invalid block proposal (e.g. quorum certificate without 2/3 vote), any other Consensus Nodes can raise an Invalid Block Proposal Witness.

Adjudication: Consensus nodes evaluate the proposal content and signature, If the witness turns out to be valid, the Consensus Node who submitted the invalid proposal will get slashed.

2.5 Invalid Block Witness (IBW): If the block contents returned by any Consensus Node is invalid, any node can raise the Invalid Block Witness:

- It is malformed or incomplete
- It doesn't match the payload hash provided by the block header

Adjudication: Consensus nodes evaluate the block content and signatures. If the witness turns out to be valid, the Consensus Node who signed the block content will get slashed.

2.6 Invalid Random Beacon Signature Witness (IRBSW):

If any participant of the random beacon returns an invalid signature, an Invalid Random Beacon Signature Witness can be reported by other Consensus Nodes.

Adjudication: Consensus nodes evaluate the random beacon signature. If the witness turns out to be valid, the Consensus Node who signed the invalid random beacon part will get slashed.

3. Execution Nodes

3.1 Faulty Computation Challenge (FCC): If any of the Verification Nodes find a fault in the execution of transactions by an Execution Node it can raise an FCC challenge.

An FCC challenge includes a faulty chunk and all the evidence.

Adjudication: Consensus nodes evaluate the challenge, by sending requests for collection contents and chunk data needed to run the faulty chunk and comparing the results against the expected state commitment. If Consensus Nodes detect any fault in the execution of that chunk, the Execution Node(s) who signed the faulty execution receipts will get slashed. If no fault is found, the Verification Node who raised the challenge will get slashed.

3.2 Conflicting Execution Results Challenge (CERC) :

If two or more variations of execution results are reported by Execution Nodes for a given block. Since only one can be valid, Consensus Nodes raise a conflicting execution results challenge.

Adjudication: As soon as this challenge is raised, all the Verification Nodes go into full check mode (checks all the chunks). The first execution result that receives result approval from at least 2/3 of Verification Nodes is the accurate one, and the other execution results will be considered faulty and Execution Nodes generating those will get slashed. If none of the execution results receive majority approval from Verification Nodes after a very long timeout, all the Consensus Nodes start executing chunks to determine the correct output.

3.3 Invalid Chunk Data Package Witness (ICDPW): If the contents of a chunk data package doesn't match the hash provided inside the execution result, or the contents is invalid, the Verification Nodes can report an Invalid Chunk Data Package Witness.

Adjudication: Consensus nodes evaluate the content of the chunk data package. If the witness turns out to be valid, the Execution Node(s) who signed the faulty chunk data package will get slashed.

3.4 Missing Chunk Data Package Challenge (MCDPC) :

If an Execution Node doesn't respond to the chunk data package request by any staked Verification Node, a Missing Chunk Data Package Challenge can be raised by the Verification Node.

Adjudication: When this challenge is received by the Consensus Nodes, they contact Execution Nodes and ask for the chunk data package. If none of the Execution Nodes respond after a long timeout, all of them get slashed. If any of the Execution Nodes responds with a valid chunk data package, Consensus Nodes redirect the chunk data package to the Verification Nodes

but will also set small penalties both for all the Execution Nodes and the challenge raiser (Verification Node) according to their revenue ratio.

3.5 Execution Results Timeout Challenge (ERTC):

If no execution receipt received in X number of blocks after the submission of each block, the liveness of the system is compromised and Consensus Nodes can raise an Execution Results Timeout Challenge for all the Execution Nodes.

Adjudication: When this challenge is received by the Consensus Nodes, they contact Execution Nodes and ask for an update. If none of the Execution Nodes respond after a long timeout, all of them get slashed. If any of the Execution Nodes return the execution receipt, the case is dismissed.

3.6 Invalid Execution Receipt Witness (IERW):

If an Execution Node provides an execution receipt that is not valid, the Consensus Nodes can report an Invalid Execution Receipt Witness.

Adjudication: Consensus nodes evaluate the content of the execution receipt.

If the witness turns out to be valid, the Execution Node(s) who signed the invalid execution receipt will get slashed.

3.7 Non-Matching SPoCKs Challenge (NMSC):

If the SPoCKs provided by the Execution Node don't match the ones provided by the Verification Node, the Consensus Nodes can raise a Non-Matching SPoCKs challenge.

Adjudication: Consensus nodes have to re-execute the chunk to be able to compute the accurate SPoCKs secret to be able to adjudicate the challenge. This requires requesting the collection and all other data needed for execution from other nodes. Any node which provided invalid SPoCKs will be slashed.

4. Verification Nodes

4.1 Non-Matching SPoCKs Challenge (NMSC):

If the SPoCKs provided by the Execution Node don't match the ones provided by the Verification Node, the Consensus Nodes can raise a Non-Matching SPoCKs challenge.

Adjudication: Consensus nodes have to re-execute the chunk to determine the accurate SPoCKs secret which is needed to adjudicate the challenge. This requires requesting the collection and all other data needed for execution from the other nodes. Any node which provided invalid SPoCKs will be slashed.

4.2 Invalid Result Approval Witness (IRAW):

If a Verification Node provides an invalid result approval,

the Consensus Nodes can report this witness.
This includes cases that a Verification Node sends a result approval for a chunk that was not assigned to the Verification Node (excluding full check mode)
or if the SPoCK's signature doesn't match the public key of the Verification Node.

Adjudication: Consensus nodes evaluate the content and signatures of the result approval.

If the witness turns out to be valid, the Verification Node who signed that result approval be slashed.

```
# spork.md:
```

```
---
```

```
title: Network Upgrade (Spork) Process
```

```
description: Steps to be carried out by node operators during a network upgrade.
```

```
sidebarposition: 15
```

```
---
```

Overview

A spork is a coordinated network upgrade process where node operators upgrade their node software and re-initialize with a consolidated representation of the previous spork's state. This enables rapid development on the Flow Protocol and minimizes the impact of breaking changes.

The Flow network sporks approximately once every year. Upcoming sporks are announced in advance on the #flowValidators-announcements Discord channel and in Upcoming Sporks. The #flowValidators-announcements channel is also used to coordinate during the spork process.

This guide is for existing operators participating in a spork. See Node Bootstrap for a guide to joining the network for the first time.

Step 1 - Cleaning Up Previous Spork State

Once the spork start has been announced on, stop your node and clear your database. The node should stay stopped for the duration of the spork.

```
<Callout type="warning">
  You can skip this step if it is your first time running a node on Flow.
</Callout>
```

1. Stop your Flow node
2. Clear the contents of your data directory that you have previously created. The default location is /var/flow/data. The data directory contains the Flow chain state.

Step 2 - Start Your Node

Once you receive an announcement that the spork process is complete (via Discord), you will need to fetch the genesis info, update your runtime configuration and then boot your Flow node up!

```
<Callout type="warning">
```

If you had set the dynamic bootstrap arguments command line arguments (`--dynamic-startup-access-address`, `--dynamic-startup-access-publickey`, `--dynamic-startup-epoch-phase`) please remove them.

```
</Callout>
```

1. Run the transit script to fetch the new genesis info:

```
./boot-tools/transit pull -b ./bootstrap -t ${PULLTOKEN} -r  
${YOURNODETYPE} --concurrency 10 --timeout 15m
```

- `PULLTOKEN` will be provided by the Flow team

- `YOURNODETYPE` should be one of collection, consensus, execution, verification, access based on the node(s) that you are running.

shell Example

```
$ ./boot-tools/transit pull -b ./bootstrap -t mainnet-16 -r consensus  
Transit script Commit: a9f6522855e119ad832a97f8b7bce555a163e490  
2020/11/25 01:02:53 Running pull  
2020/11/25 01:02:53 Downloading bootstrap/public-root-information/node-  
infos.pub.json  
2020/11/25 01:02:54 Downloading bootstrap/public-root-information/root-  
protocol-snapshot.json  
2020/11/25 01:02:54 Downloading bootstrap/random-  
beacon.priv.json.39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bf  
bf30799a.enc  
2020/11/25 01:02:54 SHA256 of the root block is:  
e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
```

```
$ tree ./bootstrap/  
./bootstrap/  
├── private-root-information  
│   └── private-node-  
info39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bfbf30799a  
|   └── node-info.priv.json  
└── public-root-information  
    ├── node-id  
    └── node-  
info.pub.39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bfbf30799a  
.json  
    └── node-infos.pub.json  
        └── root-protocol-snapshot.json  
        └── random-  
beacon.priv.json.39fa54984b8eaa463e129919464f61c8cec3a4389478df79c44eb9bf  
bf30799a
```

2. Pull the latest changes from flow-go repository
3. Get your node-id, you can find it at /path/to/bootstrap/public-genesis-information/node-id
4. Update the FLOWGONODEID inside runtime-conf.env to the node-id that you got from the previous step
5. Start your Flow node via docker or systemd

See Node Bootstrap for detailed information on Docker/Systemd configuration.

Common Issues

Error: cannot create connection

```
shell
20T18:34:21Z","message":"could not create connection"}
{"level":"error","noderole":"consensus","nodeid":"6d3fac8675a1df96f4bb7a2
7305ae531b6f4d0d2bc13a233e37bb07ab6b852dc","target":"QmVcSQuCdhmk1CMeMN7H
TgGiUYli2KqgVE2vvEmQXK4gAA","error":"failed to dial : all dials failed
[ /ip4/155.138.151.101/tcp/3569] dial tcp4 155.138.151.101:3569:
connect: connection refused","retryattempt":2,"time":"2020-05-
20T18:34:21Z","message":"could not create connection"}
```

This error is OK. Your fellow node operators have not turned on/joined the network yet. So no need to worry about it!

upcoming-sporks.md:

```
---
title: Upcoming Sporks
description: Information about upcoming Mainnet & Testnet sporks
sidebarposition: 16
---
```

The following are the upcoming Spork dates. These dates indicate the intention to Spork. Announcements will be made regarding any delays or changes to these dates in our developer Discord server.

Mainnet Spork Date	Spork Info	
Testnet Spork Date	Spork Info	
Q3 2024 (exact date tbd)		Q3
2024 (exact date tbd)		
May 20, 2024	Devnet50	
Nov 8, 2023	Mainnet 24	
Nov 2, 2023	Devnet49	

Aug 4, 2023			Devnet48	
Aug 4, 2023			Devnet47	
June 21, 2023		Mainnet 23		
Jun 8, 2023			Devnet46	
Jun 7, 2023			Devnet45	
Apr 24, 2023			Devnet44	
Apr 12, 2023			Devnet43	
Apr 12, 2023			Devnet42	
Feb 22, 2023		Mainnet 22		
Jan 30, 2023 -> Feb 22, 2023			Devnet41	
Jan 23, 2023			Devnet40	
Jan 18, 2023		Mainnet 21		
Jan 4, 2023 -> Jan 18, 2023			Devnet39	
Nov 2, 2022		Mainnet 20		
Oct 19, 2022 -> Nov 2, 2022			Devnet38	
Aug 24, 2022		Mainnet 19		
Aug 10, 2022 -> Aug 24, 2022			Devnet37	
Jul 27, 2022			Devnet36	
June 15, 2022		Mainnet 18	June 9, 2022 -> June 15,	
2022	Devnet35			
April 6, 2022		Mainnet 17	April 4, 2022 -> April 6,	
2022	Devnet34			
February 9, 2022		Mainnet 16	February 7, 2022 -> February	
9, 2022	Devnet33			
December 8, 2021		Mainnet 15	December 7, 2021 -> December	
8, 2021	Devnet32			
November 10, 2021			Cancelled	
November 9, 2021 -> November 10, 2021		Devnet31		
October 6, 2021		Mainnet 14	October 5, 2021 -> October	
6, 2021	Devnet30			
September 15, 2021		Mainnet 13	September 14, 2021 ->	
September 15, 2021	Devnet27			
August 18, 2021		Mainnet 12	August 12, 2021 -> August	
18, 2021	Devnet26			
July 21, 2021		Mainnet 11	July 20, 2021 -> July 21,	
2021	Devnet25			
June 23, 2021		Mainnet 10	June 22, 2021 -> June 23,	
2021	Devnet24			
May 26, 2021		Mainnet 9	May 25, 2021 -> May 26,	
2021	Devnet23			
April 28, 2021		Mainnet 8	April 27, 2021 -> April 28,	
2021	Devnet22			
April 7, 2021		Mainnet 7	March 30, 2021 -> March 31,	
2021	Devnet21			
March 10, 2021		Mainnet 6	March 9, 2021 -> March 10,	
2021	Devnet20			

```
</div>

# genesis-bootstrap.md:

---
title: Genesis Bootstrapping
---


<Admonition type="warning" title="Genesis Only">
    All nodes joining the network in May are required to go through this
process as part of the Genesis Bootstrapping.
</Admonition>
```

Overview

To kickstart the Flow network and build the first block, all the nodes that will participate in the first round of consensus need to be known and have exchanged some metadata in advance.

This guide will take you through setting up your nodes, running the initial metadata and key generation, exchanging data back and forth with the Flow team, and then finally starting your nodes to join the network.

Before You Begin

The Flow consensus algorithm depends on there always being a previous block, which means your nodes cannot start until after the Genesis block has been signed. The process of signing that block will be done by the Flow team, and can only be done after every node has completed the first half of the bootstrapping process, which assures that all the identities are included. Since the Flow team needs to wait for metadata from all participants, it will take hours to even days until the Flow network can start.

The bootstrapping process will be in 2 phases, with the Flow team signing the Genesis block between the two.

```
<Admonition type="info" title="Understanding Keys">
    The bootstrapping process deals with a number of different keys. Make
sure you understand their usage and terminology by reviewing the Node
Keys Guide.
</Admonition>
```

Download the Bootstrapping Toolkit

Both phases of the bootstrapping are automated with scripts. Pull a copy onto each of your nodes and extract it.

```
shell Pull-boot-tools
$ curl -sL -O storage.googleapis.com/flow-genesis-bootstrap/boot-
tools.tar
```

```
$ tar -xvf boot-tools.tar
```

Generate Your Node Keys

Start the bootstrapping process by generating your Staking Key and Networking Key. Use your Node Address that you generated in Setting Up a Node in the --address flag, and the node role.

<Admonition type="warning" title="Node Address">Your Node Address must be a publicly routable IPv4 address or valid DNS name that points to your node. This is how other nodes in the network will communicate with you.</Admonition>

```
shell Generate-bootstrap-keys"
$ mkdir ./bootstrap
$ ./boot-tools/bootstrap key --address \"${YOURNODEADDRESS}:3569\" --
role ${YOURNODEROLE} -o ./bootstrap
```

<Admonition type="info" title="BYO Entropy">

By default, the bootstrap script uses the kernel entropy source, either via a getrandom syscall or /dev/urandom. If you have a more secure source of entropy, like a hardware device, you can specify --staking-seed and --networking-seed, to provide your own seeds.

Run the bootstrap command with no flags to print usage information."</Admonition>

<Admonition type="danger" title="Protect your keys!">

The key pairs generated in the bootstrapping process are extremely sensitive and must be managed securely. This guide does not deal with storing the keys in a secure backup or controlling access, as the right approach to this will vary from user to user, but it is something you must consider.

Private keys are suffixed with .priv.json, their public counterparts are not sensitive and can be shared freely.</Admonition>

This command generates two keys, a Staking Key and a Network Key, and stores them both in a .node-info file. Both these keys are needed during runtime and must be present as a file to start your flow node.

For more details around all the keys that are needed to run nodes and their usage, see the Node Keys overview.

The bootstrapping process will create a file structure similar to the following

text bootstrap-directory

```
└──bootstrap
    └──{id}.node-info.priv.json
```

```
└─{id}.node-info.pub.json",
```

Upload Public Keys

To mint the Genesis Block, the Flow team will need the public Staking and Network keys from all your nodes.

If you have previously joined our networks, and you are generating your keys again. Ensure that you take a backup of your keys before generating it again

To facilitate this, the boot-tools directory comes with a script push-keys that will bundle your .pub.json files and send it to the flow team. You can inspect this script to make sure no private key material is being bundled or uploaded. The data not encrypted before being sent as the public keys involved are not sensitive.

In phase 2 of the bootstrapping process, the Flow team will need to securely issue each node a Random Beacon key. This key is again sensitive and unique to your node. To enable this, the push-keys script also generates another key pair called the Transit Key. The public key of this pair will be uploaded along with the Staking and Network keys, and your Random Beacon key will be encrypted with it before being sent to you. You must keep your Transit Key until you have received and decrypted your Random Beacon key from the Flow team.

```
<Admonition type="warning" title="Token Needed">
```

The transit script here need a -t token parameter flag. This token will have been provided to you by the Flow team out of band. Reach out to your contact if you don't have your token.

```
</Admonition>
```

```
shell Upload-public-keys
```

If you joined our network previously, make sure to take a backup!

```
cp /path/to/bootstrap /path/to/bootstrap.bak
```

```
$ ./boot-tools/transit push -d ./bootstrap -t ${TOKEN} -role
```

```
${YOURNODEROLE}
```

Running push

Generating keypair

Uploading ...

Uploaded 400 bytes

```
<Admonition type="danger" title="One and Done!">
```

Once you've run the bootstrap and are confident in your setup, run the transit push command only once. If you bootstrap again and transit push again with a new node ID, it will count against your quota of Nodes. Exceeding your quota will result in a long back and forth with the Flow team to see which node is the extra one.

```
</Admonition>
```

Update Node Config

As flow node requires a --nodeid flag to start. You will need to pass in the contents of the node-id into either your container, runtime-config.env file, or hard coded into the systemd unit file which the flow team provides.

You can get the node-id from the metadata that you pulled. It will be at: /path/to/bootstrap/public-genesis-information/node-id

Wait

Now the ball is in the Flow team's court. As soon as all nodes have completed the above steps, the Genesis block will be created and distributed to you.

Join the Flow discord server if you haven't already and stay tuned for updates. Your nodes need not be online during this waiting period if you want to suspend them to reduce cost, but you must not lose your key material.

```
<Admonition type="info" title="A Note on Staking">
    For the Genesis Block, your nodes will start pre-staked, which means no action on your part is needed to get your nodes staked.
```

```
    For more details on staking check the guide on Staking and Rewards.
</Admonition>
```

Receive Your Random Beacon Keys

When the Flow team gives the go-ahead, your Random Beacon keys will be available for retrieval. Each Node will need to pull their own keys down individually.

```
shell Pull-beacon-keys
$ ./boot-tools/transit pull -d ./bootstrap -t ${TOKEN} -role
${YOURNODEROLE}
Fetching keys for node ID FEF5CCFD-DC66-4EF6-8ADB-C93D9B6AE5A4
Decrypting Keys
Keys available
```

Pulling your keys will also pull a bunch of additional metadata needed for the bootstrapping process.

In the end, your bootstrap directory should look like this:

text bootstrap-directory

```
bootstrap/
└── private-genesis-information
    ├── private-node-info{node id}
    │   └── node-info.priv.json
    └── random-beacon.priv.json
└── public-genesis-information
    └── dkg-data.pub.json
```

```
└── genesis-block.json
└── genesis-cluster-block.{cid}.json
└── genesis-cluster-block.{cid}.json
└── genesis-cluster-qc.{cid}.json
└── genesis-cluster-qc.{cid}.json
└── genesis-commit.json
└── genesis-qc.json
└── node-id
└── node-info.pub.{node id}.json
└── node-infos.pub.json
└── <additional files...>
```

<Admonition type="info" title="Why are we generating the beacon keys for you?">

Unlike staking and account keys, the beacon keys are not randomly generated, and depend on inputs from all consensus nodes on the network. In typical Flow network operation, these keys will be dynamically generated on demand by the consensus nodes communicating. However for genesis, as the consensus nodes aren't communicating yet, the Flow team will generate and distribute them to kickstart the process.

</Admonition>

Move Genesis Data

This bootstrapping data is needed by your node at each startup, so it must be present on disk.

Where in the filesystem you store this data is up to you, but you may not change the folder structure generated by the bootstrapping process. By default, flow stores this data under /var/flow/bootstrap.

New Images

Once the Genesis block has been minted, it will be included into the official container images so that it's available to all nodes. Pull the new images, which should now be version v1.0.0.

Start Your Nodes

Once every node has pulled its keys and fetched the new images, the network is ready to start.

Make sure you're part of the Discord Chat. Once all nodes are ready, updates will be provided to everyone.

Start your systems, let's make some blocks!

```
# spork-practice.md:
```

```
---
```

```
title: Spork Practice
```

Sporking

The actual process of Sporking will mostly be covered by the Node Operators Quick Guide, and will not be covered here.

Spork

Instead, we'll aim to give some instructions for those that want to Practice the process themselves, before joining the Mainnet Spork.

This guide assumes you have access to the Flow-Go repo, which you'll need to build up-to-date containers and run code snippets.

[] (<https://github.com/onflow/flow-go>)

Local Testnet

One way to get a good feel of the network without too much interaction with infrastructure is to play with the local testnet, which we've named the Flow Local Instrumented Test Environment (FLITE).

<https://github.com/onflow/flow-go/blob/master/integration/localnet/README.md>

FLITE will allow you to start a full flow network locally, which means starting all 5 roles required for a functioning network. Instructions for initializing and starting the local network are provided in the README above.

When Starting FLITE, it will build all the docker images required for the network. This can also be done manually ahead of time, using make docker-build-flow from the root directory of flow-go

Remote Testnet

If you would like more control over the nodes, beyond what docker compose can provide, or you wish to deploy the docker images to separate VMs, to more closely imitate Mainnet, you will have to manually run bootstrapping for a specific configuration of nodes that you would like to test.

[] (<https://github.com/onflow/flow-go/blob/master/cmd/bootstrap/README.md>)

Example files are available in the cmd/bootstrap/examplefiles folder.

Where the node-config.json will usually store all flow's nodes, whereas partner node info usually goes into a separate folder. The last file, which will need to be manually populated, is the partner stakes file, which takes the IDs of all the partner nodes and associates a stake. For now, this can be arbitrary.

Once you have all the information, you can make use of the finalize command:

[] (<https://github.com/onflow/flow-go/tree/master/cmd/bootstrap#example-1>)

And generate the bootstrapping folder required to start up your nodes.

Once you have the bootstrapping folder, you'll be able to start up all the nodes that were included in the bootstrapping process.

Node Setup Docker

The startup command will look very similar to what is provided in the quick guide. One such example, assuming we named our bootstrap folder `bootstrap`:

```
shell
docker run --rm \
    -v /path/to/bootstrap:/bootstrap:ro \
    -v /path/to/data:/data:rw \
    --name flow-go \
    --network host \
    gcr.io/flow-container-registry/execution:latest \
    --nodeid=${FLOWGONODEID} \
    --bootstrapdir=/bootstrap \
    --datadir=/data/protocol \
    --rpc-addr=0.0.0.0:9000 \
    --nclusters=${FLOWNETWORKCOLLECTIONCLUSTERCOUNT} \
    --bind 0.0.0.0:3569 \
    --loglevel=error
```

The two missing pieces of info here are `FLOWGONODEID` which will have been generated from the bootstrap process, and will depend on which node you're trying to run, and `FLOWNETWORKCOLLECTIONCLUSTERCOUNT` which we've been defaulting to 2

Practice Testnet

Lastly, if the goal is to practice the entire Sporking procedure, including transit of staking and networking keys, and joining a network, we can help spin up a Testnet temporarily for this purpose. This will require quite a bit of coordination, and will basically be the same steps as the Mainnet spork, so please let us know if this is something you'd like to do and we'll connect to plan accordingly.

```
# starting-nodes.md:

---
title: Starting Your Nodes
---
```

Prior to starting up your nodes make sure you have the following items completed:

1. Bootstrap process completed with the bootstrap directory handy (default: /var/flow/bootstrap)
2. Flow data directory created (default: /var/flow/data)
3. node config ready
4. Firewall exposes TCP/3569, and if you are running access node also the GRPC port (default: TCP/9000)

For more details head back to [Setting up your node](#)

When you have all the above completed, you can start your Flow node via systemd or docker.

systemd

Ensure that you downloaded the systemd unit file. If you haven't, follow the [Set your node to start](#) guide to get your unit file and enabled.

Once you have your Flow service enabled you can now start your service: `systemctl start flow`

Docker

If you don't have have systemd on your system, or prefer not to use systemd, you can run the following docker commands for your respective Flow role to start your node!

Access

```
docker run --rm \
  -v /path/to/bootstrap:/bootstrap:ro \
  -v /path/to/data:/data:rw \
  --name flow-go \
  --network host \
  gcr.io/flow-container-registry/access:v0.0.6-alpha \
  --nodeid=${FLOWGONODEID} \
  --bootstrapdir=/bootstrap \
  --datadir=/data/protocol \
  --rpc-addr=0.0.0.0:9000 \
  --ingress-addr=${FLOWNETWORKCOLLECTIONNODE} \
  --script-addr=${FLOWNETWORKEXECUTIONNODE} \
  --bind 0.0.0.0:3569 \
  --loglevel=error
```

Collection

```
docker run --rm \
  -v /path/to/bootstrap:/bootstrap:ro \
  -v /path/to/data:/data:rw \
  --name flow-go \
  --network host \
  gcr.io/flow-container-registry/collection:v0.0.6-alpha \
```

```
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--rpc-addr=0.0.0.0:9000 \
--nclusters=${FLOWNETWORKCOLLECTIONCLUSTERCOUNT} \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Consensus

```
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
gcr.io/flow-container-registry/consensus:v0.0.6-alpha \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--nclusters=${FLOWNETWORKCOLLECTIONCLUSTERCOUNT} \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Execution

```
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
gcr.io/flow-container-registry/execution:v0.0.6-alpha \
--nodeid=${FLOWGONODEID} \
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--ingress-addr=0.0.0.0:9000 \
--nclusters=${FLOWNETWORKCOLLECTIONCLUSTERCOUNT} \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Verification

```
docker run --rm \
-v /path/to/bootstrap:/bootstrap:ro \
-v /path/to/data:/data:rw \
--name flow-go \
--network host \
gcr.io/flow-container-registry/verification:v0.0.6-alpha \
--nodeid=${FLOWGONODEID} \
```

```
--bootstrapdir=/bootstrap \
--datadir=/data/protocol \
--nclusters=${FLOWNETWORKCOLLECTIONCLUSTERCOUNT} \
--bind 0.0.0.0:3569 \
--loglevel=error
```

Additional Flags

Networking Layer

All networking layer settings are initialized to default values from the config/default-config.yml file when the Flow node starts up. Each attribute in this YAML file matches a flag name, allowing you to override the default setting by specifying the corresponding flag in the docker run command. For instance, to change the networking-connection-pruning setting, use its matching flag name (networking-connection-pruning) and desired value in the docker run command.

```
# 02-epoch-terminology.md:
```

```
---
```

```
title: Epoch and Staking Terminology
sidebarlabel: Epoch and Staking Terminology
description: Important Definitions for Epochs
---
```

```
<Callout type="warning">
  If you haven't read the staking introduction, please read that
  first. That document provides a non-technical overview of staking on
  Flow for
  all users and is a necessary prerequisite to this document.
</Callout>
<Callout type="warning">
  This document assumes you have some technical knowledge about the Flow
  blockchain and programming environment.
</Callout>
```

Terminology

If any of the definitions are confusing, you can find more detail in the other sections of the technical docs.

Staker: Any user who has staked tokens for the Flow network.
A node operator is a staker, and a delegator is a staker as well.

Node Operator: A user who operates a node on the Flow network. Each node operator has a unique node resource object they store in their account to perform staking operations.

Node Operator Metadata: This information is tracked for each node operator in the Flow network.

- Node ID: 32 byte identifier for the node. Usually a hash of the node public key.

- Role: Indicates what role the node operator is. (Collection, Consensus, Execution, Verification, Access)
- Networking Address: The address that the node operator uses for networking. Using a hostname is highly encouraged.
- Networking Key: The 64 byte ECDSA-P256 node operator public key for networking.
- Staking Key: The 96 byte BLS12-381 public key for the node. Used to sign node messages and votes for Quorum Certificate generation.
- Proof of Possession: A 48 byte (96 hex characters) string that acts as cryptographic proof of ownership of the node's staking key.

Delegator: A user who delegates tokens to a node operator and receives rewards for their staked tokens, minus a fee taken by the node operator. Each delegator stores a unique delegator resource object in their account that allows them to perform staking operations.

- Delegator Metadata: This information is tracked for all delegators in the network.
 - id: The ID associated with a delegator. These IDs are assigned to delegators automatically by the staking contract and are only unique within an individual node operators' record.
 - nodeID: The ID of the node operator a user delegates to.

Node Identity Table: The record of all the nodes in the network, and their delegators.

The identity table keeps separate lists for the info about node operators and delegators.

```
<Callout type="warning">
  NOTE: The staking smart contract does not associate a node or delegator with
  an account address. It associates it with the assigned resource object
  that
  corresponds to that entry in the contract. There can be any number of
  these
  objects stored in the same account, and they can be moved to different
  accounts if the owner chooses.
</Callout>
```

Epoch: The period of time between changes in the identity table and reward payments.

(Initially a week, measured in consensus views)

At the end of every epoch, insufficiently staked node operators are refunded their stake,
 rewards are paid to those who are currently staked, committed tokens are marked as staked,
 unstaking tokens are marked as unstaked, and unstaking requests are changed from staked to unstaking.

Consensus View: A internal detail that the Flow consensus algorithm, HotStuff, uses to measure time.

Views count the number of rounds in the consensus algorithm.

Each round/view the counter is incremented and a new block may be proposed.

Seat/Slot: The right to participate in the network as a node of a certain type

for a specific Epoch. There are a limited number of seats/slots for each node type per epoch.

Current Slot Limits (may be slightly different than what is shown here):

- Access Nodes: 167
- Collection Nodes: 156
- Consensus Nodes: 149
- Execution Nodes: 10
- Verification Nodes: 105

Candidate: A node that has committed tokens for the next epoch but has not been accepted yet.

There is a limited number of node slots per epoch and candidate nodes are selected randomly,

so there is a chance that a candidate node will not be chosen to participate in the next epoch

because there aren't enough slots even if they meet all the other regular requirements

Staking Auction Phase: The period of time when nodes and delegators are able to submit staking operations in preparation for the upcoming epoch. This phase is expected to take over 90% of the time of an epoch.

Epoch Setup Phase: The period of time after the staking auction, where nodes have to perform certain processes to initialize the state and communication with other nodes for the next epoch.

These processes are called Cluster Quorum Certificate Generation (QC), and Distributed Key Generation (DKG).

If any node does not perform this initialization properly, it is not included in the next epoch's Identity Table.

This phase is expected to take less than 10% of the time of an epoch, near the end.

Cluster Quorum Certificate Generation (QC): A process by which nodes using the HotStuff consensus algorithm submit signed messages in order to generate a certificate for bootstrapping HotStuff. Each collector cluster runs a mini-version of HotStuff, and since clusters are randomized each epoch, a new quorum ceritificate is required for each cluster each epoch.

Distributed Key Generation (DKG): Process for generating a shared public key to initialize the random beacon.

Consensus nodes use a shared whiteboard to communicate and submit final key vectors to generate a shared key.

Epoch Commit Phase: The final phase of an epoch, after the Epoch Setup Phase. In this phase, the identity table has been finalized for the next epoch, all setup has been completed, and the network is simply waiting for the next epoch to start.

Service Event: Special messages that are generated by the epoch smart contracts and included in execution results.

They enable communication between system smart contracts and the Flow protocol.

In other words, they serve as a communication mechanism between the execution state and the protocol state.

Service events are not any different than other Cadence events, except in the fact that

Flow nodes treat them differently because they are being emitted by the service account.

Node and Delegator Staked Token Tracking Terms:

- **Tokens Committed:** The tokens that a user has committed to stake in the next epoch, but that aren't currently staked.

- **Tokens Staked:** The tokens that a user has staked in the current epoch.

- **Tokens Requested to Unstake:** The amount of tokens that a user has requested to be unstaked

at the end of the current epoch (to be removed from the tokens staked pool).

- **Tokens Unstaking:** The tokens that were unstaked at the beginning of the current epoch and

are being held for an additional epoch holding period before being released.

- **Tokens Unstaked:** Tokens that used to be committed or staked and have been unstaked.

- **Tokens Rewarded:** Tokens that the user has received via staking rewards.

Delegation Rewards Cut: The percentage of a delegator's rewards that the node operators take. Initially set to 8%.

Epoch Payout: The total amount of tokens paid in rewards at the end of an epoch.

This value will change as the supply of FLOW changes. See the rewards page for more details.

Minimum Stake Requirement: Each node type AND delegator has a requirement for the minimum number of FLOW they have to commit to stake to be considered a valid staker and receive rewards.

If a node operator or delegator does not meet the minimum stake, they will not be included in the next epoch and will not receive any rewards.

- **Access Nodes:** 100 FLOW

- **Collection Nodes:** 250,000 FLOW

- Consensus Nodes: 500,000 FLOW
- Execution Nodes: 1,250,000 FLOW
- Verification Nodes: 135,000 FLOW
- Delegators: 50 FLOW

There is no maximum stake limit.

```
# 03-schedule.md:
```

```
---
```

title: Epoch and Reward Schedule
sidebarlabel: Epoch and Reward Schedule
description: How the Flow protocol manages the schedule of an epoch and rewards payments

```
<Callout type="warning">
```

This information covers the current state of how epoch phases are ran and how rewards are calculated and distributed to node operators and delegators. All of the information in the document is still being designed and is subject to change based on research and discussion by the Flow core team and community. If any changes are proposed, the Flow community will be notified in advance.

```
</Callout>
```

Schedule

!Flow Epoch Schedule

An Epoch is a period of time when the node operators in the network are constant. At epoch boundaries, newly staked node operators are able to join the network and existing node operators which have unstaked may exit the network.

Each epoch lasts approximately a week, and the Epoch Switchover is defined as the point in time when one epoch ends and the next epoch begins, marking a possible change in the list of valid nodes.

```
<Callout type="info">
```

The exact timing of each epoch end is influenced by the number of blocks proposed during the epoch. Therefore, epoch phase timing can vary and will likely drift over time.

All quoted epoch end times are estimates and subject to some variance (up to several hours).

See Epoch Preparation Protocol for details.

</Callout>

Staking Operations are disabled for approximately the last 6-12 hours of an epoch,
typically around 00:00 US Pacific Daylight Time (07:00 UTC) on Wednesday every week until around 12:00 US Pacific Daylight Time (19:00 UTC). See Epoch Setup for more information on this phase.

Epoch Switchovers will happen around 12:00 pm PT on Wednesday (7:00 pm UTC) every week.

Please note exact epoch ending time vary based on the performance of the network & all staking operations that interact with staked tokens will be processed by the protocol at the start of each epoch.

Rewards

Rewards are usually paid around 12 pm PT on Wednesday (7:00 pm UTC), every week, to all users that have tokens staked. This is close to the same time as the Epoch Switchover. See the Rewards Distribution section below for more information about rewards calculation and schedule.

Staking Auction

The first, and longest phase of an epoch is the staking auction. This phase is when nodes and delegators can register to stake and perform other staking operations such as staking more tokens or unstaking their existing tokens. None of these operations are fully executed until the Epoch Switchover though.

The Staking Auction lasts for at least the first 90% of the length of an Epoch

Epoch Setup and Epoch Commit

The Epoch Setup and Epoch Commit phases are the final phases of the epoch, when node operators who have been included in the next epoch perform important setup functionality to prepare for the next epoch.

The Epoch Setup and Epoch Committed phases usually last less than 10% of the time of an epoch.

Staking Operations will be rejected during the Epoch Setup and Epoch Commit phases. This is because the staking information has been finalized in preparation for the next epoch

and cannot be changed because these final phases rely on the staking information being constant.

The Staking Auction Ends every Wednesday near 00:00 PDT (07:00 UTC). This means that staking operations will be disabled for ALL users in the period between the end of the staking auction and the beginning of the next epoch, currently 6-12hrs.

Rewards Distribution

The rewards distribution schedule has been designed to ensure there is enough liquid supply of FLOW available in the ecosystem to empower a wide variety of use cases and promote fair and diverse participation in the Flow ecosystem.

The numbers in this table represent the total amount of tokens that are paid

as staking rewards at each epoch to the entire pool of participants in the Flow network.

While the total staking reward amount is known and fixed per epoch, rewards that individual stakers receive are variable depending on many factors.

The total rewards for each epoch are fixed for that epoch, but where those rewards come from can change.

When the protocol pays rewards, it first pulls from the central pool of all the transaction fees that have been paid by every user in the network since the last rewards payment.

Once that pool has been depleted, the protocol mints new tokens that are used as rewards.

Please see the next section on how to calculate an individual staking reward.

	Dec 22, 2020	Dec 29, Jan 5,
12, 19, 26 (2021) Feb 2, 2021	weekly on Wednesdays indefinitely	
----- ----- -----		
Total Rewards % (Annual)	5%	20%
5%		
Total Rewards Amount Per Week	1.2M FLOW	4.4M FLOW
1.3M FLOW		

Individual Calculation

Each user gets a percentage of the total rewards during each epoch that is proportional to their percentage of all the tokens that are staked by all participants.

The full reward calculation on a per-user basis is equal to:

$$\text{New Reward(user)} = \text{Tr} \quad (\text{Sn} / \text{St})$$

where:

- Tr = Total staking rewards to be paid out during the current epoch.
(See table above)
- Sn = Amount of FLOW Staked by the target user for the current Epoch.
(Different for each staker)
- St = Sum of all the FLOW staked by all the participants in the network.
(Changes every epoch)

Rewards for delegators are also calculated in the exact same way that rewards for node operators are calculated,
with one difference in that 8% of the calculated reward amount is given to the node operator being delegated to
(effected as a protocol layer fee, which is the same for all node operators).

The remaining 92% is awarded to the delegator.

Note: the 8% fee is only applied to the staking reward, not to the tokens delegated.

With this calculation, the node you choose to run or delegate to DOES NOT affect the amount of rewards you receive every week.
The only variable that you can control is the number of tokens you have staked.

The more tokens you stake, the more rewards you will receive.

Because of the variable nature of the rewards calculation, we cannot provide an expected weekly/yearly return for a single staker. You can plug your own numbers into the formula to see some sample calculations, but you won't be able to know exactly what you will earn until the beginning of the epoch in which you are participating in staking or delegation.

Rewards History

For the first two years of its existence, the staking rewards payments were handled with manual transactions. You can find the history of those transactions including their arguments and IDs in the Pay Rewards Section of the Flow Service Account Repo. The dates correspond to the date when the rewards were paid at the end of an epoch and the network transitioned to a new epoch.

Future rewards payments and epoch switchovers happen automatically via a system chunk transaction, which does not create regular transaction IDs.

04-epoch-preparation.md:

title: Epoch Preparation Protocol

```
sidebarlabel: Epoch Preparation Protocol
description: Technical Overview of the Flow Epoch Protocol
---

<Callout type="warning">
  If you haven't read the staking introduction, please read that
  first. That document provides a non-technical overview of staking on
  Flow for
  all users and is a necessary prerequisite to this document.
</Callout>
<Callout type="warning">
  This document assumes you have some technical knowledge about the Flow
  blockchain and programming environment.
</Callout>
```

Epochs

The epoch preparation protocol defines how information about the next epoch
is determined and propagated to the protocol state.

There are two primary actors in this protocol, the Epoch Smart Contracts,
and the Consensus Committee:

- Epoch Smart Contracts - the smart contracts that manage epochs:
 - FlowClusterQC : Manages the quorum certificate generation for bootstrapping
 - the hotstuff consensus algorithm for each collector cluster.
 - FlowDKG : Manages the Distributed Key Generation that consensus nodes participate
 - in to initialize the random beacon for each epoch.
 - FlowIDTableStaking : Manages the source of truth for the identity table,
 - and enforces rules related to staking FLOW, delegating, paying rewards, and allocating token movements between epochs.
 - FlowEpoch : Ties all of the previously mentioned contracts together to manage
 - the high level epoch lifecycle. FlowEpoch acts as a state machine that transitions
 - between different epoch phases when specific conditions from the other contracts are met and triggers important operations in the other smart contracts when phase changes happen.
- Consensus Committee - the committee of consensus nodes for the current epoch

This document describes the communication protocol between these two actors and the impact on the protocol state.

It gives an overview of the process of epochs, the staking auction, and the epoch setup and commit phases.

It is an important prerequisite to understand before proceeding with any other technical integration or interaction with the Flow Protocol, but does not provide step-by-step instructions for how to perform specific actions.

The transactions described in this document are contained in the flow-core-contracts/transactions/epoch/ directory. You can see the text of all the transactions used to interact with the smart contracts there.

Epochs Overview

Only a pre-determined set of nodes is authorized to participate in the protocol at any given time.

The set of authorized nodes is a-priori known to all network participants.

This set is referred to as the Identity Table. An Epoch is defined as a period of time where the set of authorized nodes is constant (or can only shrink due to ejection of malicious nodes).

At an Epoch switchover, which is the time when the network transitions from one epoch to the next,

the set of authorized nodes can change. For each of Flow's node roles, the Flow protocol admits a protocol-determined number of nodes.

For each Epoch, there is a Staking Auction in which new potential node operators may submit Staking Commitments.

All this is completely smart-contract based and handled through conventional transactions.

After the Staking Auction is over, the protocol determines which commitments to accept and which to reject.

The node operators whose staking commitments were accepted are added to the Identity Table for the next epoch,

and become authorized participants at the next epoch switchover.

Staked Nodes also can submit other operations to modify their existing stake, which are all carried out at the end of the current epoch.

The smart contract that determines the nodes for the next Epoch has special privileges.

Specifically, it is allowed to emit Service Events, which are how the execution state updates the consensus node-based protocol state.

At the end of the staking auction, the epoch smart contracts conclude that they have now determined

the set of nodes which will be running the network for the next Epoch, and the amount of FLOW that all the nodes have staked.

The smart contract then emits a service event with this information.

When processing the block with seat assignment, all network nodes (including future ones which are supposed to monitor the chain in anticipation) are thereby informed about the upcoming change.

<Callout type="warning">

Note: At this point in the epoch (end of the staking auction),

there is no change in participating nodes.
The change in participating nodes happens at the end of the epoch.
</Callout>

After the staking auction, there is an interim period of time until the new Epoch starts for the following tasks to be completed:

- The epoch smart contract runs the cluster assignment algorithm for all the collector nodes
- and each collector node will vote for the root block of their respective clusters
- The Random Beacon Committee for the next Epoch (currently all consensus nodes)
- will run the Distributed Key Generation (DKG),
- When completing the QC generation and DKG, the smart contracts will emit a service event.

After consensus nodes have collected all relevant information (public keys for the random beacon and cluster quorum certificates), they can update the identity table to include the information for the next Epoch.

If preparation for the next Epoch is not completed before the current Epoch ends,
the network goes into epoch fallback mode (EFM) and a special transaction, sometimes including
a spork, is required to transition to the next Epoch.

Epoch Length

The length of an Epoch is measured in terms of consensus views.
The number of views in an epoch and in the various epoch phases are determined before
the Epoch begins and stored as a field in the main epoch smart contract (FlowEpoch).

Generally, there is not a block for every view, so the view number will not change at the same rate as the block height.

Because the length of a consensus view can vary depending on many different factors,
the wall-clock time of an epoch is expected to vary from week to week.
Under typical network conditions we expect the variance in epoch length to be less than 2 hours for a 1-week epoch (1%).
Under adverse network conditions the variance in epoch length will increase (typically this will result in longer epochs).

As the average view rate changes over time, the Service Account can change the epoch length to target a 1 week wall-clock epoch length.

Phases

The preparation for the next epoch is separated into distinct phases.
Each phase occurs completely within the current epoch.

!Flow Epoch Schedule

The Epoch Smart Contract acts as a state machine. The smart contract keeps a record of the current phase, the number of views in the current phase, and the conditions that need to be met in order to advance to the next phase, or next epoch. A special Heartbeat resource is used to call the `advanceBlock()` method during every single new block in Flow. During these regular method calls, if all of the conditions are met to advance to the next phase, the smart contract performs any relevant retrieval and storage of information, emits a Service Event, and transitions to the next phase, which often involves setting certain metadata or enabling one of the connected smart contracts to begin its work.

From the perspective of the consensus committee, the phase transitions within epochs occur as a result of including a service event in a block, thus the phase transition only applies to the fork containing the block with the service event.

At the end of Phase 0 and beginning of Phase 1, the `EpochSetup` service event is emitted that contains the identity table and other initial metadata for the upcoming epoch.

At the end of Phase 1 and beginning of Phase 2, the `EpochCommit` service event is emitted that contains the results of the Epoch Setup phase.

The start of a new epoch is the first block with its view > the last view of the previous epoch, and its parent view \leq the last view of the last epoch.

Phase Transitions

The Consensus Committee triggers the phase transition coinciding with the Epoch switchover by publishing the block of the next Epoch. This block's execution state will also detect the the end view of an epoch has arrived and trigger the start of the new epoch. The transition to a new epoch is also marked by the emission of an event (`EpochStart`) from the epoch smart contract.

-
The state of the smart contracts reflect the latest epoch's new identity table and metadata.

For the Epoch-internal Phase transitions, meaning the phase transitions within an epoch,

the Epoch Smart Contract provides the trigger by emitting a respective service event:

- The EpochSetup service event triggers the phase transition Staking Auction Phase → Epoch Setup Phase
- The EpochCommit service event triggers the phase transition Epoch Setup Phase → Epoch Committed Phase

Only one of each service event may be emitted each epoch, for a given fork.

EpochCommit may only be emitted after EpochSetup has been generated in the respective given fork.

The FlowEpoch contract manages all of these phases, the FlowIDTableStaking contract manages the identity table and staking auction, the FlowClusterQC contract manages the Quorum Certificate generation for collector clusters, and the FlowDKG contract manages the Distributed Key Generation protocol for the consensus nodes.

Initially, control of these phases and contracts will be managed manually by the Flow Token Admin, but control will eventually be completely decentralized and managed by the node software, smart contracts, and democratically by all the stakers in the network.

Phase 0: Staking Auction

Purpose: During the staking auction phase, operators can put up stake in exchange for being a part of the upcoming epoch.
All voluntary commitments to register a new node, increase, or decrease stake for the next epoch must occur before the end of this phase.

Duration: The staking auction phase begins with the first block of the current Epoch
Its last block is the block in which the EpochSetup service event is emitted.

Protocol Directives:

Epoch Smart Contract

- The FlowEpoch Smart Contract is responsible for ensuring that staking, un-staking, and stake-modification transactions for the next epoch are only executed during the staking auction and fail otherwise. The contract enforces this by setting a stakingEnabled field in the staking contract.
- Every staking method checks to see if this is set before executing.

- The FlowEpoch Smart Contract must ensure that the subsequent phases

are sufficiently long to perform all required tasks before the epoch ends.

- As part of the execution result for the last block of the staking auction,
the Epoch Smart Contract computes the seat assignment information for the next epoch,
and emits a specialized service event, the EpochSetup event,
with the timing and identity table information about the next epoch.
See the Epoch Setup Event Documentation
for a detailed breakdown of the epoch setup event.

Phase 1: Epoch Setup

Purpose: During the epoch setup phase, all nodes participating in the upcoming epoch must perform setup tasks in preparation for the upcoming epoch.

Duration: The epoch setup phase begins right after the EpochSetup service event is emitted.

It ends with the block where EpochCommit service event is emitted.

Protocol Directives:

Consensus:

- When a primary constructs a block that seals the EpochSetup service event,
the primary includes an update to the protocol state in the block.
Specifically, it adds the nodes for the PendingEpoch to the list of authorized nodes.

When this block is propagated, all staked nodes will know about the participants

in the next epoch and can communicate with them.

- Based on the RandSeed field in the EpochSetup event, all nodes compute:
 - The seed to initialize the consensus node's primary selection algorithm for the next epoch
 - The seeds to initialize the collector clusters' primary selection algorithm for the next epoch
- The collector nodes generate the root block for their respective clusters
in the next Epoch and submit a vote for the root block to a specialized smart contract, FlowClusterQC.
- The Random Beacon Committee for the next Epoch (currently all consensus nodes)
will run the DKG through a specialized smart contract, FlowDKG.

Epoch Smart Contract:

- The FlowEpoch Smart Contract is responsible for ensuring that Epoch Setup transactions
are only executed during the Epoch Setup phase and fail otherwise.

The contract enforces this by setting an enabled field in the FlowClusterQC and FlowDKG contracts.
Every state-changing method from these contracts checks to see if this is set before executing.

- The FlowEpoch Smart Contract must ensure that the subsequent phase is sufficiently long to perform all required tasks before the epoch ends.
- As part of the execution of the last block of the Epoch Setup phase, the FlowEpoch Smart Contract computes the public key shares generated by the DKG and the QCs for the collector clusters and publishes these as EpochCommit service event.
The FlowEpoch Smart Contract should emit this event as soon as the artifacts are determined.

See the Epoch Commit Event Documentation for a detailed breakdown of the epoch commit event.

Phase 2: Epoch Committed

Purpose: When the epoch committed phase starts, the precise role of each node is fully specified.
From a protocol-perspective, all information is available for each node to start its operation for the next Epoch.
This phase provides some time for nodes to establish the communication channels and synchronize with the network to seamlessly switch over to the next epoch.

Duration: The epoch committed phase begins right after the EpochCommit service event has been emitted. It ends when the epoch ends.

Protocol Directives:

Consensus

- When a primary constructs a block that seals the EpochCommit service event, the primary includes an update to the protocol state in the block. Specifically, it:
 - adds the information generated in the setup phase to the Protocol State and
 - marks the updated Protocol State as committed in this respective fork.

Query Information from the Epoch Contract

See the epoch scripts and events document for detailed documentation about you can use scripts events to learn information about the state of the epoch contracts.

```
# 04-stake-slashing.md:

---
title: Stake Slashing
sidebarposition: 4
description: How Flow enforces honest node behaviour
---

Flow slashes nodes only for acts that directly impact
the security and integrity of the network and its shared execution state.
Nodes are not slashed for liveness infractions.
The protocol reserves slashing for maintaining the security of the
protocol rather than its liveness.
```

You can find more details on the conditions under which a node is slashed in the Flow whitepapers.

Direct stake slashing is not currently enforced by the protocol and staking contract.
It will be handled on a case-by-case basis for the foreseeable future to ensure network participants have time to participate in the testing and rollout of slashing.

There is a very basic form of slashing that is currently used, where nodes who have liveness issues during an epoch may have their rewards and their delegators' rewards reduced by a pre-determined amount based on
the severity of the liveness infractions. This amount is often 50% and is only taken from the stakers' rewards for a given epoch.
Their staked FLOW is not touched at all.

When slashing is enforced, slashable protocol violations must be adjudicated by a supermajority of more than 2/3 of the staked consensus nodes in order to take effect.
If a node is found guilty of committing a slashable protocol violation, the consensus nodes directly deduct a fine from the node's stake.

It is still TBD where the slashed tokens will be deposited.

The remaining un-slashed stake is deposited back into node's unstaked pool
at the end of the unstaking period.

```
# 05-epoch-scripts-events.md:
```

```
---
title: Query Epoch Info with Scripts or Events
sidebarlabel: Epoch Scripts and Events
tocmaxheadinglevel: 4
---
```

Introduction

The epoch contract stores a lot of different state, and the state is constantly changing. As an external party, there are two ways to keep track of these state changes. You can either use Cadence scripts to query the state of the contract at any given time, or you can monitor events that are emitted by the epoch contract to be notified of any important occurrences.

Monitor Epoch Service Events

These events can be queried using the Go or JavaScript SDKs to extract useful notifications and information about the state of the epoch preparation protocol.

What is a Service Event?

Service events are special messages that are generated by smart contracts and included in execution results. They enable communication between system smart contracts and the Flow protocol. In other words, they serve as a communication mechanism between the execution state and the protocol state.

Concretely, service events are defined and emitted as events like any other in Cadence. An event is considered a service event when it is:

- emitted within the service chunk
- emitted from a smart contract deployed to the service account
- conformant to an event allowlist

Each block contains a system chunk. For each system chunk, all service events emitted are included in the corresponding execution result.

When verifying the system chunk, verifier nodes will only produce result approvals when the system chunks included in the execution result are correct. Thus, the security of this communication mechanism is enforced by the verification system.

When sealing a block containing a service event, the consensus committee will update the protocol state accordingly, depending on the semantics of the event.

For example, a service event may indicate that a node's stake has diminished to the point where they should be ejected, in which case the consensus committee would mark that node as ejected in the protocol state.

Service events are fundamentally asynchronous, due the lag between block execution and sealing. Consequently they are handled slightly differently than other protocol state updates.

The diagram below illustrates the steps each service event goes through to be included in the protocol state.

!Flow Service Event Diagram

For conciseness, we say a service event is sealed when the block in which it was emitted is sealed, and we say a service event is finalized when the block containing the seal is finalized.

Event Descriptions

FlowEpoch.EpochStart

The Epoch Start service event is emitted by `FlowEpoch.startNewEpoch()` when the epoch commit phase ends and the Epoch Smart Contracts transition to the staking auction phase.

It contains the relevant metadata for the new epoch that was generated during the last epoch:

```
cadence
access(all) event EpochStart (

    /// The counter for the current epoch that is beginning
    counter: UInt64,

    /// The first view (inclusive) of the current epoch.
    firstView: UInt64,

    /// The last view (inclusive) of the current epoch's staking
    auction.
    stakingAuctionEndView: UInt64,

    /// The last view (inclusive) of the current epoch.
    finalView: UInt64,

    /// Total FLOW staked by all nodes and delegators for the current
    epoch.
    totalStaked: UFix64,

    /// Total supply of all FLOW for the current epoch
    /// Includes the rewards that will be paid for the previous epoch
    totalFlowSupply: UFix64,

    /// The total rewards that will be paid out at the end of the
    current epoch.
    totalRewards: UFix64,
)
```

FlowEpoch.EpochSetup

The Epoch Setup service event is emitted by `FlowEpoch.startEpochSetup()`

when the staking auction phase ends and the Epoch Smart Contracts transition to the Epoch Setup phase.

It contains the finalized identity table for the upcoming epoch, as well as timing information for phase changes.

```
cadence
access(all) event EpochSetup (
    /// The counter for the upcoming epoch. Must be one greater than the
    /// counter for the current epoch.
    counter: UInt64,

    /// Identity table for the upcoming epoch with all node information.
    /// Includes:
    /// nodeID, staking key, networking key, networking address, role,
    /// staking information, weight, and more.
    nodeInfo: [FlowIDTableStaking.NodeInfo],

    /// The first view (inclusive) of the upcoming epoch.
    firstView: UInt64,

    /// The last view (inclusive) of the upcoming epoch.
    finalView: UInt64,

    /// The cluster assignment for the upcoming epoch. Each element in
    /// the list
    /// represents one cluster and contains all the node IDs assigned to
    /// that
    /// cluster, with their weights and votes
    collectorClusters: [FlowClusterQC.Cluster],

    /// The source of randomness to seed the leader selection algorithm
    /// with
    /// for the upcoming epoch.
    randomSource: String,

    /// The deadlines of each phase in the DKG protocol to be completed
    /// in the upcoming
    /// EpochSetup phase. Deadlines are specified in terms of a consensus
    /// view number.
    /// When a DKG participant observes a finalized and sealed block with
    /// view greater
    /// than the given deadline, it can safely transition to the next
    /// phase.
    DKGPhase1FinalView: UInt64,
    DKGPhase2FinalView: UInt64,
    DKGPhase3FinalView: UInt64
)
```

FlowEpoch.EpochCommit

The EpochCommit service event is emitted when the Epoch Smart Contracts transition

from the Epoch Setup phase to the Epoch Commit phase.
It is emitted only when all preparation for the upcoming epoch (QC and DKG) has been completed.

```
cadence
access(all) event EpochCommit (

    /// The counter for the upcoming epoch. Must be equal to the counter
    in the
    /// previous EpochSetup event.
    counter: UInt64,

    /// The result of the QC aggregation process. Each element contains
    /// all the nodes and votes received for a particular cluster
    /// QC stands for quorum certificate that each cluster generates.
    clusterQCs: [FlowClusterQC.ClusterQC],

    /// The resulting public keys from the DKG process, encoded as by the
    flow-go
    /// crypto library, then hex-encoded.
    /// Group public key is the first element, followed by the individual
    keys
    dkgPubKeys: [String],
)
```

Query Information with Scripts

The FlowEpoch smart contract stores important metadata about the current, proposed, and previous epochs. Metadata for all historical epochs is stored permanently in the Epoch Smart Contract's storage.

```
cadence
access(all) struct EpochMetadata {

    /// The identifier for the epoch
    access(all) let counter: UInt64

    /// The seed used for generating the epoch setup
    access(all) let seed: String

    /// The first view of this epoch
    access(all) let startView: UInt64

    /// The last view of this epoch
    access(all) let endView: UInt64

    /// The last view of the staking auction
    access(all) let stakingEndView: UInt64

    /// The total rewards that are paid out for the epoch
}
```

```

access(all) var totalRewards: UFix64

    /// The reward amounts that are paid to each individual node and its
delegators
access(all) var rewardAmounts: [FlowIDTableStaking.RewardsBreakdown]

    /// Tracks if rewards have been paid for this epoch
access(all) var rewardsPaid: Bool

    /// The organization of collector node IDs into clusters
    /// determined by a round robin sorting algorithm
access(all) let collectorClusters: [FlowClusterQC.Cluster]

    /// The Quorum Certificates from the ClusterQC contract
access(all) var clusterQCs: [FlowClusterQC.ClusterQC]

    /// The public keys associated with the Distributed Key Generation
    /// process that consensus nodes participate in
    /// Group key is the last element at index: length - 1
access(all) var dkgKeys: [String]
}

```

Get Epoch Metadata

The FlowEpoch smart contract provides a public function, `FlowEpoch.getEpochMetadata()` to query the metadata for a particular epoch.

You can use the `Get Epoch Metadata(EP.01)` script with the following arguments:

Argument	Type	Description
epochCounter	UInt64	The counter of the epoch to get metadata for.

Get Configurable Metadata

The FlowEpoch smart contract also has a set of metadata that is configurable by the admin for phase lengths, number of collector clusters, and inflation percentage.

```

cadence
access(all) struct Config {
    /// The number of views in an entire epoch
    access(all) var numViewsInEpoch: UInt64

    /// The number of views in the staking auction
    access(all) var numViewsInStakingAuction: UInt64

    /// The number of views in each dkg phase
}

```

```

access(all) var numViewsInDKGPhase: UInt64
    /// The number of collector clusters in each epoch
access(all) var numCollectorClusters: UInt16

    /// Tracks the annualized percentage of FLOW total supply that is
    minted as rewards at the end of an epoch
    /// Calculation for a single epoch would be (totalSupply
    FLOWsupplyIncreasePercentage) / 52
    access(all) var FLOWsupplyIncreasePercentage: UFix64
}

```

You can use the Get Configurable Metadata(EP.02) script to get the list of configurable metadata:

This script does not require any arguments.

Get Epoch Counter

The FlowEpoch smart contract always tracks the counter of the current epoch.

You can use the Get Epoch Counter(EP.03) script to get the current epoch counter.

This script does not require any arguments.

Get Epoch Phase

The FlowEpoch smart contract always tracks the active phase of the current epoch.

```

cadence
access(all) enum EpochPhase: UInt8 {
    access(all) case STAKINGAUCTION
    access(all) case EPOCHSETUP
    access(all) case EPOCHCOMMIT
}

```

You can use the Get Epoch Phase(EP.04) script to get the current epoch phase.

This script does not require any arguments.

06-technical-overview.md:

```

---
title: Staking Technical Overview
sidebarlabel: Staking Technical Overview
description: Technical Overview of the Flow Staking Auction Phase
---
```

```
<Callout type="warning">
  If you haven't read the Introduction, please read that first. That
  document
    provides a non-technical overview of staking on Flow for all users and
    is a
      necessary prerequisite to this document.
</Callout>
<Callout type="warning">
  This document assumes you have some technical knowledge about the Flow
  blockchain and programming environment.
</Callout>
```

Staking

This document describes the functionality of the core identity table and staking smart contract. It gives an overview of the process of epochs, staking as a node, and delegation. It is an important prerequisite to understand before proceeding with any other technical integration or interaction with the Flow Protocol, but does not provide step-by-step instructions for how to perform specific actions. See the Staking Collection Docs for instructions

This document also describes how to read public staking data from the contract.

Anyone can read public data from the staking smart contract with these instructions.

The transactions described in this document are contained in the flow-core-contracts/transactions/idTableStaking/ directory. You can see the text of all the transactions used to interact with the smart contract there.

Smart Contract Summary

The Flow staking smart contract manages a record of stakers who have staked tokens for the network.

Users who want to stake can register with the staking contract at any time during the staking auction, and their tokens will be locked for staking until they request to unstake them.

You should already understand from reading the epoch documentation that an epoch lasts roughly a week. The FlowIDTableStaking contract focuses on the identity table and staking part of the epoch schedule.

Epoch Schedule from the perspective of the FlowIDTableStaking contract:

1. Start of Epoch: Generic metadata about the current epoch is updated and shared and the staking auction is enabled.

2. Staking Auction: Stakers can perform any action they want to manage their stake, like
 - initially registering, staking new tokens, unstaking tokens, or withdrawing rewards.
 This phase takes up the vast majority of time in the epoch.
3. End Staking Auction: Stakers cannot perform any more staking actions until the start of the next epoch/staking auction.
4. Remove Insufficiently Staked Nodes: All node operators who don't meet the minimum
 - or are not operating their node properly will be removed.
5. Randomly Assign Nodes to New Slots: Each node type has a configurable number of nodes that can operate during any given epoch.
 - The contract will randomly select nodes from the list of newly staked and approved nodes
 - to add them to the ID table. Once all the slots have been filled, the remaining nodes are refunded
 - and can apply again for the next epoch if there are slots available.
6. Rewards Calculation: Calculate rewards for all the node operators staked in the current epoch.
7. Move tokens between pools. (See the token pools section for the order of movements)
8. End Epoch: Set the reward payout for the upcoming epoch and go to the top of this list.
9. Rewards Payout: Pay rewards to all the node operators staked from the previous epoch using the calculation from earlier in the epoch.

The FlowIDTableStaking contract manages the identity table, and all of these phases.

Control of these phases is controlled by the `FlowIDTableStaking.Admin` resource

object stored in the Flow Epoch account storage.

The FlowEpoch smart contract uses this resource to autonomously manage the functioning of the network. It is decentralized and managed by the node software, smart contracts, and democratically by all the stakers in the network.

Staking as a Node Operator

For a node to stake, node operators first need to generate their staking key, staking key proof-of-possession, networking address, and networking key.

The node operation guide describes how to run a node and generate node information.

To generate a node ID, simply hash the staking key.

Node operators need to determine the role of node they will be running (Collection, Consensus, Execution, Verification, or Access).

<Callout type="warning">

NOTE: Access Nodes are eligible to stake and have a staking minimum of 100 FLOW,

```
but will not receive rewards for their stake.  
Please register as a different node type if you would like to receive  
rewards.  
</Callout>
```

Once the info has been determined:

- Node role: UInt8 (1 = Collection, 2 = Consensus, 3 = Execution, 4 = Verification, 5 = Access)
- Node ID: 32 byte String (64 hex characters)
- Networking Address: String (Length must be less than 510 characters and be a properly formatted IP address or hostname)
- Networking Key: 64 byte String (128 hex characters, must be a valid ECDSA-P256 Key)
- Staking Key: 96 byte String (192 hex characters, must be a valid BLS key)
- Staking Key Proof of Possession: (48 byte (96 hex characters) string)

The node operator is ready to register their node.

```
<Callout type="warning">  
NOTE: The staking smart contract validates that the strings for the  
keys are  
valid public keys. The staking admin and node software also checks the  
keys  
and networking address to make sure they are valid and if they are not,  
the  
registered node will not be eligible to stake.  
</Callout>
```

To register a node, the node operator calls the addNodeRecord function on the staking contract, providing all the node info and the tokens that they want to immediately stake, if any.

This registers the node in the Flow node identity table and commits the specified tokens to stake during the next epoch. This also returns a special node operator object that is stored in the node operator's account. This object is used for staking, unstaking, and withdrawing rewards.

Consensus and Collection nodes also need to create a separate machine account for use in the DKG and QC processes, respectively. This machine account creation is handled automatically by the staking collection smart contract. More information is in the machine account documentation.

```
<Callout type="warning">  
The register node transaction only needs to be submitted once per node.  
A node  
does not need to register every epoch. A registration cannot be used to  
manage
```

multiple nodes. Multiple nodes need to be registered separately (with the Staking Collection).
</Callout>

<Callout type="warning">
Once a node operator has registered their node and its metadata, the metadata cannot be modified. The only exception is the networking address, which can be modified with the Update Networking Address transaction. If a node operator wants to update any of their other metadata such as ID, keys, or role, they need to unstake, withdraw their tokens, and register a completely new node.
</Callout>

Once node operators have registered and have the special node object, they will be able to perform any of the valid staking options with it, assuming that they have the required amount of tokens to perform each operation.

When the staking auction ends, if a node operator has committed less than the minimum stake required, or if their node information is invalid and they haven't been approved by the network, their committed tokens are moved to their unstaked pool, which they can withdraw from at any time.

Nodes who did have enough tokens committed and are approved will have their committed tokens moved to the staked state at the end of the epoch if they are selected as a node operator by the random node slot filling algorithm. There is a configurable cap on the number of nodes of each type, so if the number of selected nodes equals the cap, than newly registered nodes will not be added to the network until the cap is raised or other nodes unstake.

If a node operator has users delegating to them, they cannot withdraw their own tokens such that their own staked tokens would fall below the minimum requirement for that node type. If they have delegators and try to submit an unstaking transaction that would put their stake below the minimum, it will fail.

If they want to unstake below the minimum, they must unstake all of their tokens using the special unstakeAll method, which also unstakes all of the tokens that have been delegated to them.

Consequently, a node operator cannot accept delegation unless their own stake is above the minimum.

Staking as a Delegator

Every staked non-access node in the Flow network is eligible for delegation by any other user.

The user only needs to know the node ID of the node they want to delegate to.

To register as a delegator, the delegator submits a Register Delegator transaction that calls the registerNewDelegator function, providing the ID of the node operator they want to delegate to. This transaction should store the NodeDelegator object in the user's account, which is what they use to perform staking operations.

Users are able to get a list of possible node IDs to delegate to via on-chain scripts.

This information will also be provided off-chain, directly from the node operators or via third-party services. Available node IDs are listed in a public repo.

The fee that node operators take from the rewards their delegators receive is 8%.

A node operator cannot be delegated to unless the total tokens they have committed to stake are above the minimum requirement for their node types.

The delegation logic keeps track of the amount of tokens each delegator has delegated for the node operator.

When rewards are paid, the protocol automatically takes the 8% cut of the delegator's rewards for the node operator and the delegator's rewards are deposited in the delegator's reward pool.

Staking Operations Available to All Stakers

Regardless of whether they are a node operator or delegator, a staker has access to all the same staking operations, outlined below. Specific implementations of these transactions are detailed in the Staking Collection Docs

Stake More Tokens

A staker can commit more tokens to stake for the next epoch at any time during the staking auction, and there are three different ways to do it.

1. They can commit new tokens to stake by submitting a stakenewtokens transaction,

which withdraws tokens from their account's flow token vault and commits them.

2. They can commit tokens that are in their unstaked token pool, which holds the tokens
 - that they have unstaked. Submit a `stakeunstakedtokens` transaction to move the tokens from the unstaked pool to the committed pool.
3. They can commit tokens that are in their rewarded token pool, which holds the tokens
 - they have been awarded. They submit a `stakerewardetokens` transaction to move the tokens from the rewards pool to the committed pool.

Cancel Committed Stake / Unstake Tokens

At any time during the staking auction, a staker can submit a request to unstake tokens with a `requestunstaking` transaction.
If there are tokens that have been committed but are not staked yet, they are moved to the unstaked pool and are available to withdraw.

If the requested tokens are in the staked pool, it marks the specified amount of tokens to be unstaked at the end of the epoch.

At the end of the epoch, the tokens are moved to the unstaking pool. They will sit in this pool for one (1) additional epoch, at which point they will be moved to the unstaked tokens pool.

Cancel an Unstake Request

Unstaking requests are not fulfilled until the end of the epoch where they are submitted, so a staker can cancel the unstaking request before it is carried out. A staker can do this by submitting a `stakeunstakedtokens` transaction, specifying the number of tokens of their unstake request they would like to cancel. If the specified number of tokens have been requested to unstake, the request will be canceled.

Withdraw Unstaked Tokens

At any time, stakers are able to freely withdraw from their unstaked tokens pool with the `withdrawunstaked` transaction.

Withdraw Rewarded Tokens

Staking rewards are paid out at the end of every epoch based on how many tokens are in a user's tokensStaked pool. Every staker's rewards are deposited into their rewarded tokens pool. Rewards can be withdrawn at any time by submitting a `withdrawrewardtokens` transaction.

These tokens are unlocked and can be transferred on-chain if desired, or re-staked.

The source code for the staking contract and more transactions

can be found in the Flow Core Contracts GitHub Repository.

Monitor Events from the Identity Table and Staking Contract

See the staking events document
for information about the events that can be emitted by the staking
contract.

Appendix

Token Pools

Each node operator has five token pools allocated to them:

- Committed Tokens: Tokens that are committed for the next epoch.
They are automatically moved to the staked pool when the next epoch starts.
- Staked Tokens: Tokens that are staked by the node operator for the current epoch.
They are only moved at the end of an epoch and if the staker has submitted an unstaking request.
- Unstaking Tokens: Tokens that have been unstaked, but are not free to withdraw until the following epoch.
- Unstaked Tokens: Tokens that are freely available to withdraw or re-stake.
Unstaked tokens go to this pool.
- Rewarded Tokens: Tokens that are freely available to withdraw or re-stake.
Rewards are paid and deposited to the rewarded Pool after each epoch.

At the end of every epoch, tokens are moved between pools in this order:

1. All committed tokens will get moved either to the staked tokens pool, or to the unstaked tokens pool (depending on if the registered node has met the minimum stake requirements).
2. All committed tokens get moved to staked tokens pool.
3. All unstaking tokens get moved to the unstaked tokens pool.
4. All requested unstaking tokens get moved from the staked pool to the unstaking pool.

```
# 07-staking-scripts-events.md:
```

```
---
```

```
title: Query Staking Info with Scripts or Events
sidebarlabel: Staking Scripts and Events
---
```

Introduction

The staking contract stores a lot of different state, and the state is constantly changing.

As an external party, there are two ways to keep track of these state changes.

You can either use Cadence scripts to query the state of the contract at any given time, or you can monitor events that are emitted by the staking contract to be notified of any important occurrences.

Query Information with Scripts

Get the list of proposed nodes for the next epoch:

`FlowIDTableStaking.getProposedNodeIDs()`: Returns an array of node IDs for proposed nodes.
Proposed nodes are nodes that have enough staked and committed for the next epoch to be above the minimum requirement and have been selected to participate in the next epoch.
This means that new access nodes that have not been selected with the random slot selection algorithm will not be included in this list.

You can use the Get Proposed Table(SC.05) script for retrieving this info.

This script requires no arguments.

Get the list of all nodes that are currently staked:

`FlowIDTableStaking.getStakedNodeIDs()` and
`FlowIDTableStaking.getParticipantNodeList()`:
Returns an array of nodeIDs that are currently staked.
Staked nodes are nodes that are staked and participating in the current epoch.

You can use the Get Current Table(SC.04) script for retrieving this info.

This script requires no arguments.

Get the list of all Candidate Nodes

`getCandidateNodeList(): {UInt8: {String: Bool}}`:
Returns a dictionary of nodes that are candidates to stake in the next epoch but are not staked in the current epoch.

You can use the Get Candidate Node List script for retrieving this info.

This script requires no arguments.

Get all of the info associated with a single node staker:

`FlowIDTableStaking.NodeInfo(nodeID: String)`: Returns a `NodeInfo` struct with all of the metadata associated with the specified node ID. You can see the `NodeInfo` definition in the `FlowIDTableStaking` smart contract.

You can use the Get Node Info(SC.08) script with the following arguments:

Argument	Type	Description
nodeID	String	The node ID of the node to search for.

You can also query the info from an address that uses the staking collection by using the Get Node Info From Address(SCO.15) script with the following arguments:

Argument	Type	Description
address	Address	The address of the account that manages the nodes.

Get the total committed balance of a node (with delegators):

FlowIDTableStaking.NodeInfo(nodeID: String).totalCommittedWithDelegators(): Returns the total committed balance for a node, which is their total tokens staked + committed, plus all of the staked + committed tokens of all their delegators.

You can use the Get Node Total Commitment(SC.09) script with the following argument:

Argument	Type	Description
nodeID	String	The node ID of the node to search for.

Get the total committed balance of a node (without delegators):

FlowIDTableStaking.NodeInfo(nodeID: String).totalCommittedWithoutDelegators(): Returns the total committed balance for a node, which is their total tokens staked + committed, plus all of the staked + committed tokens of all their delegators.

You can use the Get Only Node Total Commitment(SC.11) script with the following argument:

Argument	Type	Description
nodeID	String	The node ID of the node to search for.

Get all the info associated with a single delegator:

FlowIDTableStaking.DelegatorInfo(nodeID: String, delegatorID: UInt32): Returns a DelegatorInfo struct with all of the metadata

associated with the specified node ID and delegator ID. You can see the `DelegatorInfo` definition in the `FlowIDTableStaking` smart contract.

You can use the `Get Delegator Info`(SC.10) script with the following arguments:

Argument	Type	Description
--		
<code>nodeID</code>	String	The node ID that the delegator delegates to.
<code>delegatorID</code>	String	The ID of the delegator to search for.

You can also query the info from an address by using the `Get Delegator Info From Address`(SC.16) script with the following arguments:

Argument	Type	Description
--		
<code>address</code>	Address	The address of the account that manages the delegator.

Get the delegation cut percentage:

`FlowIDTableStaking.getRewardCutPercentage()`: `UFix64`: Returns a `UFix64` number for the cut of delegator rewards that each node operator takes.

You can use the `Get Cut Percentage`(SC.01) script to retrieve this info.

This script requires no arguments.

Get the minimum stake requirements:

`FlowIDTableStaking.getMinimumStakeRequirements()`: `{UInt8: UFix64}`: Returns a mapping for the stake requirements for each node type.

You can use the `Get stake requirements`(SC.02) script to retrieve this info.

This script requires no arguments.

Get the total weekly reward payout:

`FlowIDTableStaking.getEpochTokenPayout()`: `UFix64`: Returns a `UFix64` value for the total number of FLOW paid out each epoch (week).

You can use the `Get weekly payout`(SC.03) script to retrieve this info.

This script requires no arguments.

Get the total FLOW staked:

You can use the Get total FLOW staked(SC.06) script to retrieve this info.

This script requires no arguments.

Get the total FLOW staked by all the nodes of a single node role:

You can use the Get total FLOW staked by node type(SC.07) script with the following arguments:

Argument	Type	Description
nodeType	UInt8	The type of node to search for.

Staking Events

Staking events can be queried using the Go or JavaScript SDKs to extract useful notifications and information about the state of the staking process.

Global Staking and Epoch Events

NewEpoch

```
cadence
access(all) event NewEpoch(totalStaked: UFix64, totalRewardPayout: UFix64, newEpochCounter: UInt64)
```

Field	Type	Description
totalStaked	UFix64	The total number of tokens staked for the new Epoch
totalRewardPayout	UFix64	The total number of tokens that will be paid as rewards for this epoch
newEpochCounter	UInt64	The epoch counter for this new epoch

Emitted by FlowIDTableStaking.Admin.moveTokens() when the tokens are moved between pools, which signals a new epoch.

NewWeeklyPayout

```
cadence
access(all) event NewWeeklyPayout(newPayout: UFix64)
```

Field	Type	Description

newPayout UFix64 The new number of tokens that will be paid at the end of the epoch

Emitted by `FlowIDTableStaking.Admin.setEpochTokenPayout()` when the Admin changes the total tokens paid at the end of the epoch.

After this event the `epochTokenPayout` is equal to the new value.

Node Events

These are events that concern the operation of a node.

NewNodeCreated

cadence
access(all) event `NewNodeCreated(nodeID: String, role: UInt8, amountCommitted: UFix64)`

Field	Type	Description
nodeID String The unique ID string for the node. 32 bytes. Usually the hash of the node's public key.		
role UInt8 The node's role type. From 1 to 5 inclusive.		
amountCommitted UFix64 The amount of FLOW tokens staked to register the node. This is determined by the role.		

Emitted by `FlowIDTableStaking.NodeRecord.init()` when a new node is successfully created.

After this event is emitted for your node, you can begin to perform staking transactions using it.

NodeRemovedAndRefunded

cadence
access(all) event `NodeRemovedAndRefunded(nodeID: String, amount: UFix64)`

Field	Type	Description
nodeID String The unique ID string for the node. 32 bytes. The same value emitted in the <code>NewNodeCreated</code> event for the node.		
amount UFix64 The amount of FLOW tokens returned to the node.		

Emitted by `FlowIDTableStaking.Admin.endStakingAuction()` if the node is being removed from the next epoch due to a failure to meet the minimum requirements of committed tokens for the next epoch.

After this event, the refunded FLOW tokens will be part of the node's tokensUnstaked balance.

Token Events

These are events that concern the direct usage of FLOW tokens - staking or unstaking locked tokens, withdrawing rewards, etc.

Events emitted when using delegation are described in the next section.

TokensCommitted

cadence

```
access(all) event TokensCommitted(nodeID: String, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
amount	UFix64	The amount of additional FLOW tokens committed to the node.

Emitted whenever additional tokens are staked on the node for the following epoch. Specifically:

1. By FlowIDTableStaking.NodeStaker.stakeNewTokens() when new tokens (tokens that have not previously been staked) are added to the system to stake on the node during the next epoch.
2. By FlowIDTableStaking.NodeStaker.stakeUnstakedTokens() when unstaked tokens (tokens that were previously staked and then unstaked) are staked again with the node for the next epoch.
3. By FlowIDTableStaking.NodeStaker.stakeRewardedTokens() when reward tokens (tokens paid in return for previous staking) are staked with the node for the next epoch.

After this event, the FLOW tokens will be part of the node's tokensCommitted balance.

TokensStaked

cadence

```
access(all) event TokensStaked(nodeID: String, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
amount	UFix64	The amount of FLOW tokens staked to the node.

Emitted by FlowIDTableStaking.Admin.moveTokens() at the end of an epoch if committed tokens are being added to the node's tokensStaked balance.

After this event, the tokens will be part of the node's staked balance.

TokensUnstaking

cadence

```
access(all) event TokensUnstaking(nodeID: String, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
amount	UFix64	The amount of FLOW tokens unstaked from the node.

Emitted by FlowIDTableStaking.Admin.moveTokens() at the end of an epoch if

a node operator's staked tokens are being unstaked in response to a request from the node operator.

After this event, the tokens will be a part of the node operator's tokensUnstaking balance, where they are held for a whole epoch "unstaking period" with no rewards.

TokensUnstaked

cadence

```
access(all) event TokensUnstaked(nodeID: String, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
amount	UFix64	The amount of FLOW tokens unstaked from the node.

Emitted by FlowIDTableStaking.NodeStaker.requestUnstaking() and FlowIDTableStaking.Admin.moveTokens()

when tokens are deposited into the tokensUnstaked pool:

RewardsPaid

cadence

```
access(all) event RewardsPaid(nodeID: String, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
amount	UFix64	The amount of FLOW tokens paid to the node this epoch as a reward.

Emitted by FlowIDTableStaking.Admin.payRewards() at the end of the epoch to pay rewards to node operators based on the tokens that they have staked.

After this event, the reward tokens will be part of the node's tokensRewarded balance.

The Delegator rewards are paid at the same time, see DelegatorRewardsPaid below.

UnstakedTokensWithdrawn

cadence
access(all) event UnstakedTokensWithdrawn(nodeID: String, amount: UFix64)

Field	Type	Description
nodeID String The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.		
amount UFix64 The amount of unstaked FLOW tokens that the node operator is withdrawing.		

Emitted by FlowIDTableStaking.NodeStaker.withdrawUnstakedTokens() when the node operator calls that function to withdraw part or all of their unstaked tokens balance.

After this event, the FLOW tokens will be withdrawn to a newly created FungibleToken.Vault which the caller can deposit to the vault of their choice.

RewardTokensWithdrawn

cadence
access(all) event RewardTokensWithdrawn(nodeID: String, amount: UFix64)

Field	Type	Description
nodeID String The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.		
amount UFix64 The amount of rewarded FLOW tokens that the node operator is withdrawing.		

Emitted by FlowIDTableStaking.NodeStaker.withdrawRewardedTokens() when the node operator calls that function to withdraw part or all of their reward tokens balance.

After this event, the FLOW tokens will be withdrawn to a newly created FungibleToken.Vault which the caller can deposit to the vault of their choice.

Delegator Events

These are events that concern FLOW token delegation.

NewDelegatorCreated

cadence

```
access(all) event NewDelegatorCreated(nodeID: String, delegatorID: UInt32)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UFix64	The ID for the new delegator. Unique within the node but not globally.

Emitted by `FlowIDTableStaking.Admin.registerNewDelegator()` when the node operator registers a new delegator for the node.

Note that the delegatorID is unique within the node but is not globally unique.

After this event, the new delegator is registered with the node.

DelegatorTokensCommitted

cadence

```
access(all) event DelegatorTokensCommitted(nodeID: String, delegatorID: UInt32, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UInt32	The ID for the delegator.
amount	UFix64	The amount of additional FLOW tokens committed to the node.

Emitted whenever additional tokens are committed for a delegator for the following epoch. Specifically:

1. By `FlowIDTableStaking.NodeDelegator.delegateNewTokens()` when new tokens (tokens that have not previously been staked) are added to the system
 - to stake with the delegator during the next epoch.
2. By `FlowIDTableStaking.NodeDelegator.delegateUnstakedTokens()` when unstaked tokens (tokens that were previously staked and then unstaked) are staked again with the delegator for the next epoch.
3. By `FlowIDTableStaking.NodeDelegator.delegateRewardedTokens()` when reward tokens (tokens paid in return for previous staking) are staked with the delegator for the next epoch.

After this event, the FLOW tokens will be part of the delegator's tokensCommitted balance.

DelegatorTokensStaked

cadence
access(all) event DelegatorTokensStaked(nodeID: String, delegatorID: UInt32, amount: UFix64)

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UInt32	The ID for the delegator.
amount	UFix64	The amount of FLOW tokens staked to the node.

Emitted by FlowIDTableStaking.Admin.moveTokens() at the end of an epoch if committed tokens are being added to the delegator's tokensStaked balance.

After this event, the tokens will be part of the delegator's staked balance.

DelegatorTokensUnstaking

cadence
access(all) event DelegatorTokensUnstaking(nodeID: String, delegatorID: UInt32, amount: UFix64)

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UInt32	The ID for the delegator.
amount	UFix64	The amount of FLOW tokens unstaked from the node.

Emitted by FlowIDTableStaking.Admin.moveTokens() at the end of an epoch if a delegator's staked tokens are being unstaked in response to a request from the delegator. After this event, the tokens will be a part of the delegator's tokensUnstaking balance, where they are held for a whole epoch "unstaking period" with no rewards.

DelegatorTokensUnstaked

cadence
access(all) event DelegatorTokensUnstaked(nodeID: String, delegatorID: UInt32, amount: UFix64)

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UInt32	The ID for the delegator.
amount	UFix64	The amount of FLOW tokens unstaked from the node.

Emitted by FlowIDTableStaking.NodeDelegator.requestUnstaking() and FlowIDTableStaking.Admin.moveTokens()
when tokens are deposited into the delegator's tokensUnstaked pool:

DelegatorRewardsPaid

cadence
access(all) event DelegatorRewardsPaid(nodeID: String, delegatorID: UInt32, amount: UFix64)

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UFix64	The ID for the delegator. Unique within the node but not globally.
amount	UFix64	The amount of rewarded FLOW tokens that the delegator is paid.

Emitted by FlowIDTableStaking.Admin.payRewards() at the end of an epoch when rewards are being paid.

After this event is emitted, the reward tokens will be part of the delegator's tokensRewarded balance.

The Node rewards are paid at the same time, see RewardsPaid above.

DelegatorUnstakedTokensWithdrawn

cadence
access(all) event DelegatorUnstakedTokensWithdrawn(nodeID: String, delegatorID: UInt32, amount: UFix64)

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UFix64	The ID for the delegator. Unique within the node but not globally.
amount	UFix64	The amount of unstaked FLOW tokens that the delegator is withdrawing.

Emitted by FlowIDTableStaking.NodeDelegator.withdrawUnstakedTokens() when the delegator calls that function to withdraw part or all of their

unstaked tokens balance.

After this event, the FLOW tokens will be withdrawn to a newly created FungibleToken.Vault which the caller can deposit to the vault of their choice.

DelegatorRewardTokensWithdrawn

cadence

```
access(all) event DelegatorRewardTokensWithdrawn(nodeID: String,  
delegatorID: UInt32, amount: UFix64)
```

Field	Type	Description
nodeID	String	The unique ID string for the node. 32 bytes. The same value emitted in the NewNodeCreated event for the node.
delegatorID	UFix64	The ID for the delegator. Unique within the node but not globally.
amount	UFix64	The amount of rewarded FLOW tokens that the delegator is withdrawing.

Emitted by FlowIDTableStaking.NodeDelegator.withdrawRewardedTokens() when the delegator calls that function to withdraw part or all of their unstaked tokens balance.

After this event, the FLOW tokens will be withdrawn to a newly created FungibleToken.Vault which the caller can deposit to the vault of their choice.

08-staking-rewards.md:

```
title: Staking and Delegation rewards  
sidebarlabel: How to Query Staking rewards  
description: How to check the staking and delegation rewards
```

Current method to check staking rewards

Rewards payout happens automatically after the end of the epoch and without the need of an explicit transaction being submitted by the service account.

Instead of a separate reward payout transaction, the reward payout events will be recorded in the system chunk in the block that is produced at the time of the epoch transition without creating a regular transaction ID.

The rewards payout can be queried by querying the block which contains the system chunk that contains the reward payout events.

```
flow events get A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid  
A.8624b52f9ddcd04a.FlowIDTableStaking.DelegatorRewardsPaid --start <block  
Height> --end <block height> -n mainnet
```

where block height is the height of the block containing the rewards payout events

Example

```
$ flow events get A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
A.8624b52f9ddcd04a.FlowIDTableStaking.DelegatorRewardsPaid --start
51753836 --end 51753836 -n mainnet

Events Block #51753836:
  Index 6
  Type   A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
  Tx ID  f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
  Values
    - nodeID (String):
      "a3075cf9280cab4fa0b7b1e639b675bdae3e8874557d98ee78963f0799338a5f"
    - amount (UFix64): 1660.21200000

  Index 9
  Type   A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
  Tx ID  f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
  Values
    - nodeID (String):
      "cf0ff514b6aa659914b99ab1d17743edb2b69fbb338ab01945a08530a98c97d4"
    - amount (UFix64): 3762.20370347

  Index 12
  Type   A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
  Tx ID  f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
  Values
    - nodeID (String):
      "de988efc8cb79d02876b7beffd404fc24b61c287ebeede567f90056f0eece90f"
    - amount (UFix64): 939.85630919

  Index 27
  Type   A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
  Tx ID  f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
  Values
    - nodeID (String):
      "fa5f24a66c2f177ebc09b8b51429e9f157037880290e7858f4336479e57dc26b"
    - amount (UFix64): 1660.21200000

  Index 30
  Type   A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid
  Tx ID  f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
  Values
```

```

        - nodeID (String):
"581525fa93d8fe4b334c179698c6e72bacb802593e55e40da61d24e589d85be"
        - amount (UFix64): 1937.24727662
        ...
        ...
<clipped for brevity>
        ...
        ...
Index 50115
Type A.8624b52f9ddcd04a.FlowIDTableStaking.DelegatorRewardsPaid
Tx ID
f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
Values
        - nodeID (String):
"95ffacf0c05757cff71a4ee49e025d5a6d1103a3aa7d91253079e1fb7c22458"
        - delegatorID (UInt32): 23
        - amount (UFix64): 0.10424555

Index 50118
Type A.8624b52f9ddcd04a.FlowIDTableStaking.DelegatorRewardsPaid
Tx ID
f31815934bff124e332b3c8be5e1c7a949532707251a9f2f81def8cc9f3d1458
Values
        - nodeID (String):
"95ffacf0c05757cff71a4ee49e025d5a6d1103a3aa7d91253079e1fb7c22458"
        - delegatorID (UInt32): 18
        - amount (UFix64): 17.31047712

```

Example using Flow Go SDK

```

package main

import (
    "context"
    "fmt"
    client "github.com/onflow/flow-go-sdk/access/grpc"
)

func main() {

    // the Flow testnet community Access node API endpoint
    accessNodeAddress := "access.mainnet.nodes.onflow.org:9000"

    // create a gRPC client for the Access node
    accessNodeClient, err := client.NewClient(accessNodeAddress)
    if err != nil {
        fmt.Println("err:", err.Error())
        panic(err)
    }

    ctx := context.Background()

    blockEvents, err := accessNodeClient.GetEventsForHeightRange(ctx,

```

```

        "A.8624b52f9ddcd04a.FlowIDTableStaking.RewardsPaid",
        51753836,
        51753836)
    if err != nil {
        panic(err)
    }

    for , blockEvent := range blockEvents {
        fmt.Println("Block: " + blockEvent.BlockID.String())
        for , event := range blockEvent.Events {
            fmt.Println("\tEvent type: " + event.Type)
            fmt.Println("\tEvent: " + event.Value.String())
            fmt.Println("\tEvent payload: " + string(event.Payload))
        }
    }
}

```

Check staking rewards before May 2023

Before May 2023, rewards payouts were done manually by the Flow governance committee.

When the transactions executed, they generated events for the rewards paid to each node and delegator.

To check the staking and delegation rewards, those transactions should be queried directly.

Example using Flow cli

```
$ flow transactions get
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84 -n
mainnet
```

Status	✓ SEALED
ID	84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Payer	e467b9dd11fa00df
Authorizers	[e467b9dd11fa00df]

Proposal Key:

Address	e467b9dd11fa00df
Index	11
Sequence	118

No Payload Signatures

```
Envelope Signature 0: e467b9dd11fa00df
Envelope Signature 1: e467b9dd11fa00df
Envelope Signature 2: e467b9dd11fa00df
Envelope Signature 3: e467b9dd11fa00df
Envelope Signature 4: e467b9dd11fa00df
Signatures (minimized, use --include signatures)
```

```

Events:
Index 0
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 64.59694884
- from (Address?): 0xf919ee77447b7497

Index 1
Type A.f919ee77447b7497.FlowFees.TokensWithdrawn
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 64.59694884

Index 2
Type A.1654653399040a61.FlowToken.TokensMinted
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 1326397.40305116

Index 3
Type A.1654653399040a61.FlowToken.TokensDeposited
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 1326397.40305116
- to (Never?): nil

Index 4
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 1004.16460872
- from (Never?): nil

Index 5
Type A.1654653399040a61.FlowToken.TokensDeposited
Tx ID
84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b6382cc84
Values
- amount (UFix64): 1004.16460872
- to (Address?): 0x8624b52f9ddcd04a
...
...
<clipped for brevity>

```

Example using Flow Go SDK

```
package main
```

```

import (
    "context"
    "fmt"
    "github.com/onflow/flow-go-sdk"
    client "github.com/onflow/flow-go-sdk/access/grpc"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
)
func main() {
    // the Flow mainnet community Access node API endpoint
    accessNodeAddress := "access.mainnet.nodes.onflow.org:9000"

    maxGRPCMessageSize := 1024 1024 20 // to accommodate for the
    large transaction payload

    // create a gRPC client for the Access node
    accessNodeClient, err := client.NewClient(accessNodeAddress,
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithDefaultCallOptions(grpc.MaxCallRecvMsgSize(maxGRPCMessageS
ize)))
    if err != nil {
        fmt.Println("err:", err.Error())
        panic(err)
    }

    ctx := context.Background()

    txID :=
flow.HexToID("84eca4ff612ef70047d60510710cca872c8a17c1bd9f63686e74852b638
2cc84")

    rewardsTxResult, err := accessNodeClient.GetTransactionResult(ctx,
txID)
    if err != nil {
        panic(err)
    }

    for , event := range rewardsTxResult.Events {
        fmt.Println("Event type: " + event.Type)
        fmt.Println("Event: " + event.Value.String())
        fmt.Println("Event payload: " + string(event.Payload))
    }
}

# 09-qc-dkg.md:

---
title: Quorum Certificate and Distributed Key Generation
sidebarlabel: QC and DKG

```

```
description: How the Flow protocol manages the Epoch Setup Phase
tocmaxheadinglevel: 4
---
:::warning

If you haven't read the Intro to Flow Staking document and the Epoch
protocol document,
please read that first. Those documents provide an overview of epochs
on Flow for
all users and are necessary prerequisites to this document.

:::

:::warning

This document assumes you have some technical knowledge about the Flow
blockchain and programming environment.

:::

Epoch Setup Phase

Purpose: During the epoch setup phase, all nodes participating in the
upcoming epoch
must perform setup tasks in preparation for the upcoming epoch, including
the Collector Cluster Quorum Certificate Generation and Consensus
Committee Distributed Key Generation.

Duration: The epoch setup phase begins right after the EpochSetup service
event is emitted.
It ends with the block where EpochCommit service emitted.

Machine Accounts

The processes described in this document are fully automated.

They are expected to be performed entirely by the node software with no
manual
interaction required by the node operator after the node has been set up
and registered.

To facilitate this, we recommend that node operators use a secondary
"machine account"
that only stores the FlowClusterQC.Voter or FlowDKG.Participant resource
objects
in addition to FLOW to pay for transaction fees. You can connect your
node to this account
to participate in the Epoch Setup Phase without having to do the actions
manually.

If you are using the Staking Collection for your node,
this functionality is built-in. When you register a node with the staking
collection,
```

you also have to provide a public key or keys for your machine account for the node.

If you have a node without a machine account (if you were operating a node from the time before epochs and staking collection were enabled, for example) the staking collection also provides a method to create a machine account for an existing node.

See the Staking Collection Docs for more information.

Collector Cluster Quorum Certificate Generation Protocol

The collector nodes are organized into clusters and must bootstrap the Hotstuff consensus algorithm for each cluster. To do this, they generate the root block for their respective clusters and submit a vote for the root block to a specialized smart contract, FlowClusterQC.

If 2/3 of the collectors in a cluster have voted with the same unique vote, then the cluster is considered complete. Once all clusters are complete, the QC is complete.

FlowClusterQC Transactions

Create QC Voter Object

A node uses the `getClusterQCVoter()` function in the `FlowEpoch` contract to create their Voter object and needs to provide a reference to their `FlowIDTableStaking.NodeStaker` object to prove they are the node owner.

When registering a node with the staking collection, this process is handled by the transaction to register.

It also creates a machine account for the QC object.

If a user already has a registered node with the staking collection, but hasn't created their QC Voter object yet, they can use the `createmachineaccount.cdc` transaction.

If a user is not using the staking collection, they can use the Create QC Voter (QC.01) transaction. This will only store the QC Voter object in the account that stores the `NodeStaker` object. It does not create a machine account or store it elsewhere, so it is not recommended to use. We encourage to use the staking collection instead.

Submit Vote

During the Epoch Setup Phase, the node software should submit the votes for the QC generation

automatically using the Submit QC Vote (QC.02) transaction with the following arguments.

Argument	Type	Description
voteSignature	String	The signed message (signed with the node's staking key)
voteMessage	String	The raw message itself.

Consensus Committee Distributed Key Generation Protocol (DKG)

The Random Beacon Committee for the next Epoch (currently all consensus nodes)

will run the DKG through a specialized smart contract, FlowDKG.

To do this, they post a series of messages to a public "whiteboard" to collectively generate a shared key array. When each node has enough information

to generate their array of keys, they send the generated array to the smart contract

as their final submission.

If $(\text{of consensus nodes}-1)/2$ consensus nodes submit the same key array, the DKG is considered to be complete.

FlowDKG Transactions

Create DKG Participant Object

A node uses the `getDKGParticipant()` function in the `FlowEpoch` contract to create their `Voter` object and needs to provide a reference to their `FlowIDTableStaking.NodeStaker` object to prove they are the node owner.

When registering a node with the staking collection, this process is handled by the transaction to register.

It also creates a machine account for the DKG Object.

If a user already has a registered node with the staking collection, but hasn't created their DKG Participant object yet, they can use the `createmachineaccount.cdc` transaction.

If a user is not using the staking collection, they can use the Create DKG Participant (DKG.01)

transaction. This will only store the DKG Participant object in the account that stores the `NodeStaker` object.

It does not create a machine account or store it elsewhere, so it is not recommended to use.

The staking collection is the recommended method.

Post Whiteboard Message

During the Epoch Setup Phase, the node software should post whiteboard messages to the DKG

automatically using the Post Whiteboard Message (DKG.02) transaction with the following arguments.

Argument	Type	Description
content	String	The content of the whiteboard message

Send Final Submission

During the Epoch Setup Phase, the node software should send its final submission for the DKG automatically using the Send Final Submission (DKG.03) transaction with the following arguments.

Argument	Type	Description
submission	[String?]	The final key vector submission for the DKG.

Monitor Events and Query State from the QC and DKG Contracts

See the QC and DKG events and scripts document for information about the events that can be emitted by these contracts and scripts you can use to query information.

```
# 10-qc-dkg-scripts-events.md:
```

```
---
```

```
title: Query QC/DKG Info with Scripts or Events
```

```
sidebarlabel: QC/DKG Scripts and Events
```

```
---
```

Introduction

The Cluster Quorum Certificate (QC) and Distributed Key Generation (DKG) protocol smart contracts store a lot of different state, and the state is constantly changing. As an external party, there are two ways to keep track of these state changes. You can either use Cadence scripts to query the state of the contract at any given time, or you can monitor events that are emitted by the contracts to be notified of any important occurrences.

Query Information with Scripts

These events can be queried using the Go or JavaScript SDKs to extract useful notifications and information about the state of these processes.

QC Scripts

These scripts allow anyone to query information about the state of the QC contract.

Get Clusters

To return a struct representing the information associated with a collector cluster, can use the Get Cluster (QC.03) script with the following argument:

Argument	Type	Description
clusterIndex	UInt16	The index of the cluster to query

Get QC Enabled

To return a boolean representing if the QC is enabled, can use the Get QC Enabled (QC.04) script with no arguments.

Get Node Has Voted

To return a boolean representing if a node has voted for the current QC, you can use the Get Node Has Voted (QC.05) script with the following argument:

Argument	Type	Description
nodeID	String	The node ID to check for

Get Voting Complete

To return a boolean representing if the voting for the QC phase is complete, can use the Get Voting Complete (QC.06) script with no arguments.

QC Events

Documentation coming soon

DKG Scripts

Get DKG Enabled

To return a boolean representing if the DKG is enabled, you can use the Get DKG Enabled (DKG.04) script with no arguments.

Get DKG Completed

To return a boolean representing if the dkg is complete, you can use the Get DKG Complete (DKG.05) script with no arguments.

Get Whiteboard Messages

To return an array of structs representing all the whiteboard messages, you can use the Get DKG Whiteboard Messages (DKG.06) script with no arguments.

Get Final Submissions

To return an array of key vectors for the nodes' final submissions, you can use the Get Final Submissions (DKG.07) script with no arguments.

Get Node Has Submitted

To return a boolean representing if a node has sent their final submission for the DKG, you can use the Get Node Has Submitted (DKG.08) script with the following argument:

Argument	Type	Description
nodeID	String	The node ID to check for

DKG Events

```
cadence
/// Emitted when the admin enables the DKG
access(all) event StartDKG()

/// Emitted when the admin ends the DKG after enough submissions have
been recorded
access(all) event EndDKG(finalSubmission: [String?]?)

/// Emitted when a consensus node has posted a message to the DKG
whiteboard
access(all) event BroadcastMessage(nodeID: String, content: String)
```

11-machine-account.md:

```
---
title: Machine Account
sidebarlabel: Machine Account
description: Usage and Purpose of the Machine Account
---
```

What is a Machine Account?

A Machine Account is a Flow account which is used autonomously by a node to interact with system smart contracts. Machine Accounts contain Cadence resources granted to network participants which are used to participate in smart-contract-mediated protocols. Currently, Machine Accounts are used in the Epoch Preparation Protocol, which prepares the network for the next epoch.

Your Machine Account is distinct from the account you use for staking your node (your Staking Account).

The Machine Account is intended for use by node software and does not have access to your staked tokens or staking rewards.

```
<Callout type="info">
```

Currently Machine Accounts are required only for collection and consensus nodes.

```
</Callout>
```

Creation

Machine Accounts are created during the staking process in Flow Port.

Funding

Machine Accounts must maintain a balance of liquid FLOW tokens to pay fees on transactions they submit to system smart contracts. Typically very few transactions will be sent (about 1-5 per week) however more may be required under certain circumstances and network conditions.

```
<Callout type="info">
```

Because some transactions sent by the Machine Account are system critical, we recommend maintaining a balance sufficient to accommodate worst-case transaction submission numbers at all times. See here for how to monitor.

```
</Callout>
```

When creating a new machine account, we recommend initially funding with 0.005 FLOW for collection nodes and 0.25 FLOW for consensus nodes.

Machine account balances should be monitored and periodically refilled to ensure they have sufficient funds.

We recommend a minimum balance at all times of 0.002 FLOW for collection nodes and 0.1 FLOW for consensus nodes.

A node operator can easily withdraw their FLOW from their machine account if they decide they don't need them there any more.

```
# 12-faq.md:
```

```
---
```

```
title: Staking FAQ
```

```
sidebarlabel: FAQs
```

```
description: Frequently Asked Questions
```

Where will users receive their staking reward for each staking option?

Staking rewards are not deposited directly into a user's account. They are deposited to the user's rewards pool in their connected staking object and can be withdrawn or restaked at any time.

If you staked using Flow Port, then you should be able to see your staking rewards there.

You can also withdraw the rewards or manually re-stake them through Flow Port.

If you staked using a staking provider such as Kraken, Blocto or Finoa, please ask them how they manage staking rewards.

Will staking rewards be automatically re-staked?

There will be no automatic re-staking of staking rewards with Flow Port (i.e. using Ledger or Blocto).

If you want to re-stake your rewards, you must manually do so yourself.

If you staked using a staking provider such as Kraken, Blocto or Finoa, please ask them what their policies are.

DeFi liquid staking strategies such as offered by incrementFi are not managed by the protocol or nodes, but are more sophisticated ways to manage your staking.

Does it make a difference as to what TYPE of node we delegate to in terms of rewards?

No, rewards are calculated the same for every node type.

Can a validator change its fees?

The network enforces a delegation fee of 8% which cannot be directly changed.

Any different fees that nodes claim they have are rebates that they offer using their own methods and are not enforced by the protocol.

Can a token holder stake to multiple nodes? If yes, how is the stake split between them?

A token holder can delegate to multiple nodes from a single account if they use the Staking Collection.

The staking collection is enabled by default on Flow port, and many custody providers also support it.

Is the wallet key transferred to the staked node?

No - The keys on the node are different from the wallet keys. The wallet keys always stay in the wallet.

A node operator generates the staking and networking keys separately which will be used on the node.

Can I stake through multiple accounts to meet the minimum FLOW required for staking a node?

No, the minimum stake must come from a single account for all node types. Temporarily, the minimum for consensus nodes can come from a combination of staking actions from two accounts controlled by the same party.

How can I reach the Consensus node minimum stake of 500K FLOW

The consensus node minimum of 500K FLOW can be met with a minimum 250,000 FLOW staking action and additional delegation adding a minimum of 250,000 more FLOW from the same entity.

Is rewards payout another spork?

No, rewards payout is not a spork but is an automatic transaction that happens at the beginning of every new epoch.

Can I query an account address of a node ID or delegator ID?

The staking smart contract does not directly associate a node or delegator with an account address.

It associates it with the assigned resource object that corresponds to that entry in the contract.

There can be any number of these objects stored in the same account, and they can be moved to different accounts if the owner chooses.

It is possible to query the information about a node that an address runs though, by using the `getnodeinfofromaddress.cdc` script.

13-staking-options.md:

```
---
title: Options for Building Staking Integrations
sidebarlabel: Technical Staking Options
---
```

This document describes two different methods for staking at a high level.

<Callout type="warning">

We highly recommended you use the Staking Collection paradigm, as this will be the most supported method for staking with any set up.

</Callout>

Staking Collection

A Staking Collection is a resource that allows its owner to manage multiple staking objects in a single account via a single storage path, and perform staking actions using Flow. It also supports machine accounts, a necessary feature for Flow epoch node operation.

The staking collection paradigm is the most flexible of the three choices and will receive the most support in the future. It is the set-up that Flow Port and many other staking providers use.

The staking collection setup and guide is detailed in the staking collection guide.

Staking

The basic method to stake is to stake directly with the FlowIDTableStaking smart contract. This would involve calling the node or delegator registration functions directly in the staking contract and storing the staking objects directly in account storage.

This is probably the simplest way to implement this, but it is not very flexible and not recommended.

The basic staking guide is detailed here

14-staking-collection.md:

```
---
title: Manage a Staking Collection
sidebarlabel: Staking Collection Guide
---
```

This document outlines the steps a token holder can take to stake using the FlowIDTableStaking contract and the FlowStakingCollection contract. This is the recommended and most supported way to stake FLOW. It supports any number of nodes and delegators per account, supports locked and unlocked FLOW, and supports easily interaction with a node's machine account for collector and consensus nodes.

Staking Collection Overview

A Staking Collection is a resource that allows its owner to manage multiple staking objects in a single account via a single storage path, and perform staking actions using both locked and unlocked Flow.

Before the staking collection, accounts could use the instructions in

the unlocked staking guide to stake with tokens. This was a bit restrictive, because that guide (and the corresponding transactions) only supports one node and one delegator object per account. If a user wanted to have more than one per account, they would either have to use custom transactions with custom storage paths for each object, or they would have had to use multiple accounts, which comes with many hassles of its own.

The staking collection is a solution to both of these deficiencies. When an account is set up to use a staking collection, the staking collection recognizes the existing locked account capabilities (if they exist) and unlocked account staking objects, and incorporates their functionality so any user can stake for a node or stake as a delegator through a single common interface, regardless of if they have a brand new account, or have been staking through the locked account or unlocked account before.

The staking collection also easily allows a user to transfer their node or delegator objects to other accounts without interrupting the staking process!

Staker Object Fields

The staking collection resource has two main fields, cadence
access(self) var nodeStakers: @{String: FlowIDTableStaking.NodeStaker}
access(self) var nodeDelegators: @{String: FlowIDTableStaking.NodeDelegator}

These dictionaries store the staking objects that are managed by the staking collection. Access to these dictionaries are mediated by the staking methods. When a user wants to perform a staking operation, they specify the nodeID and/or delegatorID they want to stake for, and the function routes the function call to the correct staking object and performs the specified operation.

Vault Capability Fields

The staking collection also has a field that stores a capability for the unlocked FLOW Vault and locked FLOW vault (if applicable) cadence
/// unlocked vault
access(self) var unlockedVault: Capability<&FlowToken.Vault>

/// locked vault
/// will be nil if the account has no corresponding locked account
access(self) var lockedVault: Capability<&FlowToken.Vault>?

When a user performs staking operations like staking new tokens, the staking collection tracks the number of unlocked tokens and locked tokens (if applicable) that are used by the staking objects in the collection.

The collection will always try to stake any available locked tokens first.

Once all locked tokens are staked, if the user requested to stake more than the locked token balance, the collection will then dip into the unlocked balance for the remaining tokens.

If the user has no locked tokens, the staking collection will simply ignore the locked tokens part of the functionality and only use unlocked tokens.

When a user withdraws tokens from a staking object, the collection will always try to withdraw unlocked tokens first.

Any unlocked tokens are then deposited directly into the vault on the unlocked account,

and remaining locked tokens are deposited to the vault in the locked account.

Machine Account Support

The staking collection also supports an important feature for epochs, machine accounts.

Machine accounts are where node operators store important resource objects

that are critical to the functionality of the epoch preparation protocol. Every collector and consensus node should have an associated machine account that stores these objects, and the staking collection helps the user create and manage these accounts.

When a user registers a new collector or consensus node, the staking collection also creates a machine account for them and registers

the required object that needs to go in the machine account.

The node operator is then responsible for adding keys to the account. (the Register Node transaction includes this step).

Once the machine account is created and set up, the node operator just has to

connect it to their node software and make sure the account has enough FLOW

to pay for transaction fees, which can be handled simply by submitting a regular FLOW transfer to the machine account's address

Staking Collection Public Getter Methods

The staking collection also defines many getter methods to query information

about an account's staking collection. You can simply call one of these methods on the contract,

```

providing the account address, and the contract will retrieve
the relevant info for you, like so:
cadence
import FlowStakingCollection from 0xSTAKINGCOLLECTIONADDRESS
import FlowIDTableStaking from 0xIDENTITYTABLEADDRESS

/// Gets an array of all the delegator metadata for delegators stored in
the staking collection
access(all) fun main(address: Address):
[FlowIDTableStaking.DelegatorInfo] {
    return FlowStakingCollection.getAllDelegatorInfo(address: address)
}

```

Remember: A Staking Collection does not require an account to have a secondary locked account or locked FLOW. However, if an account does have an associated locked account, when the Staking Collection is initialized, it will connect to that locked account's node and delegator objects as well as its locked token vault allowing it to perform staking actions with locked and unlocked FLOW.

```

<Callout type="info">
Staking Collection is backwards compatible with other methods of staking
on Flow.
Existing accounts with associated locked accounts
will still be able to stake in the same way as before,
but they will also be able to use the staking collection, if desired.
</Callout>

```

How to use the Staking Collection

There is a standard set of transactions provided with the staking collection.

Setup

Setup a Staking Collection

To set up a Staking Collection, you must run the Setup Staking Collection (SCO.01) transaction.

This transaction requires no arguments and will perform the following actions:

1. Create private capabilities for the unlocked vault and locked vault (if applicable).
2. Create a new staking collection resource object, initializing it with the unlocked and locked vault capabilities.
3. Store the staking collection at a pre-defined storage path.
4. Create a public link to the staking collection so others can query metadata about it.
5. If there are any node or delegator objects in the unlocked account, the transaction stores those in the staking collection

so they can be used through the same interface as usual.

No arguments are required for the Setup Staking Collection transaction.

Once this transaction is complete, your existing staking objects (if any) from your unlocked account and locked account will be available via the staking collection and you can use all the transactions described below to access them.

Create a Machine Account for an existing Node

Many nodes will have been created before the staking collection was set up and before epochs were enabled, meaning that they don't already have an associated machine account. These nodes need a new transaction to create the machine account for the node and save it to the staking collection.

To create a machine account for a node that doesn't already have one, you must submit the Create Machine Account (SCO.03) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The ID of the node.
publicKeys	[String]	The public keys to add to the machine account.

If the node is a collector or consensus node, this transaction creates the associated machine account, registers the QC or DKG object, stores it in the machine account, and adds the provided public key(s) to the machine account. If no public keys are provided, the transaction will fail.

Register Stakers

Register a New Staked Node

To register a new staked node, you must submit the Register Node (SCO.03) transaction with the following arguments:

Argument	Type	Description
id	String	The ID of the new node. It must be a 32 byte String. The operator is free to choose this value, but it must be unique across all nodes. A recommended process to generate this is to hash the staking public key.
role	UInt8	The role of the new node. (1: collection, 2: consensus, 3: execution, 4: verification, 5: access)
networkingAddress	String	The IP address of the new node.
networkingKey	String	The networking public key as a hex-encoded string.
stakingKey	String	The staking public key as a hex-encoded string.
stakingKeyPoP	String	The staking key Proof-of-Possession as a hex-encoded string.

amount	UFix64	The number of FLOW tokens to stake.
publicKeys	[String]?	The public keys to add to the machine account. nil if no machine account

This transaction registers the account as a staker with the specified node information and attaches a NodeStaker resource to the Staking Collection. This NodeStaker resource can then later be used to perform staking actions via the staking collection staking methods.

If the node is a collector or consensus node, it also creates the associated machine account, registers the QC or DKG object, stores it in the machine account, and adds the provided public key(s) to the machine account. If the node requires a machine account and no public keys are provided, the transaction will fail.

Once the account has registered their node using their Staking Collection, their tokens and node information are committed to the central staking contract for the next epoch.

At this point, the Staking Collection now has access to various staking operations that they can perform, assuming they have the correct number of tokens to perform the action.

Register a New Staked Delegator

To register a new delegator, you must submit the Register Delegator (SCO.02) transaction with the following arguments:

Argument	Type	Description
id	String	The ID of the node to delegate to.
amount	UFix64	The number of FLOW tokens to delegate.

This transaction registers the account as a delegator to the node identified by the supplied node id. It also attaches a NodeDelegator resource to the Staking Collection. This NodeDelegator resource can then later be used to perform delegation actions.

Once the account has registered their new delegator using their Staking Collection, their tokens are committed to the central staking contract for the next epoch.

At this point, the Staking Collection now has access to various delegator operations that they can perform, assuming they have the correct number of tokens to perform the action.

Staking Operations

These transactions perform actions that directly interact with the staking contract.

Most of them will only succeed during the Staking Auction phase of the epoch.

Stake New Tokens

The Staking Collection can stake additional tokens for any Node or Delegator managed by it at any time.

The owner of a Staking Collection can use the Stake New Tokens (SCO.06) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to stake new tokens to.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.
amount	UFix64	The number of FLOW tokens to stake.

<Callout type="info">
To stake new tokens for an active node, leave the **delegatorID** argument as **nil**.

If staking for a delegator, **delegatorID** should be the delegator ID you are staking for.

</Callout>

The amount may be any number of tokens up to the sum of an accounts locked and unlocked FLOW.

Re-stake Unstaked Tokens

After tokens become unstaked, the owner of a Staking Collection can choose to re-stake the unstaked tokens to the same Node or Delegator.

The owner of a Staking Collection can use the Stake Unstaked Tokens (SCO.08) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to stake the unstaked tokens to.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.
amount	UFix64	The number of FLOW tokens to restake.

```
<Callout type="info">
To stake unstaked tokens for an active node, leave the <b>delegatorID</b>
arguement as <b>nil</b>.
```

```
If staking for a delegator, <b>delegatorID</b> should be the delegator ID
you are staking for.
</Callout>
```

Re-stake Rewarded Tokens

After earning rewards from staking, the owner of a Staking Collection can choose to re-stake the rewarded tokens to the same node or delegator.

The owner of a Staking Collection can use the Stake Unstaked Tokens (SCO.07) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to stake the rewarded tokens to.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.
amount	UFix64	The number of FLOW tokens to restake.

```
<Callout type="info">
To stake rewarded tokens for an active node, leave the <b>delegatorID</b>
arguement as <b>nil</b>.
</Callout>
```

Request to Unstake Tokens at the end of the Epoch

The owner of a Staking Collection can submit a request to unstake their tokens at any time for any Node or Delegator in their collection.

If the tokens aren't staked yet, they will be uncommitted and available to withdraw.

Note: unstaked tokens will be held by the central staking contract until the end of the following epoch.

Once the tokens are released (unstaked), they can be claimed via the Withdraw Unstaked Tokens action below.

The owner of a Staking Collection can use the Unstake Tokens (SCO.05) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the chosen node.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.

amount	UFix64	The number of FLOW tokens to restake.
--------	--------	---------------------------------------

<Callout type="info">
To unstake tokens from an active node, leave the delegatorID arguement as nil. </Callout>

Unstake All Tokens

The owner of a Staking Collection can use the Unstake All (SCO.09) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to unstake all tokens from.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.

Withdraw Unstaked Tokens

After tokens for an active Node or Delegator become unstaked, the ownder of Staking Collection can withdraw them from the central staking contract.

The owner of a Staking Collection can use the Withdraw Unstaked Tokens (SCO.11) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to withdraw the unstaked tokens from.
delegatorID	Optional(UInt32)	nil if staking for a node. If staking for a delegator, the delegator ID.
amount	UFix64	The number of FLOW tokens to withdraw.

<Callout type="info">
To withdraw unstaked tokens from an active node, leave the delegatorID arguement as nil. </Callout>

Withdraw Rewarded Tokens

After earning rewards from staking, the token holder can withdraw them from the central staking contract.

The owner of a Staking Collection can use the Withdraw Rewarded Tokens (SCO.10) transaction with the following arguments:

Argument	Type	Description
----------	------	-------------

----- ----- -----
nodeID String The nodeID of the node to withdraw the rewarded tokens from.
delegatorID Optional(UInt32) nil if staking for a node. If staking for a delegator, the delegator ID.
amount UFix64 The number of FLOW tokens to withdraw.

<Callout type="info">
To withdraw rewarded tokens from an active node, leave the delegatorID arguement as nil. </Callout>

Staking Collection Modification

Close a Node or Delegator

Once a Node or Delegator has no tokens staked, committed or in an unstaking state, it is eligible to be closed.

Closing a Node or Delegator first returns any unstaked or rewarded tokens to the account for which the Staking Collection is stored in. It then destroys the NodeStaker or NodeDelegator object from within the Staking Collection.

Note: Once a Node or Delegator has been closed, it cannot be accessed again, and no staking or delegation actions can be further preformed on it.

The owner of a Staking Collection can use the Close Stake (SCO.12) transaction with the following arguments:

Argument	Type	Description
----- ----- -----		
nodeID String The nodeID of the node to close.		
delegatorID Optional(UInt32) nil if staking for a node. If staking for a delegator, the delegator ID.		

<Callout type="info">
To close an active node, leave the delegatorID arguement as nil. </Callout>

Transfer a Node

A user may transfer an existing Node to another another account's Staking Collection.

The account to transfer the Node to must have a valid Staking Collection set up.

Transferring a Node will remove it from the authorizer's Staking Collection

and deposit it to the receiver's Staking Collection.

Note: Once a Node or Delegator has been transferred, it cannot be accessed again by the sender.

As well, all staked tokens will be considered staked by the receiver's Staking Collection.

```
<Callout type="warning">
Transferring a Node will result in loss of custody of any Staked tokens
for the sender.
</Callout>
```

The owner of a Staking Collection can use the Transfer Node (SCO.13) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the node to transfer.
to	Address	The address of the account which contains the Staking Collection to transfer the Node to.

Transfer a Delegator

A user may transfer an existing Delegator to another another account's Staking Collection.

The account to transfer the Delegator to must have a valid Staking Collection set up.

Transferring a Delegator will remove it from the authorizer's Staking Collection and deposit it to the receiver's Staking Collection.

Note: Once a Node or Delegator has been transferred, it cannot be accessed again by the sender.

As well, all staked tokens will be considered staked by the receiver's Staking Collection.

```
<Callout type="warning">
Transferring a Delegator will result in loss of custody of any Staked tokens
for the sender.
</Callout>
```

The owner of a Staking Collection can use the Transfer Delegator (SCO.14) transaction with the following arguments:

Argument	Type	Description
nodeID	String	The nodeID of the delegator to transfer.
delegatorID	UInt32	The delegatorID of the delegator to transfer.

to	Address	The address of the account which contains the Staking Collection to transfer the Delegator to.
----	---------	------------------------------------------------------------------------------------------------

Update A Node's Networking Address

A user may update their node's networking address if it has become inconsistent with the protocol state.

This operation can only be performed in the staking auction phase of an epoch.

Note: Currently, if a node updates its networking address and the new address does not match what is stored in the protocol state for the node, the node will not be able to participate in the upcoming epoch. Only update your networking address if you have already confirmed with the Flow team that you can. This restriction will be removed once fully automated epochs are completely implemented.

The owner of a Staking Collection can use the Update Networking Address (SCO.22) transaction with the following arguments:

Argument	Type	Description
-----	-----	-----
nodeID	String	The nodeID of the node to update.
newAddress	String	The new networking address

Staking Collection Scripts

These scripts allow anyone to query information about an account's staking collection.

Get All Node Info

To return an array of structs representing the information associated with each node managed by an account's Staking Collection, anyone can use the Get All Node Info (SCO.15) script with the following arguments:

Argument	Type	Description
-----	-----	-----
address	Addresss	The Address of the account holding the Staking Collection to query from

This script returns an array of FlowIDTableStaking.NodeInfo structs representing the nodes managed by an accounts Staking Collection.

Get All Delegator Info

To return an array of structs representing the information associated with each delegator managed by an account's Staking Collection, anyone can use the Get All Delegator Info (SCO.16) script with the following arguments:

Argument	Type	Description
address	Addressss	The Address of the account holding the Staking Collection to query from

This script returns an array of FlowIDTableStaking.DelegatorInfo structs representing the delegators managed by an accounts Staking Collection.

Get All Node Ids

To return an array of Strings representing the ids associated with each node managed by an account's Staking Collection, anyone can use the Get All Node Ids (SCO.17) script with the following arguments:

Argument	Type	Description
address	Addressss	The Address of the account holding the Staking Collection to query from

This script returns an array of String representing each id of each node managed by an accounts Staking Collection.

Get All Delegator Ids

To return an array of structs representing the delegator ids associated with each delegation managed by an account's Staking Collection, anyone can use the Get All Delegator Ids (SCO.22) script with the following arguments:

Argument	Type	Description
address	Addressss	The Address of the account holding the Staking Collection to query from

This script returns an array of FlowStakingCollection.DelegatorIDs structs representing the delegator Ids of each delegator managed by an accounts Staking Collection.

Get Locked Tokens Used

To query how many Locked FLOW tokens an account has staked using their Staking Collection, anyone can use the Get Locked Tokens Used (SCO.19) script with the following arguments:

Argument	Type	Description

-----	-----	-----
address	Addressss	The Address of the account holding the Staking Collection to query from

This script returns a UFix64 representing the number of Locked FLOW tokens staked using an accounts Staking Collection.

```
<Callout type="info">
Note: This number does not include Locked FLOW tokens staked not through
an accounts Staking Collection.
</Callout>
```

Get Unlocked Tokens Used

To query how many Unlocked FLOW tokens an account has staked using their Staking Collection, anyone can use the Get Unlocked Tokens Used (SCO.20) script with the following arguments:

Argument	Type	Description
-----	-----	-----
address	Addressss	The Address of the account holding the Staking Collection to query from

This script returns a UFix64 representing the number of Unlocked FLOW tokens staked using an accounts Staking Collection.

```
<Callout type="info">
Note: This number does not include Unlocked FLOW tokens staked not
through an accounts Staking Collection.
</Callout>
```

Get Does Node Exist

To query if a Node or Delegator is managed by an accounts Staking Collection, anyone can use the Get Does Node Exist (SCO.21) script with the following arguments:

Argument	Type	Description
-----	-----	-----
address	Addressss	The Address of the account holding the Staking Collection to query from
nodeID	String	The nodeID of the node to check, or the nodeID of the node delegating to to check.
delegatorID	Optional(UInt32)	The delegatorID of the delegator to check, if checking for a delegator.

This script returns a Bool.

```
<Callout type="info">
To query if a Node is managed by an accounts Staking Collection, leave
the <b>delegatorID</b> arguement as <b>nil</b>.
Otherwise, fill it in with the <b>delegatorID</b> of the Delegator.
```

```
</Callout>
```

Get Machine Account Info

To query the machine account information for an account's staking collection, anyone can use the Get Machine Account Info (SCO.21) script with the following arguments:

Argument	Type	Description
address	Address	The Address of the account holding the Staking Collection to query from

This script returns a {String: FlowStakingCollection.MachineAccountInfo}, which is a mapping of nodeIDs to the FlowStakingCollection.MachineAccountInfo struct.

```
# 15-staking-guide.md:
```

```
---
title: Basic Staking with FLOW
sidebarlabel: Basic Staking Guide (Deprecated)
---
```

This document outlines the steps a token holder can take to stake and manage a Flow node with FLOW using only the types defined in the FlowIDTableStaking contract. It only supports having one node or delegator object per account and is not supported by ledger and will likely not be supported by other wallets, so it is recommended to use the staking collection instead.

```
<Callout type="warning">
This guide covers staking with FLOW tokens.
</Callout>
```

Staking

Setup

Register a New Staked Node

To register as a node operator with FLOW, the token holder can use the Register Node (SC.11) transaction with the following arguments:

Argument	Type	Description
id	String	The ID of the new node. It must be a 32 byte String. The operator is free to choose this value, but it must be

```

unique across all nodes. A recommended process to generate this is to
hash the staking public key. |
| role | UInt8 | The role of the new node. (1: collection,
2: consensus, 3: execution, 4: verification, 5: access) |
| networkingAddress | String | The IP address of the new node. (Length
must be less than 255 bytes (510 Hex characters)) |
| networkingKey | String | The networking public key as a 64 byte
hex-encoded String (128 hex characters) |
| stakingKey | String | The staking public key as a 96 byte hex-
encoded String (192 hex characters) |
| amount | UFix64 | The number of FLOW tokens to stake. |

```

This transaction registers the account as a node operator with the specified node information and creates a public link to query the nodes ID from the account address.

Once the token holder has registered their node, their tokens and node information are committed to the central staking contract for the next epoch.

At this point, the token holder now has access to various staking operations that they can perform, assuming they have the correct number of tokens to perform the action.

Stake Tokens

The token holder can stake additional tokens at any time.

Note: this transaction stakes additional tokens to the same node that was registered in the setup phase.

To stake tokens, the token holder can use the Stake FLOW (SC.12) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of FLOW tokens to stake.

This transaction commits tokens to stake from the token holder's account.

Re-stake Unstaked Tokens

After tokens become unstaked, the token holder can choose to re-stake the unstaked tokens to the same node.

To staked unstaked tokens, the token holder can use the Re-stake Unstaked FLOW (SC.13) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of unstaked FLOW tokens to stake.

Re-stake Rewarded Tokens

After earning rewards from staking, the token holder can choose to re-stake the rewarded tokens to the same node.

To stake rewarded tokens, the token holder can use the Re-stake Rewarded FLOW (SC.14) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of rewarded FLOW tokens to stake.

Request Unstake Tokens

The token holder can submit a request to unstake some of their tokens at any time.

If the tokens aren't staked yet, they will be uncommitted and available to withdraw.

To request to unstake staked tokens, the token holder can use the Request Unstaking (SC.15) transaction.

Argument	Type	Description
amount	UFix64	The number of rewarded FLOW tokens to request to unstake.

Note: this transaction will not succeed if the node operator has delegators and the request would put the node operator below the minimum required tokens staked for their node type.

Use the Unstake All transaction instead, which will also unstake all delegators.

Note: unstaked tokens will be held by the central staking contract until the end of the following epoch.

Once the tokens are released (unstaked), they can be claimed via the Withdraw Unstaked Tokens action below.

Unstake All Tokens

The token holder can submit a request to unstake all their tokens at any time.

If the tokens aren't staked yet, they will be uncommitted and available to withdraw.

To unstake all staked tokens, the token holder can use the Unstake All FLOW (SC.16) transaction.

This transaction requires no arguments.

Warning: this will unstake all of the user's staked tokens and unstake all of the tokens from users that are delegating FLOW to the node.

Withdraw Unstaked Tokens

After tokens become unstaked, the token holder can withdraw them from the central staking contract.

To withdraw unstaked tokens, the token holder can use the Withdraw Unstaked FLOW (SC.17) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of unstaked FLOW tokens to withdraw.

This transaction moves the unstaked tokens back into the FlowToken.Vault owned by the token holder.

Withdraw Rewarded Tokens

After earning rewards from staking, the token holder can withdraw them from the central staking contract.

To withdraw rewarded tokens, the token holder can use the Withdraw Rewarded FLOW (SC.18) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of rewarded FLOW tokens to withdraw.

This transaction moves the rewarded tokens back into the FlowToken.Vault owned by the token holder.

Stake Multiple Nodes from the Same Account

Currently, the default staking transactions can only be used as they are to stake one node per account.

If a token holder wants to create a second staking relationship using the transactions as is, they must create a new account and transfer their tokens to the new account.

It is possible to have multiple nodes per account by storing the node objects at different storage paths, but this would require small changes to these transactions to use the new storage paths.

Delegating

Setup

Register as a Delegator

To register as a delegator, the token holder can use the Register Delegator (SC.19) transaction with the following arguments:

Argument	Type	Description
id	String	The ID of the node to delegate to.
amount	UFix64	The number of FLOW tokens to delegate.

This transaction registers the account as a delegator to the node ID they specified.

Delegate New Tokens

The token holder can delegate additional tokens after registering as a delegator.

Note: this transaction delegates additional tokens to the same node that was registered in the setup phase.

To delegate new tokens,
the token holder can use the Delegate New FLOW (SC.20) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of FLOW tokens to delegate.

Re-delegate Unstaked Tokens

After delegated tokens become unstaked, the token holder can choose to re-delegate the unstaked tokens to the same node.

To delegate unstaked tokens,
the token holder can use the Re-delegate Unstaked FLOW (SC.21) transaction with the following arguments:

Argument	Type	Description
amount	UFix64	The number of unstaked FLOW tokens to delegate.

Re-delegate Rewarded Tokens

After earning rewards from delegation, the token holder can choose to re-delegate the rewarded tokens to the same node.

To delegate rewarded tokens,
the token holder can use the Re-delegate Rewarded FLOW (SC.22)

Argument	Type	Description
----------	------	-------------

amount UFix64 The number of rewarded FLOW tokens to delegate.

Unstake Delegated Tokens

The token holder can submit a request to unstake their delegated tokens at any time.

If the tokens aren't staked yet, they will be uncommitted and available to withdraw.

To unstake delegated tokens,

the token holder can use the Unstake Delegated FOW (SC.23)

Argument	Type	Description
amount UFix64 The number of FLOW tokens to unstake.		

Note: unstaked delegated tokens will be held by the central staking contract for a period of time

(the rest of the current epoch plus all of the next epoch) before they are

released to the token holder. Once the tokens are released (unstaked), they can be claimed via the Withdraw Unstaked Tokens action below.

Withdraw Unstaked Tokens

After delegated tokens become unstaked, the token holder can withdraw them from the central staking contract.

To withdraw unstaked tokens,

the token holder can use the Withdraw Unstaked FLOW (SC.24) transaction with the following arguments:

Argument	Type	Description
amount UFix64 The number of unstaked FLOW tokens to withdraw.		

This transaction moves the unstaked tokens back into the FlowToken.Vault owned by the token holder.

Withdraw Rewarded Tokens

After earning rewards from delegation, the token holder can withdraw them from the central staking contract.

To withdraw rewarded tokens,

the token holder can use the Withdraw Rewarded FLOW (SC.25) transaction with the following arguments:

Argument	Type	Description
amount UFix64 The number of rewarded FLOW tokens to withdraw.		

This transaction moves the rewarded tokens back into the FlowToken.Vault owned by the token holder.

Delegate to Multiple Nodes from the Same Account

Currently, the default delegating transactions can only be used as they are to stake one node per account.

If a token holder wants to create a second delegating relationship using the transactions as is, they must create a new account and transfer their tokens to the new account.

It is possible to have multiple delegator objects per account by storing the node objects at different storage paths, but this would require small changes to these transactions to use the new storage paths.

```
# index.md:
```

```
---
```

```
title: Epochs, Staking & Delegating on Flow
sidebarlabel: Staking and Epochs
sidebarposition: 1
description: Introduction to how staking works on Flow
---
```

This document provides an introduction to staking FLOW tokens on the Flow network for token holders and node operators. Staking is an important part of the security protocol of a proof-of-stake (PoS) blockchain. Running nodes and staking tokens contributes to the blockchain's security and is rewarded accordingly.

What is Staking?

Flow is a global network of computers working together to maintain the security and integrity of its users' data.

This global network is made up of many individual nodes: software applications run by people. Every node in the network shares a small part of the responsibility to keep the network running smoothly and to ensure that other nodes are doing the same. This shared responsibility is a core premise of decentralization, because no single central node is solely responsible for the security and integrity of the network and the data it contains.

Node operators are what we call the people who run nodes. In order to connect their software applications as nodes on the network, a node operator must first purchase tokens. Every node operator has to temporarily give (or 'stake') a large number of their tokens to the network as a promise that they will not modify their node

to do something that is against the rules of the network, like steal funds from users' accounts.

This process of temporarily giving up tokens is called staking.

If a node ever breaks the rules defined by the network, a number of the node operator's staked tokens will be taken from them as a punishment.

This process is automatic. Every node knows the rules defined by the network

and automatically watches other nodes and reports them if they misbehave. Meanwhile, the network pays the node operator a reward from a mixture of transaction fees and newly minted tokens on a regular basis provided their node does not break the rules.

If a node operator breaks the rules, they lose the tokens they've staked. If they operate their node with integrity, they get rewarded with more tokens!

This is the basic incentive that enables a decentralized proof-of-stake network, like Flow.

How Does Staking Work on Flow?

The Flow protocol maintains a list of node operators.

The list contains important information about each node, like their public keys, node address, and what kind of node they are running.

(Collection, Consensus, Execution, Verification, or Access)

A node operator registers a node by submitting a transaction containing their node information, a cryptographic proof that they control their node info,

and the FLOW they wish to stake.

If they meet the requirements to run a node, then will be accepted to join the network!

Once a node is staking and operating properly, it will receive periodic reward payments,

assuming it stays online and actively participates in the protocol without committing any actions that would harm the network, which we call slashable offenses.

Once nodes have registered, they are required to operate for a protocol-specified timeframe.

This timeframe is otherwise known as an Epoch.

Epochs

An Epoch is a roughly week-long period that the network uses to manage list of nodes and pay rewards.

- Only a pre-determined set of nodes is authorized to participate in the protocol.

The set of authorized nodes is known to all network participants. This set is referred to as the Identity Table.

- An Epoch is defined as a period of time, where the set of authorized nodes is constant
(or can only shrink due to ejection of malicious nodes).

Every epoch, a list of committed nodes are chosen to be the staked nodes of the network.

This list is called the Identity Table (ID Table).

The node's staked tokens are locked in and cannot change for the duration of the epoch.

At the end of the epoch, rewards are paid to each staked node based on how many tokens they had staked for that epoch and how well they performed during the epoch. Nodes can choose to join or leave, but changes to the Identity Table can only happen at end of an epoch, which is also the beginning of a new epoch.

This process repeats itself indefinitely, as long as the network remains functioning.

To determine the list of nodes that are included as officially staked nodes in the next epoch,

the protocol looks at the records of all the nodes that have committed tokens.

It checks to make sure each node's information is correct and that the node is running properly.

Each node also has to have committed tokens above the minimum stake required for their node role

and be authorized by the service account.

If any of these checks are insufficient, the node is not included in the next epoch.

Every epoch, some nodes also have to perform certain processes to initialize the state and communication

with other nodes for the next epoch. These processes are called Cluster Quorum Certificate Generation (QC),

and Distributed Key Generation (DKG). If any node does not perform this initialization properly,

it is not included in the next epoch's Identity Table.

If a node passes all the checks and initializations, it is approved and included as an official node for the next epoch.

Nodes (and users who delegate to them) do not have to continue to submit staking registration transactions every epoch in order to remain staked. As long as they continue to run their node properly, their tokens will remain staked.

A node operator only needs to take action if they want to stake more tokens

or if they want to unstake their staked tokens.

If a node operator or delegator decides to stake or unstake tokens, their requests are not carried out until the end of the current epoch. In the case of unstaking requests, they also must wait an additional epoch before their unstaked tokens are available to withdraw.

This allows the protocol to deal with any slashable offenses that may have happened in the previous epoch.

See the Epochs section of the documentation for in-depth explanations of the identity table, epoch schedule, QC, and DKG.

Rewards

Please see the schedule section of the documentation for information about reward calculations and schedule and what you can do with the rewards you earn by staking a node!

Delegation

Any account in the network may also participate in staking by delegating their tokens to a node operator.

Every node operator in the network is eligible to receive delegations, there is no opting out.

To delegate to a node, a user simply specifies the ID of the node they want to delegate to and the amount of tokens they want to delegate. The tokens are committed and managed in the exact same way that normal staked tokens are managed.

Rewards for delegators are also calculated in the exact same way that rewards for node operators are calculated, with one difference in that 8% of the calculated amount is given to the delegatee (the node being delegated to). The remaining 92% is awarded to the delegator.

How Do I Stake?

So you have decided you want to be a part of the Flow network? Welcome! You are joining a group of people from all around the world that are a part of a movement that is bringing decentralization and transparency into the world.

Staking using Flow Port

Flow Port is a simple browser-based app for the Flow blockchain that provides functionality for sending, receiving, and staking tokens. Any wallet that uses the Flow Client Library is compatible with Flow Port.

If you created your account using Flow Port, you can also stake and earn rewards using the Flow Port. Follow this step-by-step guide to stake using Flow Port. Flow Port currently supports staking as a node, delegating, and reward withdrawal using Flow Reference Wallet, Ledger, Shadow, NuFi, and any other FCL compatible accounts / wallets.

Staking via a Custody Provider

If you are using a custody provider who controls your account and private keys for you, such as Kraken, Finoa, or Coinlist, they all have different policies and processes for what you need to do to stake your tokens, the rewards you receive, and the fees that they take from your staking rewards.

Manual Staking or Building your own Staking Integration

If you are self-custodying your Flow account and keys, or you want to build a staking service for customers, you will need to learn more about how staking works, the various methods for staking, and how you can participate safely and reliably. See the staking technical overview first for information about technical integration.

```
# error-codes.md:
```

```
---
```

```
sidebarposition: 2
```

```
---
```

Error Codes

List of error codes returned from failing transactions and scripts. The error code has an accompanied error message that usually gives more clarification. This list is meant to give more information and helpful hints.

Code file

```
1006
ErrCodeInvalidProposalSignatureError
```

Example:

```
...
```

```
1007
ErrCodeInvalidProposalSeqNumberError
```

Example:

```
[Error Code: 1007] invalid proposal key: public key 0 on account xxx has
sequence number xxx, but given xxx
```

1008
ErrCodeInvalidPayloadSignatureError

Example:

[Error Code: 1008] invalid payload signature: public key 0 on account xxx does not have a valid signature: signature is not valid

1009
ErrCodeInvalidEnvelopeSignatureError

Example:

[Error Code: 1009] invalid envelope key: public key 1 on account xxx does not have a valid signature: signature is not valid

1051
ErrCodeValueError

Example:

[Error Code: 1051] invalid value (xxx): invalid encoded public key value: rlp: expected input list for flow.runtimeAccountPublicKeyWrapper...

1052
ErrCodeInvalidArgumentError

Example:

[Error Code: 1052] transaction arguments are invalid: (argument is not json decodable: failed to decode value: runtime error: slice bounds out of range [:2] with length 0)

1053
ErrCodeInvalidAddressError

Example:

...

```
1054
ErrCodeInvalidLocationError
```

Example:

```
[Error Code: 1054] location (../contracts/FungibleToken.cdc) is not a
valid location: expecting an AddressLocation, but other location types
are passed ../contracts/FungibleToken.cdc
```

```
1055
ErrCodeAccountAuthorizationError
```

Example:

```
[Error Code: 1055] authorization failed for account e85d442d61a611d8:
payer account does not have sufficient signatures (1 < 1000)
```

```
1056
ErrCodeOperationAuthorizationError
```

Example:

```
[Error Code: 1056] (RemoveContract) is not authorized: removing contracts
requires authorization from specific accounts goroutine 5688834491
[running]:
```

```
1057
ErrCodeOperationNotSupportedError
```

Example:

...

```
1101
ErrCodeCadenceRunTimeError
```

Example:

[Error Code: 1101] cadence runtime error Execution failed: error: precondition failed: Amount withdrawn must be less than or equal than the balance of the Vault

1103
ErrCodeStorageCapacityExceeded

Example:

[Error Code: 1103] The account with address (xxx) uses 96559611 bytes of storage which is over its capacity (96554500 bytes). Capacity can be increased by adding FLOW tokens to the account.

For more information refer to Fees

1105
ErrCodeEventLimitExceededError

Example:

[Error Code: 1105] total event byte size (256200) exceeds limit (256000)

1106
ErrCodeLedgerInteractionLimitExceededError

Example:

[Error Code: 1106] max interaction with storage has exceeded the limit (used: 20276498 bytes, limit 20000000 bytes)

1107
ErrCodeStateKeySizeLimitError

Example:

...

```
1108
ErrCodeStateValueSizeLimitError
```

Example:

...

```
1109
ErrCodeTransactionFeeDeductionFailedError
```

Example:

```
[Error Code: 1109] failed to deduct 0 transaction fees from
14af75b8c487333c: Execution failed: f919ee77447b7497.FlowFees:97:24
```

```
1110
ErrCodeComputationLimitExceededError
```

Example:

```
[Error Code: 1110] computation exceeds limit (100)
```

```
1111
ErrCodeMemoryLimitExceededError
```

Example:

...

```
1112
ErrCodeCouldNotDecodeExecutionParameterFromState
```

Example:

...

```
1113
ErrCodeScriptExecutionTimedOutError
```

Example:

...

```
1114
ErrCodeScriptExecutionCancelledError
```

Example:

...

```
1115
ErrCodeEventEncodingError
```

Example:

...

```
1116
ErrCodeInvalidInternalStateAccessError
```

Example:

...

```
1118
ErrCodeInsufficientPayerBalance
```

Example:

```
[Error Code: 1118] payer ... has insufficient balance to attempt
transaction execution (required balance: 0.00100000)
```

```
1201
ErrCodeAccountNotFoundError
```

Example:

```
[Error Code: 1201] account not found for address xxx
```

```
1202
ErrCodeAccountPublicKeyNotFoundError
```

Example:

[Error Code: 1202] account public key not found for address xxx and key index 3

1203
ErrCodeAccountAlreadyExistsError

Example:

...

1204
ErrCodeFrozenAccountError

Example:

...

1206
ErrCodeAccountPublicKeyLimitError

Example:

...

1251
ErrCodeContractNotFoundError

Example:

...

2000
FailureCodeUnknownFailure

Example:

...

2001
FailureCodeEncodingFailure

Example:

...

2002
FailureCodeLedgerFailure

Example:

...

2003
FailureCodeStateMergeFailure

Example:

...

2004
FailureCodeBlockFinderFailure

Example:

...

2006
FailureCodeParseRestrictedModeInvalidAccessFailure

Example:

...

2007
FailureCodePayerBalanceCheckFailure

Example:

...

```
# index.md:

Clients

Go SDK

Flow Go SDK provides a set of packages for Go developers to build
applications that interact with the Flow network.

Python SDK

Flow Python SDK provides a set of packages for Python developers to build
applications that interact with the Flow network.

Ruby

FlowClient is a Ruby gRPC client for Flow (onflow.org).

JVM

Flow JVM SDK is a library for JVM-compatible languages, implemented in
Kotlin, that provides utilities to interact with the Flow blockchain.

JavaScript (FCL)

Flow Client Library (FCL) is a package used to interact with user wallets
and the Flow blockchain.

Swift

flow-swift is a tool to build iOS applications in the Flow mobile realm.

.NET

flow.net is a tool for building .NET applications on Flow.

Rust

Rust SDK for the Flow blockchain network.

PHP

PHP SDK for the Flow blockchain.

Elixir

OnFlow is an Elixir client for interacting with the Flow blockchain.
Documentation is available here.

HTTP API
```

Flow OpenAPI specification.

```
# api.md:
```

```
---
```

```
sidebarlabel: FCL Reference
```

```
sidebarposition: 2
```

```
---
```

Flow Client Library (FCL) API Reference

> For release updates, see the repo

Configuration

FCL has a mechanism that lets you configure various aspects of FCL. When you move from one instance of the Flow Blockchain to another (Local Emulator to Testnet to Mainnet) the only thing you should need to change for your FCL implementation is your configuration.

```
---
```

Setting Configuration Values

Values only need to be set once. We recommend doing this once and as early in the life cycle as possible. To set a configuration value, the put method on the config instance needs to be called, the put method returns the config instance so they can be chained.

Alternatively, you can set the config by passing a JSON object directly.

```
javascript
import as fcl from '@onflow/fcl';

fcl
  .config() // returns the config instance
  .put('foo', 'bar') // configures "foo" to be "bar"
  .put('baz', 'buz'); // configures "baz" to be "buz"

// OR

fcl.config({
  foo: 'bar',
  baz: 'buz',
});
```

Getting Configuration Values

The config instance has an asynchronous get method. You can also pass it a fallback value.

```
javascript
```

```

import  as fcl from '@onflow/fcl';

fcl.config().put('foo', 'bar').put('woot', 5).put('rawr', 7);

const FALLBACK = 1;

async function addStuff() {
    var woot = await fcl.config().get('woot', FALLBACK); // will be 5 --
set in the config before
    var rawr = await fcl.config().get('rawr', FALLBACK); // will be 7 --
set in the config before
    var hmmmm = await fcl.config().get('hmmm', FALLBACK); // will be 1 --
uses fallback because this isnt in the config

    return woot + rawr + hmmmm;
}

addStuff().then((d) => console.log(d)); // 13 (5 + 7 + 1)

```

Common Configuration Keys

Name	Example
Description	
accessNode.api (required)	https://rest-testnet.onflow.org
API URL for the Flow Blockchain Access Node you want to be communicating with. See all available access node endpoints here.	
app.detail.title	Cryptokitties
Your applications title, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.	
app.detail.icon	https://fcl-discovery.onflow.org/images/blocto.png
Url for your applications icon, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.	
app.detail.description	Cryptokitties is a blockchain game
Your applications description, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.	
app.detail.url	https://cryptokitties.co
Your applications url, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.	
challenge.handshake	DEPRECATED
Use discovery.wallet instead.	
discovery.authn.endpoint	https://fcl-discovery.onflow.org/api/testnet/authn
	Endpoint for

```

alternative configurable Wallet Discovery mechanism. Read more on
discovery
|
| discovery.wallet (required) | https://fcl-
discovery.onflow.org/testnet/authn | Points FCL at the
Wallet or Wallet Discovery mechanism.
|
| discovery.wallet.method | IFRAME/RPC, POP/RPC, TAB/RPC,
HTTP/POST, or EXT/RPC | Describes which service strategy a wallet should
use.
|
| fcl.limit | 100
| Specifies fallback compute limit if not provided in transaction.
Provided as integer.
|
| flow.network (recommended) | testnet
| Used in conjunction with stored interactions and provides
FCLCryptoContract address for testnet and mainnet. Possible values:
local, testnet, mainnet.
|
| walletconnect.projectId | YOURPROJECTID
| Your app's WalletConnect project ID. See WalletConnect Cloud to obtain
a project ID for your application.
|

```

Using Contracts in Scripts and Transactions

Address Replacement

Configuration keys that start with 0x will be replaced in FCL scripts and transactions, this allows you to write your script or transaction Cadence code once and not have to change it when you point your application at a difference instance of the Flow Blockchain.

```

javascript
import as fcl from '@onflow/fcl';

fcl.config().put('0xFungibleToken', '0xf233dcee88fe0abe');

async function myScript() {
    return fcl
        .send([
            fcl.script
                import FungibleToken from 0xFungibleToken // will be replaced with
0xf233dcee88fe0abe because of the configuration

                access(all) fun main() { / Rest of the script goes here / }
        ])
        .then(fcl.decode);
}

async function myTransaction() {
    return fcl
        .send([
            fcl.transaction

```

```

        import FungibleToken from 0xFungibleToken // will be replaced with
0xf233dcee88fe0abe because of the configuration

    transaction { / Rest of the transaction goes here / }

    ,
])
.then(fcl.decode);
}

```

Example

```

javascript
import  as fcl from '@onflow/fcl';

fcl
.config()
.put('flow.network', 'testnet')
.put('walletconnect.projectId', 'YOURPROJECTID')
.put('accessNode.api', 'https://rest-testnet.onflow.org')
.put('discovery.wallet', 'https://fcl-
discovery.onflow.org/testnet/authn')
.put('app.detail.title', 'Test Harness')
.put('app.detail.icon', 'https://i.imgur.com/r23Zhvu.png')
.put('app.detail.description', 'A test harness for FCL')
.put('app.detail.url', 'https://myapp.com')
.put('service.OpenID.scopes', 'email emailverified name zoneinfo')
.put('0xFlowToken', '0x7e60df042a9c0868');

```

Using Flow.json

A simpler way to import contracts in scripts and transactions is to use the config.load method to ingest your contracts from your flow.json file. This keeps the import syntax unified across tools and lets FCL figure out which address to use for what network based on the network provided in config. To use config.load you must first import your flow.json file and then pass it to config.load as a parameter.

```

javascript
import { config } from '@onflow/fcl';
import flowJSON from '../flow.json';

config({
  'flow.network': 'testnet',
  'accessNode.api': 'https://rest-testnet.onflow.org',
  'discovery.wallet': https://fcl-discovery.onflow.org/testnet/authn,
}).load({ flowJSON });

```

Let's say your flow.json file looks like this:

```
{

```

```
"contracts": {
    "HelloWorld": "cadence/contracts/HelloWorld.cdc"
}
}
```

Then in your scripts and transactions, all you have to do is:

```
import "HelloWorld"
```

FCL will automatically replace the contract name with the address for the network you are using.

> Note: never put private keys in your flow.json. You should use the key/location syntax to separate your keys into a separate git ignored file.

Wallet Interactions

These methods allows dapps to interact with FCL compatible wallets in order to authenticate the user and authorize transactions on their behalf.

> **⚠**These methods are async.

authenticate

> **⚠**This method can only be used in web browsers.

Calling this method will authenticate the current user via any wallet that supports FCL. Once called, FCL will initiate communication with the configured discovery.wallet endpoint which lets the user select a wallet to authenticate with. Once the wallet provider has authenticated the user, FCL will set the values on the current user object for future use and authorization.

Note

⚠discovery.wallet value must be set in the configuration before calling this method. See FCL Configuration.

💡 The default discovery endpoint will open an iframe overlay to let the user choose a supported wallet.

Usage

```
javascript
```

```
import  as fcl from '@onflow/fcl';
fcl
  .config()
    .put('accessNode.api', 'https://rest-testnet.onflow.org')
    .put('discovery.wallet', 'https://fcl-
discovery.onflow.org/testnet/authn');
// anywhere on the page
fcl.authenticate();
```

Note

⚠ authenticate can also take a service returned from discovery with
fcl.authenticate({ service }).

unauthenticate

> ⚠This method can only be used in web browsers.

Logs out the current user and sets the values on the current user object to null.

Note

⚠The current user must be authenticated first.

Usage

```
javascript
import  as fcl from '@onflow/fcl';
fcl.config().put('accessNode.api', 'https://rest-testnet.onflow.org');
// first authenticate to set current user
fcl.authenticate();
// ... somewhere else & sometime later
fcl.unauthenticate();
// fcl.currentUser.loggedIn === null
```

reauthenticate

> ⚠This method can only be used in web browsers.

A convenience method that calls fcl.unauthenticate() and then
fcl.authenticate() for the current user.

Note

⚠The current user must be authenticated first.

Usage

```
javascript
import  as fcl from '@onflow/fcl';
// first authenticate to set current user
fcl.authenticate();
// ... somewhere else & sometime later
fcl.reauthenticate();
// logs out user and opens up login/sign-up flow
```

signUp

> **⚠**This method can only be used in web browsers.

A convenience method that calls and is equivalent to `fcl.authenticate()`.

logIn

> **⚠**This method can only be used in web browsers.

A convenience method that calls and is equivalent to `fcl.authenticate()`.

authz

A convenience method that produces the needed authorization details for the current user to submit transactions to Flow. It defines a signing function that connects to a user's wallet provider to produce signatures to submit transactions.

> **💡** You can replace this function with your own authorization function if needed.

Returns

Type	Description
-----	-----
-----	AuthorizationObject An object containing the necessary details from the current user to authorize a transaction in any role.

Usage

Note: The default values for proposer, payer, and authorizations are already fcl.authz so there is no need to include these parameters, it is shown only for example purposes. See more on signing roles.

```
javascript
import as fcl from '@onflow/fcl';
// login somewhere before
fcl.authenticate();
// once logged in authz will produce values
console.log(fcl.authz);
// prints {addr, signingFunction, keyId, sequenceNum} from the current
authenticated user.

const txId = await fcl.mutate({
  cadence:
    import Profile from 0xba1132bc08f82fe2

    transaction(name: String) {
      prepare(account: auth(BorrowValue) &Account) {
        account.storage.borrow<&{Profile.Owner}>(from:
Profile.privatePath)!.setName(name)
      }
    }
  ,
  args: (arg, t) => [arg('myName', t.String)],
  proposer: fcl.authz, // optional - default is fcl.authz
  payer: fcl.authz, // optional - default is fcl.authz
  authorizations: [fcl.authz], // optional - default is [fcl.authz]
});
```

Current User

Holds the current user, if set, and offers a set of functions to manage the authentication and authorization of the user.

> **⚠**The following methods can only be used in web browsers.

currentUser.subscribe

The callback passed to subscribe will be called when the user authenticates and un-authenticates, making it easy to update the UI accordingly.

Arguments

Name	Type	

```
| ----- | ----- | -----  
----- |  
| callback | function | The callback will be called with the current user  
as the first argument when the current user is set or removed. |
```

Usage

```
javascript  
import React, { useState, useEffect } from 'react';  
import as fcl from '@onflow/fcl';  
  
export function AuthCluster() {  
  const [user, setUser] = useState({ loggedIn: null });  
  useEffect(() => fcl.currentUser.subscribe(setUser), []); // sets the  
callback for FCL to use  
  
  if (user.loggedIn) {  
    return (  
      <div>  
        <span>{user?.addr ?? 'No Address'}</span>  
        <button onClick={fcl.unauthenticate}>Log Out</button> /* once  
logged out in setUser(user) will be called */  
      </div>  
    );  
  } else {  
    return (  
      <div>  
        <button onClick={fcl.logIn}>Log In</button>{' '}  
        /* once logged in setUser(user) will be called */  
        <button onClick={fcl.signUp}>Sign Up</button> /* once signed up,  
setUser(user) will be called */  
      </div>  
    );  
  }  
}
```

```
currentUser.snapshot
```

Returns the current user object. This is the same object that is set and available on `fcl.currentUser.subscribe(callback)`.

Usage

```
javascript  
// returns the current user object  
const user = fcl.currentUser.snapshot();  
  
// subscribes to the current user object and logs to console on changes  
fcl.currentUser.subscribe(console.log);
```

currentUser.authenticate

Equivalent to fcl.authenticate.

currentUser.unauthenticate

Equivalent to fcl.unauthenticate.

currentUser.authorization

Equivalent to fcl.authz

currentUser.signUserMessage

A method to use allowing the user to personally sign data via FCL Compatible Wallets/Services.

> **⚠** This method requires the current user's wallet to support a signing service endpoint. Currently, only Blocto is compatible with this feature by default.

Arguments

Name	Type	Description
message	string (required)	A hexadecimal string to be signed

Returns

Type	Description
Array	An Array of CompositeSignatures: {addr, keyId, signature}

Usage

```
javascript
import  as fcl from '@onflow/fcl';

export const signMessage = async () => {
  const MSG = Buffer.from('FOO').toString('hex');
  try {
    return await currentUser.signUserMessage(MSG);
  }
}
```

```
        } catch (error) {
          console.log(error);
        }
      };
    }

---
```

Discovery

discovery

Discovery abstracts away code so that developers don't have to deal with the discovery of Flow compatible wallets, integration, or authentication. Using discovery from FCL allows dapps to list and authenticate with wallets while having full control over the UI. Common use cases for this are login or registration pages.

(Alternatively, if you don't need control over your UI you can continue to use the `discovery.wallet` config value documented in the Quickstart for the simplest configuration.)

> **⚠**The following methods can only be used in web browsers.

Note

⚠`discovery.authn.endpoint` value must be set in the configuration before calling this method. See FCL Configuration.

Suggested Configuration

Environment	Example
Mainnet	<code>https://fcl-discovery.onflow.org/api/authn</code>
Testnet	<code>https://fcl-discovery.onflow.org/api/testnet/authn</code>

If the Discovery endpoint is set in config, then you can iterate through authn services and pass the chosen service to authenticate to authenticate a user.

Usage

```
javascript
import './config';
import { useState, useEffect } from 'react';
import { fcl } from '@onflow/fcl';

function Component() {
  const [wallets, setWallets] = useState([]);
  useEffect(
    () => fcl.discovery.authn.subscribe((res) =>
      setWallets(res.results)),
    []
  );
}
```

```

        return (
      <div>
        {wallets.map((wallet) => (
          <button
            key={wallet.provider.address}
            onClick={() => fcl.authenticate({ service: wallet })}
          >
            Login with {wallet.provider.name}
          </button>
        )));
      </div>
    );
}

```

authn

More Configuration

By default, limited functionality services or services that require developer registration, like Ledger or Dapper Wallet, require apps to opt-in in order to display to users. To enable opt-in services in an application, use the `discovery.authn.include` property in your configuration with a value of an array of services you'd like your app to opt-in to displaying for users.

```

javascript
import { config } from '@onflow/fcl';

config({
  'discovery.authn.endpoint':
    'https://fcl-discovery.onflow.org/api/testnet/authn', // Endpoint set
    to Testnet
  'discovery.authn.include': ['0x9d2e44203cb13051'], // Ledger wallet
    address on Testnet set to be included
});

```

Opt-In Wallet Addresses on Testnet and Mainnet

Service	Testnet	Mainnet
Dapper Wallet	0x82ec283f88a62e65	0xead892083b3e2c6c
Ledger	0x9d2e44203cb13051	0xe5cd26afebe62781

For more details on wallets, view the service list here.

`discovery.authn.snapshot()`

Return a list of authn services.

```
discovery.authn.subscribe(callback)
```

The callback sent to subscribe will be called with a list of authn services.

On-chain Interactions

> **💡** These methods can be used in browsers and NodeJS.

These methods allows dapps to interact directly with the Flow blockchain via a set of functions that currently use the Access Node API.

Query and Mutate Flow with Cadence

If you want to run arbitrary Cadence scripts on the blockchain, these methods offer a convenient way to do so without having to build, send, and decode interactions.

query

Allows you to submit scripts to query the blockchain.

Options

Pass in the following as a single object with the following keys. All keys are optional unless otherwise stated.

Key	Type	Description
----- ----- -----		

cadence string (required)		A valid cadence script.
args ArgumentFunction Any arguments to the script if needed should be supplied via a function that returns an array of arguments.		
limit number Compute (Gas) limit for query. Read the documentation about computation cost for information about how computation cost is calculated on Flow.		

Returns

Type	Description	
any A JSON representation of the response.		

Usage

```

javascript
import  as fcl from '@onflow/fcl';

const result = await fcl.query({
  cadence:
    access(all) fun main(a: Int, b: Int, addr: Address): Int {
      log(addr)
      return a + b
    }
  ,
  args: (arg, t) => [
    arg(7, t.Int), // a: Int
    arg(6, t.Int), // b: Int
    arg('0xba1132bc08f82fe2', t.Address), // addr: Address
  ],
});
console.log(result); // 13

```

Examples

- Additional Explanation

mutate

Allows you to submit transactions to the blockchain to potentially mutate the state.

⚠When being used in the browser, `fcl.mutate` uses the built-in `fcl.authz` function to produce the authorization (signatures) for the current user. When calling this method from Node.js, you will need to supply your own custom authorization function.

Options

Pass in the following as a single object with the following keys. All keys are optional unless otherwise stated.

Key	Type	
Description		
----- ----- -----		

cadence string (required)		A valid
cadence transaction.		
args ArgumentFunction		Any arguments to the script if
needed should be supplied via a function that returns an array of		
arguments.		

```
| limit      | number                                | Compute  
| (Gas) limit for query. Read the documentation about computation cost for  
| information about how computation cost is calculated on Flow. |  
| proposer   | AuthorizationFunction | The authorization function that  
| returns a valid AuthorizationObject for the proposer role.  
|
```

Returns

Type	Description
string	The transaction ID.

Usage

```
javascript
import { fcl } from '@onflow/fcl';
// login somewhere before
fcl.authenticate();

const txId = await fcl.mutate({
  cadence:
    import Profile from 0xb1132bc08f82fe2

    transaction(name: String) {
      prepare(account: auth(BorrowValue) & Account) {
        account.storage.borrow<&{Profile.Owner}>(from:
          Profile.privatePath)!.setName(name)
      }
    }
  ,
  args: (arg, t) => [arg('myName', t.String)],
});
```

Examples

- Additional explanation
- Custom authorization function

`verifyUserSignatures` (Deprecated)

Use `fcl.AppUtils.verifyUserSignatures`

AppUtils

`AppUtils.verifyUserSignatures`

A method allowing applications to cryptographically verify a message was signed by a user's private key/s. This is typically used with the response from `currentUser.signUserMessage`.

Note

⚠ fcl.config.flow.network or options override is required to use this api. See FCL Configuration.

Arguments

Name	Type	Description
message	string (required)	A hexadecimal string
compositeSignatures	Array (required)	An Array of CompositeSignatures
opts	Object (optional)	opts.fclCryptoContract can be provided to override FCLCryptoContract address for local development

Returns

Type	Description
Boolean	true if verified or false

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const isValid = await fcl.AppUtils.verifyUserSignatures(
  Buffer.from('FOO').toString('hex'),
  [
    {
      ftype: 'CompositeSignature',
      fvsn: '1.0.0',
      addr: '0x123',
      keyId: 0,
      signature: 'abc123',
    },
  ],
  { fclCryptoContract },
);
```

Examples

- fcl-next-harness

AppUtils.verifyAccountProof

A method allowing applications to cryptographically prove that a user controls an on-chain account. During user authentication, some FCL compatible wallets will choose to support the FCL account-proof service. If a wallet chooses to support this service, and the user approves the signing of message data, they will return account-proof data and a signature(s) that can be used to prove a user controls an on-chain account.

See [proving-authentication documentaion](#) for more details.

⚠ `fcl.config.flow.network` or options override is required to use this api. See [FCL Configuration](#).

Arguments

Name	Type	Description
appIdentifier	string (required)	A hexadecimal string
accountProofData	Object (required)	Object with properties: address: string - A Flow account address. nonce: string - A random string in hexadecimal format (minimum 32 bytes in total, i.e 64 hex characters) signatures: Object[] - An array of composite signatures to verify
opts	Object (optional)	opts.fclCryptoContract can be provided to override FCLCryptoContract address for local development

Returns

Type	Description
Boolean	true if verified or false

Usage

```
javascript
import  as fcl from "@onflow/fcl"

const accountProofData = {
  address: "0x123",
  nonce: "F0123"
  signatures: [{ftype: "CompositeSignature", fvsn: "1.0.0", addr:
  "0x123", keyId: 0, signature: "abc123"}],
}

const isValid = await fcl.AppUtils.verifyAccountProof(
  "AwesomeAppId",
  accountProofData,
  {fclCryptoContract}
```

)

Examples

- `fcl.next.harness`

Query and mutate the blockchain with Builders

In some cases, you may want to utilize pre-built interactions or build more complex interactions than what the `fcl.query` and `fcl.mutate` interface offer. To do this, FCL uses a pattern of building up an interaction with a combination of builders, resolving them, and sending them to the chain.

> **⚠Recommendation:** Unless you have a specific use case that require usage of these builders, you should be able to achieve most cases with `fcl.query({...options})` or `fcl.mutate({...options})`

send

Sends arbitrary scripts, transactions, and requests to Flow.

This method consumes an array of builders that are to be resolved and sent. The builders required to be included in the array depend on the interaction that is being built.

Note

⚠Must be used in conjunction with `fcl.decode(response)` to get back correct keys and all values in JSON.

Arguments

Name	Type	Description
<code>builders</code>	[Builders]	See builder functions.

Returns

Type	Description
<code>responseObject</code>	An object containing the data returned from the chain. Should always be decoded with <code>fcl.decode()</code> to get back appropriate JSON keys and values.

Usage

```

javascript
import  as fcl from '@onflow/fcl';

// a script only needs to resolve the arguments to the script
const response = await fcl.send([fcl.script${script}, fcl.args(args)]);
// note: response values are encoded, call await fcl.decode(response) to
get JSON

// a transaction requires multiple 'builders' that need to be resolved
prior to being sent to the chain - such as setting the authorizations.
const response = await fcl.send([
  fcl.transaction
    ${transaction}

  ,
  fcl.args(args),
  fcl.proposer(proposer),
  fcl.authorizations(authorizations),
  fcl.payer(payer),
  fcl.limit(9999),
]);
// note: response contains several values (Cad)

---

```

decode

Decodes the response from `fcl.send()` into the appropriate JSON representation of any values returned from Cadence code.

Note

 To define your own decoder, see tutorial.

Arguments

Name	Type	Description
response ResponseObject Should be the response returned from fcl.send([...])		

Returns

Type	Description
any A JSON representation of the raw string response depending on the cadence code executed. The return value can be a single value and type or an object with multiple types.	

Usage

```
javascript
import  as fcl from '@onflow/fcl';

// simple script to add 2 numbers
const response = await fcl.send([
  fcl.script
    access(all) fun main(int1: Int, int2: Int): Int {
      return int1 + int2
    }
  ,
  fcl.args([fcl.arg(1, fcl.t.Int), fcl.arg(2, fcl.t.Int)]),
]);
const decoded = await fcl.decode(response);

assert(3 === decoded);
assert(typeof decoded === 'number');
```

Builders

These methods fill out various portions of a transaction or script template in order to build, resolve, and send it to the blockchain. A valid populated template is referred to as an Interaction.

⚠These methods must be used with
fcl.send([...builders]).then(fcl.decode)

Query Builders

getAccount

A builder function that returns the interaction to get an account by address.

⚠Consider using the pre-built interaction fcl.account(address) if you do not need to pair with any other builders.

Arguments

Name	Type	Description
address	Address	Address of the user account with or without a prefix (both formats are supported).

Returns after decoding

Type	Description
AccountObject	A JSON representation of a user account.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

// somewhere in an async function
// fcl.account is the same as this function
const getAccount = async (address) => {
  const account = await
fcl.send([fcl.getAccount(address)]).then(fcl.decode);
  return account;
};
```

getBlock

A builder function that returns the interaction to get the latest block.

⚠️ Use with fcl.atBlockId() and fcl.atBlockHeight() when building the interaction to get information for older blocks.

⚠️ Consider using the pre-built interaction fcl.getblock(isSealed) if you do not need to pair with any other builders.

Arguments

Name	Type	Default	Description
isSealed	boolean	false	If the latest block should be sealed or not. See block states.

Returns after decoding

Type	Description
BlockObject	The latest block if not used with any other builders.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const latestSealedBlock = await fcl
  .send([
    ])
```

```
fcl.getBlock(true), // isSealed = true
])
.then(fcl.decode);
```

atBlockHeight

A builder function that returns a partial interaction to a block at a specific height.

⚠️Use with other interactions like fcl.getBlock() to get a full interaction at the specified block height.

Arguments

Name	Type	Description
-----	-----	-----
blockHeight number The height of the block to execute the interaction at.		

Returns

Type	Description
-----	-----
-	-
Partial Interaction A partial interaction to be paired with another interaction such as fcl.getBlock() or fcl.getAccount().	

Usage

```
javascript
import  as fcl from '@onflow/fcl';

await fcl.send([fcl.getBlock(),
fcl.atBlockHeight(123)]).then(fcl.decode);
```

atBlockId

A builder function that returns a partial interaction to a block at a specific block ID.

⚠️Use with other interactions like fcl.getBlock() to get a full interaction at the specified block ID.

Arguments

Name	Type	Description
blockId	string	The ID of the block to execute the interaction at.

Returns

Type	Description
Partial Interaction	A partial interaction to be paired with another interaction such as fcl.getBlock() or fcl.getAccount().

Usage

```
javascript
import  as fcl from '@onflow/fcl';

await fcl.send([fcl.getBlock(),
fcl.atBlockId('23232323232')]).then(fcl.decode);
```

getBlockHeader

A builder function that returns the interaction to get a block header.

 Use with fcl.atBlockId() and fcl.atBlockHeight() when building the interaction to get information for older blocks.

Returns after decoding

Type	Description
BlockHeaderObject	The latest block header if not used with any other builders.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const latestBlockHeader = await fcl
.send([fcl.getBlockHeader()])
.then(fcl.decode);
```

getEventsAtBlockHeightRange

A builder function that returns all instances of a particular event (by name) within a height range.

⚠️The block range provided must be from the current spork.

⚠️The block range provided must be 250 blocks or lower per request.

Arguments

Name	Type	Description
eventName	EventName	The name of the event.
fromBlockHeight	number	The height of the block to start looking for events (inclusive).
toBlockHeight	number	The height of the block to stop looking for events (inclusive).

Returns after decoding

Type	Description
[EventObject] (#event-object)	An array of events that matched the eventName.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const events = await fcl
  .send([
    fcl.getEventsAtBlockHeightRange(
      'A.7e60df042a9c0868.FlowToken.TokensWithdrawn',
      35580624,
      35580624,
    ),
  ])
  .then(fcl.decode);
```

getEventsAtBlockIds

A builder function that returns all instances of a particular event (by name) within a set of blocks, specified by block ids.

⚠The block range provided must be from the current spork.

Arguments

Name	Type	Description
eventName	EventName	The name of the event.
blockIds	number	The ids of the blocks to scan for events.

Returns after decoding

Type	Description
[EventObject]	An array of events that matched the eventName.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const events = await fcl
  .send([
    fcl.getEventsAtBlockIds('A.7e60df042a9c0868.FlowToken.TokensWithdrawn', [
      'c4f239d49e96d1e5fbef1f31027a6e582e8c03fc9954177b7723fdb03d938c7',
      '5dbaa85922eb194a3dc463c946cc01c866f2ff2b88f3e59e21c0d8d00113273f',
    ]),
  ])
  .then(fcl.decode);
```

getCollection

A builder function that returns all a collection containing a list of transaction ids by its collection id.

⚠The block range provided must be from the current spork. All events emitted during past sporks is current unavailable.

Arguments

Name	Type	Description
collectionID	string	The id of the collection.

Returns after decoding

Type	Description
CollectionObject	An object with the id and a list of transactions within the requested collection.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const collection = await fcl
  .send([
    fcl.getCollection(
      'cccdbe0c67d015dc7f6444e8f62a3244ed650215ed66b90603006c70c5ef1f6e5',
    ),
  ])
  .then(fcl.decode);
```

getTransactionStatus

A builder function that returns the status of transaction in the form of a TransactionStatusObject.

⚠️The transactionID provided must be from the current spork.

⚠️ Considering subscribing to the transaction from fcl.tx(id) instead of calling this method directly.

Arguments

Name	Type	Description
transactionId	string	The transactionID returned when submitting a transaction. Example: 9dda5f281897389b99f103a1c6b180eec9dac870de846449a302103ce38453f3

Returns after decoding

Returns

Type	Description

TransactionStatusObject Object representing the result/status of a transaction

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const status = await fcl
  .send([
    fcl.getTransactionStatus(
      '9dda5f281897389b99f103a1c6b180eec9dac870de846449a302103ce38453f3',
    ),
  ])
  .then(fcl.decode);
```

getTransaction

A builder function that returns a transaction object once decoded.

⚠️ The transactionID provided must be from the current spork.

👉 Considering using `fcl.tx(id).onceSealed()` instead of calling this method directly.

Arguments

Name	Type	Description
transactionID	string	The transactionID returned when submitting a transaction. Example: 9dda5f281897389b99f103a1c6b180eec9dac870de846449a302103ce38453f3

Returns after decoding

Returns

Type	Description
TransactionObject	An full transaction object containing a payload and signatures

Usage

```
javascript
import  as fcl from '@onflow/fcl';
```

```
const tx = await fcl
  .send([
    fcl.getTransaction(
      '9ddda5f281897389b99f103a1c6b180eec9dac870de846449a302103ce38453f3',
    ),
  ])
  .then(fcl.decode);
```

subscribeEvents

```
<Callout type="info">
The subscribeEvents SDK builder is for more advanced use cases where you
wish to directly specify a starting block to listen for events. For most
use cases, consider using the pre-built interaction
fcl.events(eventTypes).
</Callout>
```

A build that returns a event stream connection once decoded. It will establish a WebSocket connection to the Access Node and subscribe to events with the given parameters.

Arguments

Name	Type	Description
opts	Object	An object with the following keys:
opts.startBlockId	string | undefined	The block ID to start listening for events. Example: 9ddda5f281897389b99f103a1c6b180eec9dac870de846449a302103ce38453f3
opts.startHeight	number | undefined	The block height to start listening for events. Example: 123
opts.eventTypes	string[] | undefined	The event types to listen for. Example: A.7e60df042a9c0868.FlowToken.TokensWithdrawn
opts.addresses	string[] | undefined	The addresses to listen for. Example: 0x7e60df042a9c0868
opts.contracts	string[] | undefined	The contracts to listen for. Example: 0x7e60df042a9c0868
opts.heartbeatInterval	number | undefined	The interval in milliseconds to send a heartbeat to the Access Node. Example: 10000

Returns after decoding

Returns

Type	Description
EventStreamConnection	A connection to the event stream

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const eventStream = await fcl
  .send([
    fcl.subscribeEvents({
      eventTypes: 'A.7e60df042a9c0868.FlowToken.TokensWithdrawn',
    }),
  ])
  .then(fcl.decode);

eventStream.on('heartbeat', (heartbeat) => {
  console.log(heartbeat);
});

eventStream.on('events', (event) => {
  console.log(event);
});

eventStream.on('error', (error) => {
  console.log(error);
});

eventStream.on('end', () => {
  console.log('Connection closed');
});

eventStream.close();

---
```

getEvents (Deprecated)

Use fcl.getEventsAtBlockHeightRange or fcl.getEventsAtBlockIds.

getLatestBlock (Deprecated)

Use fcl.getBlock.

getBlockById (Deprecated)

Use fcl.getBlock and fcl.atBlockId.

getBlockByHeight (Deprecated)

Use fcl.getBlock and fcl.atBlockHeight.

Utility Builders

These builders are used to compose interactions with other builders such as scripts and transactions.

> **⚠Recommendation:** Unless you have a specific use case that require usage of these builders, you should be able to achieve most cases with fcl.query({...options}) or fcl.mutate({...options})

arg

A utility builder to be used with fcl.args[...] to create FCL supported arguments for interactions.

Arguments

Name	Type	Description
----- ----- -----		

value any		Any value that you are looking to pass to other builders.
type FType		A type supported by Flow.

Returns

Type	Description
----- ----- -----	
ArgumentObject	Holds the value and type passed in.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

await fcl
  .send([
    ...]
```

```

fcl.script
  access(all) fun main(a: Int, b: Int): Int {
    return a + b
  }
,
fcl.args([
  fcl.arg(5, fcl.t.Int), // a
  fcl.arg(4, fcl.t.Int), // b
]),
])
.then(fcl.decode);

```

args

A utility builder to be used with other builders to pass in arguments with a value and supported type.

Arguments

Name	Type	Description
args [[Argument Objects]](#argumentobject)	An array of arguments that you are looking to pass to other builders.	

Returns

Type	Description
Partial Interaction An interaction that contains the arguments and types passed in. This alone is a partial and incomplete interaction.	

Usage

```

javascript
import  as fcl from '@onflow/fcl';

await fcl
  .send([
    fcl.script
      access(all) fun main(a: Int, b: Int): Int {
        return a + b
      }
,
    fcl.args([
      fcl.arg(5, fcl.t.Int), // a
      fcl.arg(4, fcl.t.Int), // b
    ])
  ])
  .then(fcl.decode);

```

```
    ],
])
.then(fcl.decode); // 9
```

Template Builders

> **⚠Recommended:** The following functionality is simplified by `fcl.query({...options})` or `fcl.mutate({...options})` and is recommended to use over the functions below.

script

A template builder to use a Cadence script for an interaction.

👉 Use with `fcl.args(...)` to pass in arguments dynamically.

Arguments

Name	Type	Description
CODE	string	Should be valid Cadence script.

Returns

Type	Description
Interaction	An interaction containing the code passed in.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const code =
  access(all) fun main(): Int {
    return 5 + 4
  }
;
const answer = await fcl.send([fcl.script(code)]).then(fcl.decode);
console.log(answer); // 9
```

transaction

A template builder to use a Cadence transaction for an interaction.

⚠ Must be used with `fcl.payer`, `fcl.proposer`, `fcl.authorizations` to produce a valid interaction before sending to the chain.

📣 Use with `fcl.args[...]` to pass in arguments dynamically.

Arguments

Name	Type	Description
<code>CODE</code>	string	Should be valid a Cadence transaction.

Returns

Type	Description
<code>Partial Interaction</code>	An partial interaction containing the code passed in. Further builders are required to complete the interaction - see warning.

Usage

```
javascript
import  as fcl from '@onflow/fcl';

const code =
  access(all) fun main(): Int {
    return 5 + 4
  }
;
const answer = await fcl.send([fcl.script(code)]).then(fcl.decode);
console.log(answer); // 9
```

Pre-built Interactions

These functions are abstracted short hand ways to skip the send and decode steps of sending an interaction to the chain. More pre-built interactions are coming soon.

account

A pre-built interaction that returns the details of an account from their public address.

Arguments

Name	Type	Description

----- ----- -----
----- ----- -----
address Address Address of the user account with or without a prefix (both formats are supported).

Returns

Type	Description
AccountObject A JSON representation of a user account.	

Usage

```
javascript
import as fcl from '@onflow/fcl';
const account = await fcl.account('0x1d007d755706c469');
```

block

A pre-built interaction that returns the latest block (optionally sealed or not), by id, or by height.

Arguments

Name	Type	Default	Description
sealed boolean false If the latest block should be sealed or not. See block states.			
id string ID of block to get.			
height int Height of block to get.			

Returns

Type	Description
BlockObject A JSON representation of a block.	

Usage

```
javascript
import as fcl from '@onflow/fcl';
await fcl.block(); // get latest finalized block
await fcl.block({ sealed: true }); // get latest sealed block
await fcl.block({
  id: '0b1bdafa9ddaaaf31d53c584f208313557d622d1fedee1586ffc38fb5400979faa',
```

```
}); // get block by id
await fcl.block({ height: 56481953 }); // get block by height
```

latestBlock (Deprecated)

A pre-built interaction that returns the latest block (optionally sealed or not).

Arguments

Name	Type	Default	Description
isSealed	boolean	false	If the latest block should be sealed or not. See block states.

Returns

Type	Description
BlockObject	A JSON representation of a block.

Usage

```
javascript
import as fcl from '@onflow/fcl';
const latestBlock = await fcl.latestBlock();
```

Transaction Status Utility

tx

A utility function that lets you set the transaction to get subsequent status updates (via polling) and the finalized result once available.

⚠The poll rate is set at 2500ms and will update at that interval until transaction is sealed.

Arguments

Name	Type	Description
transactionId	string	A valid transaction id.

Returns

Name	Type	Description

-----	-----	-----
snapshot()	function	Returns the current state of the transaction.
subscribe(cb)	function	Calls the cb passed in with the new transaction on a status change.
onceFinalized()	function	Provides the transaction once status 2 is returned. See Transaction Statuses.
onceExecuted()	function	Provides the transaction once status 3 is returned. See Transaction Statuses.
onceSealed()	function	Provides the transaction once status 4 is returned. See Transaction Statuses.

Usage

```
javascript
import { FCL } from '@onflow/fcl';

const [txStatus, setTxStatus] = useState(null);
useEffect(() => FCL.tx(txId).subscribe(setTxStatus));
```

Event Polling Utility

events

A utility function that lets you set the transaction to get subsequent status updates (via polling) and the finalized result once available.

⚠️The poll rate is set at 10000ms and will update at that interval for getting new events.

Note:

⚠️`FCL.eventPollRate` value could be set to change the polling rate of all events subscribers, check FCL Configuration for guide.

Arguments

Name	Type	Description
eventname	string	A valid event name.

Returns

Name	Type	Description
--		
subscribe(cb)	function	Calls the cb passed in with the new event.

Usage

```
javascript
import as fcl from '@onflow/fcl';
// in some react component
fcl.events(eventName).subscribe((event) => {
  console.log(event);
});
```

Examples

- Flow-view-source example

Types, Interfaces, and Definitions

Builders

Builders are modular functions that can be coupled together with `fcl.send([...builders])` to create an Interaction. The builders needed to create an interaction depend on the script or transaction that is being sent.

Interaction

An interaction is an object containing the information to perform an action on chain. This object is populated through builders and converted into the appropriate access node API call. See the interaction object here. A 'partial' interaction is an interaction object that does not have sufficient information to the intended on-chain action. Multiple partial interactions (through builders) can be coupled to create a complete interaction.

CurrentUserObject

Key	Type	Default	Description
addr	Address	null	The public address of the current user
cid	string	null	Allows wallets to specify a content identifier for user metadata.

expiresAt number	null	Allows wallets to specify a time-frame for a valid session.
ftype string	'USER'	A type identifier used internally by FCL.
fvsn string	'1.0.0'	FCL protocol version.
loggedIn boolean	null	If the user is logged in.
services [ServiceObject]	[]	A list of trusted services that express ways of interacting with the current user's identity, including means to further discovery, authentication, authorization, or other kinds of interactions.

AuthorizationObject

This type conforms to the interface required for FCL to authorize transaction on behalf of the current user.

Key	Value Type	Description
addr	Address	The address of the authorizer
signingFunction	function	A function that allows FCL to sign using the authorization details and produce a valid signature.
keyId	number	The index of the key to use during authorization. (Multiple keys on an account is possible).
sequenceNum	number	A number that is incremented per transaction using they keyId.

SignableObject

An object that contains all the information needed for FCL to sign a message with the user's signature.

Key	Value Type	Description
addr	Address	The address of the authorizer
keyId	number	The index of the key to use during authorization. (Multiple keys on an account is possible).
signature	function	A SigningFunction that can produce a valid signature for a user from a message.

AccountObject

The JSON representation of an account on the Flow blockchain.

Key	Value Type	Description
address	Address	The address of the account
balance	number	The FLOW balance of the account in 10^8 .
code	Code	The code of any Cadence contracts stored in the account.
contracts	Object: Contract	An object with keys as the contract name deployed and the value as the the cadence string.
keys	[[KeyObject]](#keyobject)	Any contracts deployed to this account.

Address

Value Type	Description
string(formatted)	A valid Flow address should be 16 characters in length. A 0x prefix is optional during inputs. eg. f8d6e0586b0a20c1

ArgumentObject

An argument object created by `fcl.arg(value,type)`

Key	Value Type	Description
value	any	Any value to be used as an argument to a builder.
xform	FType	Any of the supported types on Flow.

ArgumentFunction

An function that takes the `fcl.arg` function and `fcl` types `t` and returns an array of `fcl.arg(value,type)`.

```
(arg, t) => Array<Arg>

| Parameter Name | Value Type           | Description
|
| ----- | ----- | -----
| arg      | function        | A function that returns an
ArgumentObject - fcl.arg. |
| t        | FTypes          | An object with access to all of the supported
types on Flow. |
```

Returns

Value Type	Description
[fcl.args]	Array of fcl.args.

Authorization Function

An authorization function must produce the information of the user that is going to sign and a signing function to use the information to produce a signature.

⚠This function is always async.

⚠ By default FCL exposes fcl.authz that produces the authorization object for the current user (given they are signed in and only on the browser). Replace this with your own function that conforms to this interface to use it wherever an authorization object is needed.

Parameter Name	Value Type	Description
account	AccountObject	The account of the user that is going to sign.

Returns

Value Type	Description
Promise<AuthorizationObject>	The object that contains all the information needed by FCL to authorize a user's transaction.

Usage

javascript

```

const authorizationFunction = async (account) => {
  // authorization function need to return an account
  const { address, keys } = account
  const tempId = `${address}-${keys[process.env.minterAccountIndex]}`;
  const keyId = Number(KEYID);
  let signingFunction = async signable => {
    return {
      keyId,
      addr: fcl.withPrefix(address),
      signature: sign(process.env.FLOWMINTERPRIVATEKEY,
signable.message), // signing function, read below
    }
  }
  return {
    ...account,
    address,
    keyId,
    tempId,
    signingFunction,
  }
}

```

- Detailed explanation

Signing Function

Consumes a payload and produces a signature for a transaction.

⚠This function is always `async`.

⚠ Only write your own signing function if you are writing your own custom authorization function.

Payload

Note: These values are destructed from the payload object in the first argument.

Parameter Name	Value Type	Description
message	string	The encoded string which needs to be used to produce the signature.
addr	string	The encoded string which needs to be used to produce the signature.
keyId	string	The encoded string which needs to be used to produce the signature.

roles	string	The encoded string which needs to be used to produce the signature.
voucher	object	The raw transactions information, can be used to create the message for additional safety and lack of trust in the supplied message.

Returns

Value Type	Description
Promise<SignableObject>	The object that contains all the information needed by FCL to authorize a user's transaction.

Usage

```
javascript
import as fcl from '@onflow/fcl';
import { ec as EC } from 'elliptic';
import { SHA3 } from 'sha3';
const ec: EC = new EC('p256');

const produceSignature = (privateKey, msg) => {
  const key = ec.keyFromPrivate(Buffer.from(privateKey, 'hex'));
  const sig = key.sign(this.hashMsg(msg));
  const n = 32;
  const r = sig.r.toArrayLike(Buffer, 'be', n);
  const s = sig.s.toArrayLike(Buffer, 'be', n);
  return Buffer.concat([r, s]).toString('hex');
};

const signingFunction = ({
  message, // The encoded string which needs to be used to produce the signature.
  addr, // The address of the Flow Account this signature is to be produced for.
  keyId, // The keyId of the key which is to be used to produce the signature.
  roles: {
    proposer, // A Boolean representing if this signature to be produced for a proposer.
    authorizer, // A Boolean representing if this signature to be produced for a authorizer.
    payer, // A Boolean representing if this signature to be produced for a payer.
  },
  voucher, // The raw transactions information, can be used to create the message for additional safety and lack of trust in the supplied message.
}) => {
  return {
    addr, // The address of the Flow Account this signature was produced for.
  }
};
```

```

    keyId, // The keyId for which key was used to produce the signature.
    signature: produceSignature(message), // The hex encoded string
    representing the signature of the message.
  };
};


```

Examples:

- Detailed explanation

TransactionObject

Key	Value Type	
Description		
args	object	A list of encoded Cadence values passed into this transaction. These have not been decoded by the JS-SDK.
authorizers	[\[Address\]](#address)	A list of the accounts that are authorizing this transaction to mutate to their on-chain account state. See more here .
envelopeSignatures	[\[SignableObject\]](#signableobject)	A list of signatures generated by the payer role. See more here .
gasLimit	number	The maximum number of computational units that can be used to execute this transaction. See more here .
payer	Address	The account that pays the fee for this transaction. See more here .
payloadSignatures	[\[SignableObject\]](#signableobject)	A list of signatures generated by the proposer and authorizer roles. See more here .
proposalKey	[\[ProposalKey\]](#proposalkeyobject)	The account key used to propose this transaction
referenceBlockId	string	A reference to the block used to calculate the expiry of this transaction.
script	string	The UTF-8 encoded Cadence source code that defines the execution logic for this transaction

TransactionRolesObject

Key Name	Value Type	Description
proposer	boolean	A Boolean representing if this signature to be produced for a proposer.
authorizer	boolean	A Boolean representing if this signature to be produced for an authorizer.
payer	boolean	A Boolean representing if this signature to be produced for a payer.

For more on what each transaction role means, see [signing roles](#).

TransactionStatusObject

Key	Value Type	Description
blockId	string	ID of the block that contains the transaction.
events	[[EventObject]](#event-object)	An array of events that were emitted during the transaction.
status	TransactionStatus	The status of the transaction on the blockchain.
statusString	TransactionStatus	The status as descriptive text (e.g. "FINALIZED").
errorMessage	string	An error message if it exists. Default is an empty string ''.
statusCode	number	The pass/fail status. 0 indicates the transaction succeeded, 1 indicates it failed.

EventName

Value Type	Description
string(formatted)	A event name in Flow must follow the format A.{AccountAddress}.{ContractName}.{EventName} eg. A.ba1132bc08f82fe2.Debug.Log

Contract

Value Type	Description

string(formatted)	A formatted string that is a valid cadence contract.
-------------------	------------------------------------------------------

KeyObject

This is the JSON representation of a key on the Flow blockchain.

Key	Value Type	Description
index	number	The address of the account
publicKey	string	The public portion of a public/private key pair
signAlgo	number	An index referring to one of ECDSAP256 or ECDSAsAcp256k1
hashAlgo	number	An index referring to one of SHA2256 or SHA3256
weight	number	A number between 1 and 1000 indicating the relative weight to other keys on the account.
sequenceNumber	number	This number is incremented for every transaction signed using this key.
revoked	boolean	If this key has been disabled for use.

ProposalKeyObject

ProposalKey is the account key used to propose this transaction.

A proposal key references a specific key on an account, along with an up-to-date sequence number for that key. This sequence number is used to prevent replay attacks.

You can find more information about sequence numbers [here](#)

Key	Value Type	Description
address	Address	The address of the account
keyIndex	number	The index of the account key being referenced
sequenceNumber	number	The sequence number associated with this account key for this transaction

BlockObject

The JSON representation of a key on the Flow blockchain.

Key	Value Type	
Description		

id	string	
The id of the block.		
parentId	string	
The id of the parent block.		
height	number	
The height of the block.		
timestamp	object	
Contains time related fields.		
collectionGuarantees	[CollectionGuaranteeObject]	Contains the ids of collections included in the block.
blockSeals	[SealedBlockObject]	
The details of which nodes executed and sealed the blocks.		
signatures	Uint8Array([numbers])	
All signatures.		

BlockHeaderObject

The subset of the BlockObject containing only the header values of a block.

Key	Value Type	Description
id	string	The id of the block.
parentId	string	The id of the parent block.
height	number	The height of the block.
timestamp	object	Contains time related fields.

CollectionGuaranteeObject

A collection that has been included in a block.

Key	Value Type	Description
collectionId	string	The id of the block.
signatures	SignatureObject	All signatures.

CollectionObject

A collection is a list of transactions that are contained in the same block.

Key	Value Type	Description
id	string	The id of the collection.
transactionIds	[string]	The ids of the transactions included in the collection.

responseObject

The format of all responses in FCL returned from `fcl.send(...)`. For full details on the values and descriptions of the keys, view [here](#).

Key
tag
transaction
transactionStatus
transactionId
encodedData
events
account
block
blockHeader
latestBlock
collection

Event Object

Key	Value Type	Description
blockId	string	ID of the block that contains the event.
blockHeight	number	Height of the block that contains the event.
blockTimestamp	string	The timestamp of when the block was sealed in a <code>DateString</code> format. eg. <code>'2021-06-25T13:42:04.227Z'</code>
type	EventName	A string containing the event name.
transactionId	string	Can be used to query transaction information, eg. via a Flow block explorer.
transactionIndex	number	Used to prevent replay attacks.
eventIndex	number	Used to prevent replay attacks.
data	any	The data emitted from the event.

Transaction Statuses

The status of a transaction will depend on the Flow blockchain network and which phase it is in as it completes and is finalized.

Status Code	Description
0	Unknown
1	Transaction Pending - Awaiting Finalization
2	Transaction Finalized - Awaiting Execution
3	Transaction Executed - Awaiting Sealing
4	Transaction Sealed - Transaction Complete. At this point the transaction result has been committed to the blockchain.
5	Transaction Expired

GRPC Statuses

The access node GRPC implementation follows the standard GRPC Core status code spec. [View here.](#)

FType

FCL arguments must specify one of the following support types for each value passed in.

Type	Example
UInt	fcl.arg(1, t.UInt)
UInt8	fcl.arg(8, t.UInt8)
UInt16	fcl.arg(16, t.UInt16)
UInt32	fcl.arg(32, t.UInt32)
UInt64	fcl.arg(64, t.UInt64)
UInt128	fcl.arg(128, t.UInt128)
UInt256	fcl.arg(256, t.UInt256)
Int	fcl.arg(1, t.Int)
Int8	fcl.arg(8, t.Int8)
Int16	fcl.arg(16, t.Int16)
Int32	fcl.arg(32, t.Int32)

```

| Int64      | fcl.arg(64, t.Int64)
|
| Int128     | fcl.arg(128, t.Int128)
|
| Int256     | fcl.arg(256, t.Int256)
|
| Word8      | fcl.arg(8, t.Word8)
|
| Word16     | fcl.arg(16, t.Word16)
|
| Word32     | fcl.arg(32, t.Word32)
|
| Word64     | fcl.arg(64, t.Word64)
|
| UFix64     | fcl.arg("64.123", t.UFix64)
|
| Fix64      | fcl.arg("64.123", t.Fix64)
|
| String      | fcl.arg("Flow", t.String)
|
| Character   | fcl.arg("c", t.String)
|
| Bool        | fcl.arg(true, t.String)
|
| Address     | fcl.arg("0xABCD123DEF456", t.Address)
|
| Optional    | fcl.arg("Flow", t.Optional(t.String))
|
| Array       | fcl.args([ fcl.arg(["First", "Second"], t.Array(t.String))
])
|
| Dictionary  | fcl.args([fcl.arg([{key: 1, value: "one"}, {key: 2, value: "two"}], t.Dictionary({key: t.Int, value: t.String}))])
|
| Path        | fcl.arg({ domain: "public", identifier: "flowTokenVault" }, t.Path)
|

```

StreamConnection

A stream connection is an object for subscribing to generic data from any WebSocket data stream. This is the base type for all stream connections. Two channels, `close` and `error`, are always available, as they are used to signal the end of the stream and any errors that occur.

```

ts
interface StreamConnection<ChannelMap extends { [name: string]: any }> {
  // Subscribe to a channel
  on<C extends keyof ChannelMap>(
    channel: C,
    listener: (data: ChannelMap[C]) => void,
  ): this;
  on(event: 'close', listener: () => void): this;
  on(event: 'error', listener: (err: any) => void): this;
}
```

```
// Unsubscribe from a channel
off<C extends keyof ChannelMap>(
  event: C,
  listener: (data: ChannelMap[C]) => void,
): this;
off(event: 'close', listener: () => void): this;
off(event: 'error', listener: (err: any) => void): this;

// Close the connection
close(): void;
}
```

Usage

```
ts
import { StreamConnection } from "@onflow/typedefs"

const stream: StreamConnection = ...

stream.on("close", () => {
  // Handle close
})

stream.on("error", (err) => {
  // Handle error
})

stream.close()
```

EventStream

An event stream is a stream connection that emits events and block heartbeats. Based on the connection parameters, heartbeats will be emitted at least as often as some fixed block height interval. It is a specific variant of a StreamConnection.

```
ts
type EventStream = StreamConnection<{
  events: Event[];
  heartbeat: BlockHeartbeat;
}>;
```

Usage

```
ts
import { EventStream } from "@onflow/typedefs"

const stream: EventStream = ...

stream.on("events", (events) => {
  // Handle events
```

```
)  
  
    stream.on("heartbeat", (heartbeat) => {  
        // Handle heartbeat  
    })  
  
    // Close the stream  
    stream.close()  
  
}
```

BlockHeartbeat

```
ts  
export interface BlockHeartbeat {  
    blockId: string;  
    blockHeight: number;  
    timestamp: string;  
}  
  
}
```

Usage

```
ts  
import { BlockHeartbeat } from "@onflow/typedefs"  
  
const heartbeat: BlockHeartbeat = ...
```

SignatureObject

Signature objects are used to represent a signature for a particular message as well as the account and keyId which signed for this message.

Key	Value Type
Description	
addr	Address the address of the account which this signature has been generated for
keyId	number The index of the key to use during authorization. (Multiple keys on an account is possible).
signature	string a hexadecimal-encoded string representation of the generated signature

```
# authentication.md:
```

Authentication

The concept of authentication in FCL is tied closely to FCL's concept of `currentUser`. In fact `fcl.authenticate` and `fcl.unauthenticate` are both aliases to `fcl.currentUser.authenticate()` and `fcl.currentUser.unauthenticate()` respectively. So let's look at `currentUser`.

As a dapp developer, using FCL, our current thought is to enable three main pieces of functionality.

- How to know the `currentUser` and if they are logged in.
- How to log a user in.
- How to log a user out.

Due to the nature of how FCL works, logging a user in and signing a user up are the same thing.

Knowing things about the current user

FCL provides two ways of getting the current users information. One way is a promise that returns a snapshot of the info, while the other way allows you to subscribe to info, calling a callback function with the latest info anytime it changes.

Snapshot of Current User

```
javascript
import as fcl from "@onflow/fcl"

const currentUser = await fcl.currentUser.snapshot()
console.log("The Current User", currentUser)
```

Subscribe to Current User

```
javascript
import as fcl from "@onflow/fcl"

// Returns an unsubscribe function
const unsubscribe = fcl.currentUser.subscribe(currentUser => {
  console.log("The Current User", currentUser)
})
```

Actually Authenticating and Unauthenticating

The TL;DR is to call `fcl.authenticate()` and `fcl.unauthenticate()` respectively.

On Flow mainnet, you wont even need to configure anything for this to work, the users of your dapp will go through the authentication process and be able to use any FCL compatible wallet providers.

During development you will probably want to configure your dapp to use @onflow/dev-wallet.

The Quick Start guide will walk you through using it.

We know this can all be fairly overwhelming, we are committed to help though. If you run into any problems, reach out to us on Discord, we are more than happy to help out.

```
# configure-fcl.md:
```

```
---
```

```
title: How to Configure FCL
```

```
---
```

Configuration

FCL has a mechanism that lets you configure various aspects of FCL. The main idea here (from an FCL perspective) should be that when you move from one instance of the Flow Blockchain to another (Local Emulator to Testnet to Mainnet) the only thing you should need to change (once again from an FCL perspective) is your configuration.

Setting Configuration Values

Values only need to be set once. We recommend doing this once and as early in the life cycle as possible.

To set a configuration value, the put method on the config instance needs to be called, the put method returns the config instance so they can be chained.

```
javascript
import as fcl from '@onflow/fcl';

fcl
  .config() // returns the config instance
  .put('foo', 'bar') // configures "foo" to be "bar"
  .put('baz', 'buz'); // configures "baz" to be "buz"
```

Getting Configuration Values

The config instance has an asynchronous get method. You can also pass it a fallback value incase the configuration state does not include what you are wanting.

```
javascript
import as fcl from '@onflow/fcl';

fcl.config().put('foo', 'bar').put('woot', 5).put('rawr', 7);

const FALBACK = 1;
```

```

async function addStuff() {
  var woot = await fcl.config().get('woot', FALBACK); // will be 5 --
  set in the config before
  var rawr = await fcl.config().get('rawr', FALBACK); // will be 7 --
  set in the config before
  var hmmm = await fcl.config().get('hmmm', FALBACK); // will be 1 --
  uses fallback because this isn't in the config

  return woot + rawr + hmmm;
}

addStuff().then((d) => console.log(d)); // 13 (5 + 7 + 1)

```

Common Configuration Keys

- accessNode.api -- Api URL for the Flow Blockchain Access Node you want to be communicating with.
- app.detail.title - (INTRODUCED @onflow/fcl@0.0.68) Your applications title, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.
- app.detail.icon - (INTRODUCED @onflow/fcl@0.0.68) Url for your applications icon, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.
- app.detail.description - (INTRODUCED @onflow/fcl@1.11.0) Your applications description, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.
- app.detail.url - (INTRODUCED @onflow/fcl@1.11.0) Your applications url, can be requested by wallets and other services. Used by WalletConnect plugin & Wallet Discovery service.
- challenge.handshake -- (DEPRECATED @onflow/fcl@0.0.68) Points FCL at the Wallet or Wallet Discovery mechanism.
- discovery.wallet -- (INTRODUCED @onflow/fcl@0.0.68) Points FCL at the Wallet or Wallet Discovery mechanism.
- discovery.wallet.method -- Describes which service strategy a wallet should use: IFRAAME/RPC, POP/RPC, TAB/RPC, HTTP/POST, EXT/RPC
- env -- (DEPRECATED @onflow/fcl@1.0.0) Used in conjunction with stored interactions. Possible values: local, testnet, mainnet
- fcl.limit -- Specifies fallback compute limit if not provided in transaction. Provided as integer.
- flow.network (recommended) -- (INTRODUCED @onflow/fcl@1.0.0) Used in conjunction with stored interactions and provides FCLCryptoContract address for testnet and mainnet. Possible values: local, testnet, mainnet.
- service.OpenID.scopes - (INTRODUCED @onflow/fcl@0.0.68) Open ID Connect claims for Wallets and OpenID services.
- walletconnect.projectId -- (INTRODUCED @onflow/fcl@1.11.0) Your app's WalletConnect project ID. See WalletConnect Cloud to obtain a project ID for your application.

Using Contracts in Scripts and Transactions

Address Replacement

Configuration keys that start with 0x will be replaced in FCL scripts and transactions, this allows you to write your script or transaction Cadence code once and not have to change it when you point your application at a difference instance of the Flow Blockchain.

```
javascript
import  as fcl from '@onflow/fcl';

fcl.config().put('0xFungibleToken', '0xf233dcee88fe0abe');

async function myScript() {
  return fcl
    .send([
      fcl.script
        import FungibleToken from 0xFungibleToken // will be replaced with
0xf233dcee88fe0abe because of the configuration

        access(all) fun main() { / Rest of the script goes here / }

    ])
    .then(fcl.decode);
}

async function myTransaction() {
  return fcl
    .send([
      fcl.transaction
        import FungibleToken from 0xFungibleToken // will be replaced with
0xf233dcee88fe0abe because of the configuration

        transaction { / Rest of the transaction goes here / }

    ])
    .then(fcl.decode);
}
```

Example

```
javascript
import  as fcl from '@onflow/fcl';

fcl
  .config()
  .put('flow.network', 'testnet')
  .put('accessNode.api', 'https://rest-testnet.onflow.org')
  .put('discovery.wallet', 'https://fcl-
discovery.onflow.org/testnet/authn')
  .put('walletconnect.projectId', 'YOURPROJECTID')
  .put('app.detail.title', 'Test Harness')
  .put('app.detail.icon', 'https://i.imgur.com/r23Zhvu.png')
  .put('app.detail.description', 'A test harness for FCL')
  .put('app.detail.url', 'https://myapp.com')
  .put('0xFlowToken', '0x7e60df042a9c0868');
```

Using Flow.json

A simpler way to import contracts in scripts and transactions is to use the config.load method to ingest your contracts from your flow.json file. This keeps the import syntax unified across tools and lets FCL figure out which address to use for what network based on the network provided in config. To use config.load you must first import your flow.json file and then pass it to config.load as a parameter.

```
javascript
import { config } from '@onflow/fcl';
import flowJSON from '../flow.json';

config({
  'flow.network': 'testnet',
  'accessNode.api': 'https://rest-testnet.onflow.org',
  'discovery.wallet': https://fcl-discovery.onflow.org/testnet/authn,
}).load({ flowJSON });
```

Let's say your flow.json file looks like this:

```
{
  "contracts": {
    "HelloWorld": "cadence/contracts/HelloWorld.cdc"
  }
}
```

Then in your scripts and transactions, all you have to do is:

```
import "HelloWorld"
```

FCL will automatically replace the contract name with the address for the network you are using.

> Note: never put private keys in your flow.json. You should use the key/location syntax to separate your keys into a separate git ignored file.

```
# discovery.md:

---
title: Wallet Discovery
---

Wallet Discovery
```

Knowing all the wallets available to users on a blockchain can be challenging. FCL's Discovery mechanism relieves much of the burden of integrating with Flow compatible wallets and let's developers focus on building their dapp and providing as many options as possible to their users.

There are two ways an app can use Discovery:

1. The UI version which can be configured for display via iFrame, Popup, or Tab.
2. The API version which allows you to access authentication services directly in your code via `fcl.discovery.authn` method which we'll describe below.

UI Version

When authenticating via FCL using Discovery UI, a user is shown a list of services they can use to login.

!FCL Default Discovery UI

This method is the simplest way to integrate Discovery and its wallets and services into your app. All you have to do is configure `discovery.wallet` with the host endpoint for testnet or mainnet.

> Note: Opt-in wallets, like Ledger and Dapper Wallet, require you to explicitly state you'd like to use them. For more information on including opt-in wallets, see these docs.
>
> A Dapper Wallet developer account is required. To enable Dapper Wallet inside FCL, you need to follow this guide.

```
javascript
import { config } from '@onflow/fcl';

config({
  'accessNode.api': 'https://rest-testnet.onflow.org',
  'discovery.wallet': 'https://fcl-discovery.onflow.org/testnet/authn',
});
```

Any time you call `fcl.authenticate` the user will be presented with that screen.

To change the default view from iFrame to popup or tab set `discovery.wallet.method` to POP/RPC (opens as a popup) or TAB/RPC (opens in a new tab). More info about service methods can be found [here](#).

Branding Discovery UI

Starting in version 0.0.79-alpha.4, dapps now have the ability to display app a title and app icon in the Discovery UI by setting a few values in their FCL app config. This branding provides users with messaging that has clear intent before authenticating to add a layer of trust.

All you have to do is set `app.detail.icon` and `app.detail.title` like this:

```
javascript
import { config } from '@onflow/fcl';

config({
  'app.detail.icon': 'https://placekitten.com/g/200/200',
  'app.detail.title': 'Kitten Dapp',
});
```

Note: If these configuration options aren't set, Dapps using the Discovery API will still display a default icon and "Unknown App" as the title when attempting to authorize a user who is not logged in. It is highly recommended to set these values accurately before going live.

API Version

If you want more control over your authentication UI, the Discovery API is also simple to use as it exposes Discovery directly in your code via `fcl`.

Setup still requires configuration of the Discovery endpoint, but when using the API it is set via `discovery.authn.endpoint` as shown below.

```
javascript
import { config } from '@onflow/fcl';

config({
  'accessNode.api': 'https://rest-testnet.onflow.org',
  'discovery.authn.endpoint':
    'https://fcl-discovery.onflow.org/api/testnet/authn',
});
```

You can access services in your Dapp from `fcl.discovery`:

```
javascript
import { fcl } from '@onflow/fcl';

fcl.discovery.authn.subscribe(callback);
// OR
fcl.discovery.authn.snapshot();
```

In order to authenticate with a service (for example, when a user click's "login"), pass the selected service to the `fcl.authenticate` method described here in the API reference:

```
jsx
fcl.authenticate({ service });
```

A simple React component may end up looking like this:

```
jsx
import './config';
import { useState, useEffect } from 'react';
import as fcl from '@onflow/fcl';

function Component() {
  const [services, setServices] = useState([]);
  useEffect(
    () => fcl.discovery.authn.subscribe((res) =>
      setServices(res.results)),
    []
  );

  return (
    <div>
      {services.map((service) => (
        <button
          key={service.provider.address}
          onClick={() => fcl.authenticate({ service })}
        >
          Login with {service.provider.name}
        </button>
      ))}
    </div>
  );
}
```

Helpful fields for your UI can be found in the provider object inside of the service. Fields include the following:

```
json
{
  ...
  "provider": {
    "address": "0xf086a545ce3c552d",
    "name": "Blocto",
    "icon": "/images/blocto.png",
    "description": "Your entrance to the blockchain world.",
    "color": "#afdf7",
    "supportEmail": "support@blocto.app",
    "authnendpoint": "https://flow-wallet-testnet.blocto.app/authn",
    "website": "https://blocto.portto.io"
  }
}
```

Network Configuration

Discovery UI URLs

Environment	Example
Mainnet	https://fcl-discovery.onflow.org/authn
Testnet	https://fcl-discovery.onflow.org/testnet/authn
Local	https://fcl-discovery.onflow.org/local/authn

Discovery API Endpoints

Environment	Example
Mainnet	https://fcl-discovery.onflow.org/api/authn
Testnet	https://fcl-discovery.onflow.org/api/testnet/authn
Local	https://fcl-discovery.onflow.org/api/local/authn

> Note: Local will return Dev Wallet on emulator for developing locally with the default port of 8701. If you'd like to override the default port add ?port=0000 with the port being whatever you'd like to override it to.

Other Configuration

> Note: Configuration works across both UI and API versions of Discovery.

Include Opt-In Wallets

Starting in FCL v0.0.78-alpha.10

Opt-in wallets are those that don't have support for authentication, authorization, and user signature services. Or, support only a limited set of transactions.

To include opt-in wallets from FCL:

```
import as fcl from "@onflow/fcl"

fcl.config({
  "discovery.wallet": "https://fcl-discovery.onflow.org/testnet/authn",
  "discovery.authn.endpoint": "https://fcl-
discovery.onflow.org/api/testnet/authn",
  "discovery.authn.include": ["0x123"] // Service account address
})
```

Opt-In Wallet Addresses on Testnet and Mainnet

Service	Testnet	Mainnet
Dapper Wallet	0x82ec283f88a62e65	0xead892083b3e2c6c
Ledger	0x9d2e44203cb13051	0xe5cd26afebe62781

To learn more about other possible configurations, check out the following links:

- Discovery API Docs
- Discovery Github Repo

```
# index.md:
```

```
---
```

```
sidebarposition: 3
```

```
---
```

Flow Client Library (FCL)

The Flow Client Library (FCL) JS is a package used to interact with user wallets and the Flow blockchain. When using FCL for authentication, dapps are able to support all FCL-compatible wallets on Flow and their users without any custom integrations or changes needed to the dapp code.

It was created to make developing applications that connect to the Flow blockchain easy and secure. It defines a standardized set of communication patterns between wallets, applications, and users that is used to perform a wide variety of actions for your dapp. FCL also offers a full featured SDK and utilities to interact with the Flow blockchain.

While FCL itself is a concept and standard, FCL JS is the javascript implementation of FCL and can be used in both browser and server environments. All functionality for connecting and communicating with wallet providers is restricted to the browser. We also have FCL Swift implementation for iOS, see FCL Swift contributed by @lmcmz.

```
---
```

Getting Started

Requirements

- Node version v12.0.0 or higher.

Installation

To use the FCL JS in your application, install using yarn or npm

```
shell
npm i -S @onflow/fcl
```

```
shell
yarn add @onflow/fcl
```

Importing

```
ES6
js
import as fcl from "@onflow/fcl";
```

```
Node.js
js
```

```

const fcl = require("@onflow/fcl");

---
FCL for Dapps
Wallet Interactions

- Wallet Discovery and Sign-up/Login: Onboard users with ease. Never
worry about supporting multiple wallets.
Authenticate users with any FCL compatible wallet.
js
// in the browser
import  as fcl from "@onflow/fcl"

fcl.config({
  "discovery.wallet": "https://fcl-discovery.onflow.org/testnet/authn",
  // Endpoint set to Testnet
})

fcl.authenticate()

!FCL Default Discovery UI

> Note: A Dapper Wallet developer account is required. To enable Dapper
Wallet inside FCL, you need to follow this guide.

- Interact with smart contracts: Authorize transactions via the user's
chosen wallet
- Prove ownership of a wallet address: Signing and verifying user signed
data

Learn more about wallet interactions >

Blockchain Interactions
- Query the chain: Send arbitrary Cadence scripts to the chain and
receive back decoded values
js
import  as fcl from "@onflow/fcl";

const result = await fcl.query({
  cadence:
    access(all) fun main(a: Int, b: Int, addr: Address): Int {
      log(addr)
      return a + b
    }
  ,
  args: (arg, t) => [
    arg(7, t.Int), // a: Int
    arg(6, t.Int), // b: Int
    arg("0xb1132bc08f82fe2", t.Address), // addr: Address
  ],
});
console.log(result); // 13

```

- Mutate the chain: Send arbitrary transactions with your own signatures or via a user's wallet to perform state changes on chain.

```
js
import  as fcl from "@onflow/fcl";
// in the browser, FCL will automatically connect to the user's wallet to
request signatures to run the transaction
const txId = await fcl.mutate({
  cadence:
    import Profile from 0xba1132bc08f82fe2

    transaction(name: String) {
      prepare(account: auth(BorrowValue) &Account) {
        account.storage.borrow<&{Profile.Owner}>(from:
Profile.privatePath)! .setName(name)
      }
    }
  ,
  args: (arg, t) => [arg("myName", t.String)],
});
```

[Learn more about on-chain interactions >](#)

Utilities

- Get account details from any Flow address
- Get the latest block
- Transaction status polling
- Event polling
- Custom authorization functions

[Learn more about utilities >](#)

Next Steps

See the [Flow App Quick Start](#).

See the [full API Reference](#) for all FCL functionality.

Learn Flow's smart contract language to build any script or transactions: Cadence.

Explore all of Flow docs and tools.

FCL for Wallet Providers

Wallet providers on Flow have the flexibility to build their user interactions and UI through a variety of ways:

- Front channel communication via Iframe, pop-up, tab, or extension
- Back channel communication via HTTP

FCL is agnostic to the communication channel and is configured to create both custodial and non-custodial wallets. This enables users to interact with wallet providers without needing to download an app or extension.

The communication channels involve responding to a set of pre-defined FCL messages to deliver the requested information to the dapp. Implementing a FCL compatible wallet on Flow is as simple as filling in the responses with the appropriate data when FCL requests them. If using any of the front-channel communication methods, FCL also provides a set of wallet utilities to simplify this process.

Current Wallet Providers

- Blocto
- Ledger (limited transaction support)
- Dapper Wallet
- Lilico
- Flipper
- NuFi

Wallet Discovery

It can be difficult to get users to discover new wallets on a chain. To solve this, we created a wallet discovery service that can be configured and accessed through FCL to display all available Flow wallet providers to the user. This means:

- Dapps can display and support all FCL compatible wallets that launch on Flow without needing to change any code
- Users don't need to sign up for new wallets - they can carry over their existing one to any dapp that uses FCL for authentication and authorization.

The discovery feature can be used via API, allowing you to customize your own UI or use the default UI without any additional configuration.

Building a FCL compatible wallet

- Read the wallet guide to understand the implementation details.
- Review the architecture of the Flow Dev Wallet for an overview.
- If building a non-custodial wallet, see the Account API and the FLIP on derivation paths and key generation.

Support

Notice a problem or want to request a feature? Add an issue.

Discuss FCL with the community on the forum.

Join the Flow community on Discord to keep up to date and to talk to the team.

scripts.md:

Scripts

Scripts let you run non-permanent Cadence scripts on the Flow blockchain. They can return data.

They always need to contain a `access(all) fun main()` function as an entry point to the script.

`fcl.query` is a function that sends Cadence scripts to the chain and receives back decoded responses.

The `cadence` key inside the object sent to the `query` function is a JavaScript Tagged Template Literal that we can pass Cadence code into.

Sending your first Script

In the following code snippet we are going to send a script to the Flow blockchain.

The script is going to add two numbers, and return them.

```
javascript
import  as fcl from "@onflow/fcl"

const response = await fcl.query({
  cadence:
    access(all) fun main(): Int {
      return 1 + 2
    }
  }

console.log(response) // 3
```

A more complicated Script

Things like Resources and Structs are fairly common place in Cadence.

In the following code snippet, our script defines a struct called `Point`, it then returns a list of them.

The closest thing to a Structure in JavaScript is an object. In this case when we decode this response, we would be expecting to get back an array of objects, where the objects have an `x` and `y` value.

```
javascript
import  as fcl from "@onflow/fcl"

const response = await fcl.query({
  cadence:
    access(all) struct Point {
      access(all) var x: Int
      access(all) var y: Int
    }
  }

console.log(response) // [ {x: 1, y: 2}, {x: 2, y: 3} ]
```

```

        init(x: Int, y: Int) {
            self.x = x
            self.y = y
        }
    }

    access(all) fun main(): [Point] {
        return [Point(x: 1, y: 1), Point(x: 2, y: 2)]
    }
}

console.log(response) // [{x:1, y:1}, {x:2, y:2}]

```

Transforming the data we get back with custom decoders.

In our dapp, we probably have a way of representing these Cadence values internally. In the above example it might be a Point class.

FCL enables us to provide custom decoders that we can use to transform the data we receive from the Flow blockchain at the edge, before anything else in our dapp gets a chance to look at it.

We add these custom decoders by Configuring FCL.

This lets us set it once when our dapp starts up and use our normalized data through out the rest of our dapp.

In the below example we will use the concept of a Point again, but this time, we will add a custom decoder, that enables fcl.decode to transform it into a custom JavaScript Point class.

```

javascript
import as fcl from "@onflow/fcl"

class Point {
    constructor({ x, y }) {
        this.x = x
        this.y = y
    }
}

fcl.config()
    .put("decoder.Point", point => new Point(point))

const response = await fcl.query({
    cadence:
        access(all) struct Point {
            access(all) var x: Int
            access(all) var y: Int

            init(x: Int, y: Int) {
                self.x = x
            }
        }
})

```

```
        self.y = y
    }
}

access(all) fun main(): [Point] {
    return [Point(x: 1, y: 1), Point(x: 2, y: 2)]
}

console.log(response) // [Point{x:1, y:1}, Point{x:2, y:2}]
```

To learn more about query, check out the API documentation.

sdk-guidelines.md:

```
---
title: SDK Reference
sidebarlabel: SDK Reference
sidebarposition: 2
---
```

Overview

This reference documents methods available in the SDK that can be accessed via FCL, and explains in detail how these methods work. FCL/SDKs are open source, and you can use them according to the licence.

The library client specifications can be found here:

```

```

Getting Started

Installing

NPM:

```
npm install --save @onflow/fcl @onflow/types
```

Yarn:

```
yarn add @onflow/fcl @onflow/types
```

Importing the Library

javascript

```
import as fcl from "@onflow/fcl"
import as types from "@onflow/types"
```

Connect

src="https://raw.githubusercontent.com/onflow/sdks/main/templates/documentation/ref.svg" width="130" />

By default, the library uses HTTP to communicate with the access nodes and it must be configured with the correct access node API URL. An error will be returned if the host is unreachable.

□ The HTTP/REST API information can be found here. The public Flow HTTP/REST access nodes are accessible at:

- Testnet <https://rest-testnet.onflow.org>
- Mainnet <https://rest-mainnet.onflow.org>
- Local Emulator 127.0.0.1:8888

Example:

```
javascript
import { config } from "@onflow/fcl"

config({
  "accessNode.api": "https://rest-testnet.onflow.org"
})
```

□ gRPC Access API URLs can be found here. sdk.transport must be specified if you wish to use the gRPC API. The public Flow gRPC access nodes are accessible at:

- Testnet <https://access-testnet.onflow.org>
- Mainnet <https://access-mainnet.onflow.org>
- Local Emulator 127.0.0.1:3569

For local development, use the flow emulator which once started provides an HTTP access endpoint at 127.0.0.1:8888 and a gRPC access endpoint at 127.0.0.1:3569.

If using the gRPC Access API, the sdk.transport configuration key must be populated as this value defaults to the HTTP API transport. The SDK can be configured to use the gRPC API transport as follows:

```
javascript
import { config } from "@onflow/fcl"
import { send as transportGRPC } from "@onflow/transport-grpc"

config({
  "accessNode.api": "https://access-testnet.onflow.org",
  "sdk.transport": transportGRPC
})
```

Querying the Flow Network

After you have established a connection with an access node, you can query the Flow network to retrieve data about blocks, accounts, events

and transactions. We will explore how to retrieve each of these entities in the sections below.

Get Blocks

```

```

Query the network for block by id, height or get the latest block.

□ Block ID is SHA3-256 hash of the entire block payload. This hash is stored as an ID field on any block response object (ie. response from GetLatestBlock).

□ Block height expresses the height of the block on the chain. The latest block height increases by one for every valid block produced.

Examples

This example depicts ways to get the latest block as well as any other block by height or ID:

```
import  as fcl from "@onflow/fcl";  
  
// Get latest block  
const latestBlock = await fcl.latestBlock(true); // If true, get the  
latest sealed block  
  
// Get block by ID (uses builder function)  
await fcl.send([fcl.getBlock(),  
fcl.atBlockId("23232323232")]).then(fcl.decode);  
  
// Get block at height (uses builder function)  
await fcl.send([fcl.getBlock(), fcl.atBlockHeight(123)]).then(fcl.decode)
```

Result output: BlockObject

Get Account

```

```

Retrieve any account from Flow network's latest block or from a specified block height.

□ Account address is a unique account identifier. Be mindful about the 0x prefix, you should use the prefix as a default representation but be careful and safely handle user inputs without the prefix.

An account includes the following data:

- Address: the account address.
- Balance: balance of the account.

- Contracts: list of contracts deployed to the account.
- Keys: list of keys associated with the account.

Examples

Example depicts ways to get an account at the latest block and at a specific block height:

```
javascript
import as fcl from "@onflow/fcl";

// Get account from latest block height
const account = await fcl.account("0x1d007d755706c469");

// Get account at a specific block height
fcl.send([
  fcl.getAccount("0x1d007d755706c469"),
  fcl.atBlockHeight(123)
]);
```

Result output: AccountObject

Get Transactions



The diagram illustrates the process of getting transaction status. It starts with a Transaction ID, which is then used to query the network for the transaction's status. The status is represented by a color-coded icon: red for UNKNOWN, yellow for PENDING, green for FINALIZED, and blue for EXECUTED. A legend below the diagram maps these colors to their respective meanings.

Retrieve transactions from the network by providing a transaction ID. After a transaction has been submitted, you can also get the transaction result to check the status.

■ Transaction ID is a hash of the encoded transaction payload and can be calculated before submitting the transaction to the network.

△ The transaction ID provided must be from the current spork.

□ Transaction status represents the state of a transaction in the blockchain. Status can change until it is finalized.

Status	Final	Description
-----	-----	-----
UNKNOWN	✗	The transaction has not yet been seen by the network
PENDING	✗	The transaction has not yet been included in a block
FINALIZED	✗	The transaction has been included in a block
EXECUTED	✗	The transaction has been executed but the result has not yet been sealed
SEALED	✓	The transaction has been executed and the result is sealed in a block

| EXPIRED | ✓ | The transaction reference block is outdated
before being executed |

```
javascript
import  as fcl from "@onflow/fcl";

// Snapshot the transaction at a point in time
fcl.tx(transactionId).snapshot();

// Subscribe to a transaction's updates
fcl.tx(transactionId).subscribe(callback);

// Provides the transaction once the status is finalized
fcl.tx(transactionId).onceFinalized();

// Provides the transaction once the status is executed
fcl.tx(transactionId).onceExecuted();

// Provides the transaction once the status is sealed
fcl.tx(transactionId).onceSealed();

Result output: TransactionStatusObject
```

Get Events

```

```

Retrieve events by a given type in a specified block height range or through a list of block IDs.

Event type is a string that follow a standard format:

A.{contract address}.{contract name}.{event name}

Please read more about events in the documentation. The exception to this standard are core events, and you should read more about them in this document.

- Block height range expresses the height of the start and end block in the chain.

Examples

Example depicts ways to get events within block range or by block IDs:

```

        fcl.getEventsAtBlockHeightRange (
            "A.7e60df042a9c0868.FlowToken.TokensWithdrawn", // event name
            35580624, // block to start looking for events at
            35580624 // block to stop looking for events at
        ),
    ])
.then(fcl.decode);

// Get events from list of block ids
await fcl
.send([

```

```

fcl.getEventsAtBlockIds ("A.7e60df042a9c0868.FlowToken.TokensWithdrawn", [
    "c4f239d49e96d1e5fbcf1f31027a6e582e8c03fc9954177b7723fdb03d938c7",
    "5dbaa85922eb194a3dc463c946cc01c866f2ff2b88f3e59e21c0d8d00113273f",
],
])
.then(fcl.decode);

```

Result output: EventObject

Get Collections



src="https://raw.githubusercontent.com/onflow/sdks/main/templates/documentation/ref.svg" width="130" />

Retrieve a batch of transactions that have been included in the same block, known as collections.

Collections are used to improve consensus throughput by increasing the number of transactions per block and they act as a link between a block and a transaction.

□ Collection ID is SHA3-256 hash of the collection payload.

Example retrieving a collection:

```

javascript
import  as fcl from "@onflow/fcl";

const collection = await fcl
.send([
    fcl.getCollection(
        "cccd80c67d015dc7f6444e8f62a3244ed650215ed66b90603006c70c5ef1f6e5"
    ),
])
.then(fcl.decode);

```

Result output: CollectionObject

Execute Scripts



src="https://raw.githubusercontent.com/onflow/sdks/main/templates/documentation/ref.svg" width="130" />

Scripts allow you to write arbitrary non-mutating Cadence code on the Flow blockchain and return data. You can learn more about Cadence here and scripts here, but we are now only interested in executing the script code and getting back the data.

We can execute a script using the latest state of the Flow blockchain or we can choose to execute the script at a specific time in history defined by a block height or block ID.

□ Block ID is SHA3-256 hash of the entire block payload, but you can get that value from the block response properties.

□ Block height expresses the height of the block in the chain.

```
javascript
import  as fcl from "@onflow/fcl";

const result = await fcl.query({
  cadence:
    access(all) fun main(a: Int, b: Int, addr: Address): Int {
      log(addr)
      return a + b
    }
  ,
  args: (arg, t) => [
    arg(7, t.Int), // a: Int
    arg(6, t.Int), // b: Int
    arg("0xba1132bc08f82fe2", t.Address), // addr: Address
  ],
});
```

Example output:

```
bash
console.log(result); // 13
```

Mutate Flow Network

Flow, like most blockchains, allows anybody to submit a transaction that mutates the shared global chain state. A transaction is an object that holds a payload, which describes the state mutation, and one or more authorizations that permit the transaction to mutate the state owned by specific accounts.

Transaction data is composed and signed with help of the SDK. The signed payload of transaction then gets submitted to the access node API. If a transaction is invalid or the correct number of authorizing signatures are not provided, it gets rejected.

Transactions

A transaction is nothing more than a signed set of data that includes script code which are instructions on how to mutate the network state and properties that define and limit its execution. All these properties are explained below.

- Script field is the portion of the transaction that describes the state mutation logic. On Flow, transaction logic is written in Cadence. Here is an example transaction script:

```
transaction(greeting: String) {  
    execute {  
        log(greeting.concat(", World!"))  
    }  
}
```

- Arguments. A transaction can accept zero or more arguments that are passed into the Cadence script. The arguments on the transaction must match the number and order declared in the Cadence script. Sample script from above accepts a single String argument.

- Proposal key must be provided to act as a sequence number and prevent replay and other potential attacks.

Each account key maintains a separate transaction sequence counter; the key that lends its sequence number to a transaction is called the proposal key.

A proposal key contains three fields:

- Account address
- Key index
- Sequence number

A transaction is only valid if its declared sequence number matches the current on-chain sequence number for that key. The sequence number increments by one after the transaction is executed.

- Payer is the account that pays the fees for the transaction. A transaction must specify exactly one payer. The payer is only responsible for paying the network and gas fees; the transaction is not authorized to access resources or code stored in the payer account.

- Authorizers are accounts that authorize a transaction to read and mutate their resources. A transaction can specify zero or more authorizers, depending on how many accounts the transaction needs to access.

The number of authorizers on the transaction must match the number of &Account parameters declared in the prepare statement of the Cadence script.

Example transaction with multiple authorizers:

```
transaction {  
    prepare(authorizer1: &Account, authorizer2: &Account) { }  
}
```

- Gas limit is the limit on the amount of computation a transaction requires, and it will abort if it exceeds its gas limit. Cadence uses metering to measure the number of operations per transaction. You can read more about it in the Cadence documentation.

The gas limit depends on the complexity of the transaction script. Until dedicated gas estimation tooling exists, it's best to use the emulator to test complex transactions and determine a safe limit.

- Reference block specifies an expiration window (measured in blocks) during which a transaction is considered valid by the network. A transaction will be rejected if it is submitted past its expiry block. Flow calculates transaction expiry using the reference block field on a transaction. A transaction expires after 600 blocks are committed on top of the reference block, which takes about 10 minutes at average Mainnet block rates.

```
Mutate

```

FCL "mutate" does the work of building, signing, and sending a transaction behind the scenes. In order to mutate the blockchain state using FCL, you need to do the following:

```
javascript
import  as fcl from "@onflow/fcl"

await fcl.mutate({
  cadence:
    transaction(a: Int) {
      prepare(acct: &Account) {
        log(acct)
        log(a)
      }
    }
  ,
  args: (arg, t) => [
    arg(6, t.Int)
  ],
  limit: 50
})
```

Flow supports great flexibility when it comes to transaction signing, we can define multiple authorizers (multi-sig transactions) and have different payer account than proposer. We will explore advanced signing scenarios below.

Single party, single signature

- Proposer, payer and authorizer are the same account (0x01).

- Only the envelope must be signed.
- Proposal key must have full signing weight.

Account	Key ID	Weight
0x01	1	1000

```

javascript
// There are multiple ways to achieve this
import { fcl } from "@onflow/fcl"

// FCL provides currentUser as an authorization function
await fcl.mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {}
    }
  ,
  proposer: currentUser,
  payer: currentUser,
  authorizations: [currentUser],
  limit: 50,
})

// Or, simplified

mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {}
    }
  ,
  authz: currentUser, // Optional. Will default to currentUser if not provided.
  limit: 50,
})

// Or, create a custom authorization function
const authzFn = async (txAccount) => {
  return {
    ...txAccount,
    addr: "0x01",
    keyId: 0,
    signingFunction: async(signable) => {
      return {
        addr: "0x01",
        keyId: 0,
        signature
      }
    }
  }
}

```

```

mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {}
    }
  ,
  proposer: authzFn,
  payer: authzFn,
  authorizations: [authzFn],
  limit: 50,
})

```

Single party, multiple signatures

- Proposer, payer and authorizer are the same account (0x01).
- Only the envelope must be signed.
- Each key has weight 500, so two signatures are required.

Account	Key ID	Weight
0x01	1	500
0x01	2	500

```


javascript
import { fcl } from "@onflow/fcl"

const authzFn = async (txAccount) => {
  return [
    {
      ...txAccount,
      addr: "0x01",
      keyId: 0,
      signingFunction: async(signable) => {
        return {
          addr: "0x01",
          keyId: 0,
          signature
        }
      }
    },
    {
      ...txAccount,
      addr: "0x01",
      keyId: 1,
      signingFunction: async(signable) => {
        return {
          addr: "0x01",
          keyId: 1,
          signature
        }
      }
    }
  ]
}

```

```

        }
    }
}

mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {}
    }
  ,
  proposer: authzFn,
  payer: authzFn,
  authorizations: [authzFn],
  limit: 50,
})
```

Multiple parties

- Proposer and authorizer are the same account (0x01).
- Payer is a separate account (0x02).
- Account 0x01 signs the payload.
- Account 0x02 signs the envelope.
 - Account 0x02 must sign last since it is the payer.

Account	Key ID	Weight
0x01	1	1000
0x02	3	1000

javascript
import as fcl from "@onflow/fcl"

```

const authzFn = async (txAccount) => {
  return {
    ...txAccount,
    addr: "0x01",
    keyId: 0,
    signingFunction: async(signable) => {
      return {
        addr: "0x01",
        keyId: 0,
        signature
      }
    }
  }
}

const authzTwoFn = async (txAccount) => {
  return {
```

```
        ...txAccount,
        addr: "0x02",
        keyId: 0,
        signingFunction: async(signable) => {
            return {
                addr: "0x02",
                keyId: 0,
                signature
            }
        }
    }
}

mutate({
    cadence:
        transaction {
            prepare(acct: &Account) {}
        }
    ,
    proposer: authzFn,
    payer: authzTwoFn,
    authorizations: [authzFn],
    limit: 50,
})
```

Multiple parties, two authorizers

- Proposer and authorizer are the same account (0x01).
 - Payer is a separate account (0x02).
 - Account 0x01 signs the payload.
 - Account 0x02 signs the envelope.
 - Account 0x02 must sign last since it is the payer.
 - Account 0x02 is also an authorizer to show how to include two &Account objects into an transaction

Account	Key ID	Weight
0x01	1	1000
0x02	3	1000

```
  
javascript  
import  as fcl from "@onflow/fcl"  
  
const authzFn = async (txAccount) => {  
  return {  
    ...txAccount,  
    addr: "0x01",  
    keyId: 0,  
    signingFunction: async(signable) => {  
      return {
```

```

        addr: "0x01",
        keyId: 0,
        signature
    }
}
}
}

const authzTwoFn = async (txAccount) => {
    return {
        ...txAccount,
        addr: "0x02",
        keyId: 0,
        signingFunction: async(signable) => {
            return {
                addr: "0x02",
                keyId: 0,
                signature
            }
        }
    }
}

mutate({
    cadence:
        transaction {
            prepare(acct: &Account, acct2: &Account) {}
        }
    ,
    proposer: authzFn,
    payer: authzTwoFn,
    authorizations: [authzFn, authzTwoFn],
    limit: 50,
})
```

Multiple parties, multiple signatures

- Proposer and authorizer are the same account (0x01).
- Payer is a separate account (0x02).
- Account 0x01 signs the payload.
- Account 0x02 signs the envelope.
 - Account 0x02 must sign last since it is the payer.
- Both accounts must sign twice (once with each of their keys).

Account	Key ID	Weight
0x01	1	500
0x01	2	500
0x02	3	500
0x02	4	500

```
javascript
import as fcl from "@onflow/fcl"
```

```
const authzFn = async (txAccount) => {
  return [
    {
      ...txAccount,
      addr: "0x01",
      keyId: 0,
      signingFunction: async(signable) => {
        return {
          addr: "0x01",
          keyId: 0,
          signature
        }
      }
    },
    {
      ...txAccount,
      addr: "0x01",
      keyId: 1,
      signingFunction: async(signable) => {
        return {
          addr: "0x01",
          keyId: 1,
          signature
        }
      }
    }
  ]
}

const authzTwoFn = async (txAccount) => {
  return [
    {
      ...txAccount,
      addr: "0x02",
      keyId: 0,
      signingFunction: async(signable) => {
        return {
          addr: "0x02",
          keyId: 0,
          signature
        }
      }
    },
    {
      ...txAccount,
      addr: "0x02",
      keyId: 1,
      signingFunction: async(signable) => {
        return {
          addr: "0x02",
          keyId: 1,
          signature
        }
      }
    }
  ]
}
```

```

        }
    }
]

mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {}
    }
  ,
  proposer: authzFn,
  payer: authzTwoFn,
  authorizations: [authzFn],
  limit: 50,
})
```

After a transaction has been built and signed, it can be sent to the Flow blockchain where it will be executed. If sending was successful you can then retrieve the transaction result.

transactions.md:

Transactions

Transactions let you send Cadence code to the Flow blockchain that permanently alters its state.

We are assuming you have read the Scripts Documentation before this, as transactions are sort of scripts with more required things.

While query is used for sending scripts to the chain, mutate is used for building and sending transactions. Just like scripts, fcl.mutate is a JavaScript Tagged Template Literal that we can pass Cadence code into.

Unlike scripts, they require a little more information, things like a proposer, authorizations and a payer, which may be a little confusing and overwhelming.

Sending your first Transaction

There is a lot to unpack in the following code snippet. It sends a transaction to the Flow blockchain. For the transaction, the current user is authorizing it as both the proposer and the payer. Something that is unique to Flow is the one paying for the transaction doesn't always need to be the one performing the transaction. Proposers and Payers are special kinds of authorizations that are always required for a transaction. The proposer acts similar to the nonce in Ethereum transactions, and helps prevent repeat attacks. The payer is who will be paying for the transaction.

If these are not set, FCL defaults to using the current user for all roles.

fcl.mutate will return a transactionId. We can pass the response directly to fcl.tx and then use the onceSealed method which resolves a promise when the transaction is sealed.

```
javascript
import  as fcl from "@onflow/fcl"

const transactionId = await fcl.mutate({
  cadence:
    transaction {
      execute {
        log("Hello from execute")
      }
    }
  ,
  proposer: fcl.currentUser,
  payer: fcl.currentUser,
  limit: 50
})

const transaction = await fcl.tx(transactionId).onceSealed()
console.log(transaction) // The transactions status and events after
being sealed
```

Authorizing a transaction

The below code snippet is the same as the above one, except for one extremely important difference.

Our Cadence code this time has a prepare statement, and we are using the fcl.currentUser when constructing our transaction.

The prepare statement's arguments directly map to the order of the authorizations in the authorizations array.
Four authorizations means four &Accounts as arguments passed to prepare. In this case though there is only one, and it is the currentUser.

These authorizations are important as you can only access/modify an accounts storage if you have the said accounts authorization.

```
javascript
import  as fcl from "@onflow/fcl"

const transactionId = await fcl.mutate({
  cadence:
    transaction {
      prepare(acct: &Account) {
        log("Hello from prepare")
      }
      execute {
        log("Hello from execute")
```

```
        }
    }

    proposer: fcl.currentUser,
    payer: fcl.currentUser,
    authorizations: [fcl.currentUser],
    limit: 50
))

const transaction = await fcl.tx(transactionId).onceSealed()
console.log(transaction) // The transactions status and events after
being sealed
```

To learn more about `mutate`, check out the API documentation.

user-signatures.md:

```
---
title: Signing and Verifying Arbitrary Data
---
```

Signing Arbitrary Data

Cryptographic signatures are a key part of the blockchain. They are used to prove ownership of an address without exposing its private key. While primarily used for signing transactions, cryptographic signatures can also be used to sign arbitrary messages.

FCL has a feature that lets you send arbitrary data to a configured wallet/service where the user may approve signing it with their private key/s.

Verifying User Signatures

What makes message signatures more interesting is that we can use Flow blockchain to verify the signatures. Cadence has a built-in function `publicKey.verify` that will verify a signature against a Flow account given the account address.

FCL includes a utility function, `AppUtils.verifyUserSignatures`, for verifying one or more signatures against an account's public key on the Flow blockchain.

You can use both in tandem to prove a user is in control of a private key or keys.

This enables cryptographically-secure login flow using a message-signing-based authentication mechanism with a user's public address as their identifier.

```
---
```

```
currentUser.signUserMessage()
```

A method to use allowing the user to personally sign data via FCL Compatible Wallets/Services.

> :Note: Requires authentication/configuration with an authorized signing service.

Arguments

Name	Type	Description
message	string	A hexadecimal string to be signed

Returns

Type	Description
Array	An Array of CompositeSignatures: {addr, keyId, signature}

Usage

```
javascript
import  as fcl from "@onflow/fcl"

const signMessage = async () => {
  const MSG = Buffer.from("FOO").toString("hex")
  try {
    return await fcl.currentUser.signUserMessage(MSG)
  } catch (error) {
    console.log(error)
  }
}
```

```
AppUtils.verifyUserSignatures
```

Note

⚠️ fcl.config.flow.network or options override is required to use this API. See FCL Configuration.

A method allowing applications to cryptographically verify the ownership of a Flow account by verifying a message was signed by a user's private key/s. This is typically used with the response from currentUser.signUserMessage.

Arguments

Name	Type	Description
message	string (required)	A hexadecimal string
compositeSignatures	Array (required)	An Array of CompositeSignatures
opts	Object (optional)	opts.fclCryptoContract can be provided to override FCLCryptoContract address for local development

Returns

Type	Description
Boolean	true if verified or false

Usage

```
javascript
/
Verify a valid signature/s for an account on Flow.

@param {string} msg - A message string in hexadecimal format
@param {Array} compSigs - An array of Composite Signatures
@param {string} compSigs[].addr - The account address
@param {number} compSigs[].keyId - The account keyId
@param {string} compSigs[].signature - The signature to verify
@param {Object} [opts={}] - Options object
@param {string} opts.fclCryptoContract - An optional override of Flow
account address where the FCLCrypto contract is deployed
@return {bool}
```

@example

```
const isValid = await fcl.AppUtils.verifyUserSignatures(
    Buffer.from('FOO').toString("hex"),
    [{ftype: "CompositeSignature", fvsn: "1.0.0", addr: "0x123", keyId:
0, signature: "abc123"}],
    {fclCryptoContract}
)
/
```

Examples

Use cases include cryptographic login, message validation, verifiable credentials, and others.

wallet-connect.md:

```
---
title: WalletConnect 2.0 Manual Configuration
---

:::warning
This guide is for advanced users who want to manually configure WalletConnect 2.0 with FCL-JS. Since @onflow/fcl@1.11.0, FCL-JS has supported WalletConnect 2.0 out of the box. For most users, we recommend using this built-in WalletConnect 2.0 support (see how to configure FCL-JS here).
:::
```

To improve developer experience and streamline Flow dApp integration with WalletConnect 2.0 wallets, FCL ^1.3.0 introduces support for discovery-service plugins. These ServicePlugins allow for injection of client configured services, service methods, and the execution strategies required to interact with them.

FCL dApps can opt-in through use of the fcl-wc package and FCL Plugin Registry.

When using FCL Discovery for authentication, dApps are able to support most FCL-compatible wallets and their users on Flow without any custom integrations or changes needed to the dApp code.

These instructions explain how dApps can also add support for FCL compatible wallets that use the WalletConnect 2.0 protocol.

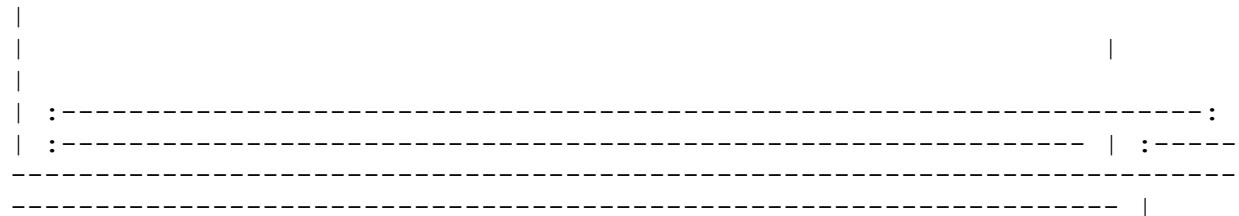
How does it work?

- The fcl-wc package is used to initialize a WalletConnect 2.0 SignClient instance, and build a discovery-service ServicePlugin based on dApp specified options.
- discovery-service plugins are used to add opt-in wallets and other services to FCL Wallet Discovery (UI/API).
- The FCL Plugin Registry offers dApps the ability to add new services, methods, and the execution strategies needed to interact with them.

Requirements

- fcl version >= 1.3.0
- fcl-wc version >= 1.0.0

Implementation path



```

|           1           | Add required packages
| Install and import minimum fcl and fcl-wc versions
|
|           2           | Obtain a WalletConnect projectID
| Visit WalletConnect Cloud Registry and register for public relay server
access and an application projectId |
| 3 | Initialize WalletConnect SignClient and FclWcServicePlugin |
Initialize WalletConnect SignClient and FclWcServicePlugin with
configuration options
|           4           | Add FclWcServicePlugin to FCL Plugin Registry
| Inject FclWcServicePlugin via
fcl.pluginRegistry.add(FclWcServicePlugin)
|

```

1. Add required packages

Install the fcl and fcl-wc packages:

```

bash
npm install @onflow/fcl@ @onflow/fcl-wc

```

2. Obtain a WalletConnect projectID

Visit WalletConnect Cloud Registry and register for public relay server access and an application projectId.

3. Initialize WalletConnect SignClient and FclWcServicePlugin

In addition to the WalletConnect SignClient, the init method of fcl-wc returns a ServicePlugin object. This object can be injected using the FCL Plugin Registry to add support for new service methods and their corresponding execution strategies (like WC/RPC for WalletConnect). A discovery-service ServicePlugin may also include additional opt-in wallets to offer your users through FCL Wallet Discovery.

Configuration options

Initialize WalletConnect SignClient and FclWcServicePlugin with the following configuration options:

Name	Type	Default	Description
projectId	boolean (required)	null	A WalletConnect projectId for public relay server access. Your Project ID can be obtained from WalletConnect Cloud Dashboard

```

| metadata | object | { } | Optional dApp
metadata to describe your application and define its appearance in a web
browser. More details can be found here
|
| includeBaseWC | boolean | false | Optional
configuration to include a generic WalletConnect service in FCL Discovery
(UI/API). <br /> :exclamation: BaseWC Service offers no deeplink support
for mobile.
|
| wcRequestHook | function | null | Optional
function is called on all desktop WalletConnect client session proposals
and signing requests. Use this to handle alerting user to check wallet
for approval.
|
| pairingModalOverride | function | null | Optional
function called to allow override of included QRCodeModal. Function
receives two arguments: <br /> 1. Connection uri to display QR code or
send to wallet to create pairing. <br /> 2. Callback function to manually
cancel the request.
|
| wallets | array | [ ] | Optional list
of WalletConnect authn services to include in FCL Wallet Discovery
(UI/API). <br /> :exclamation: Only available for use on testnet. These
services will be combined with wallets returned from WalletConnect cloud
registry API and sent to Discovery for display in UI and inclusion in API
response.

```

Returns

Name	Type
Description	
-----	-----
-----	-----
FclWcServicePlugin ServicePlugin A ServicePlugin of type discovery- service. May also include optional authn services to offer through FCL Wallet Discovery (UI/API).	
client SignClient An initialized WalletConnect SignClient.	

```

js
const FclWcServicePlugin = {
  name: "fcl-plugin-service-walletconnect",
  ftype: "ServicePlugin", // the type of FCL plugin
  type: "discovery-service", // the is a service sent to Discovery
  services: [Service], // (optional) Generic, Cloud Registry and
  client injected WalletConnect services
  serviceStrategy: {method: "WC/RPC", exec: execStrategy, // the method
  name and execution strategy for WalletConnect services
}

```

```
:exclamation: Setting FCL config flow.network to testnet or mainnet is required to use fcl-wc as it enables "WC/RPC" service strategy to request correct chain permissions.
```

```
import {config} from '@onflow/config'

config({
  "flow.network": "mainnet"
})
```

See FCL Configuration for more information.

4. Add FclWcServicePlugin to FCL Plugin Registry

In addition to the WalletConnect SignClient, the init method of fcl-wc returns a ServicePlugin object. This object can be injected into the FCL Plugin Registry to add FCL support for new service methods, (like WC/RPC for WalletConnect) and their corresponding execution strategies.

Usage

```
js
import as fcl from '@onflow/fcl';
import { init } from 'fcl-wc';

const { FclWcServicePlugin, client } = await init({
  projectId: WCPROJECTID, // required
  metadata: WCAPPMETADATA, // optional
  includeBaseWC: false, // optional, default: false
  wallets: [], // optional, default: []
  wcRequestHook: (wcrequestData) => {
    // optional, default: null
    handlePendingRequest(data);
  },
  pairingModalOverride: (uri, rejectPairingRequest) => {
    // optional, default: null
    handlePendingPairingRequest(data);
  },
});
fcl.pluginRegistry.add(FclWcServicePlugin);
```

ServicePlugin Spec

Key	Value Type	Description

```
----- |  
| name | string | The name of  
the plugin.  
|  
| ftype | string | The type of  
plugin (currently only supports ServicePlugin type).  
|  
| type | string | The plugin  
subtype (currently only supports discovery-service type).  
|  
| services | array | A list of  
services to add to FCL.  
|  
| serviceStrategy | { method: string, exec: function } | The method and  
corresponding strategy FCL uses to interact with the service. A service  
with the service.method property set to "WC/RPC" tells FCL to use the  
corresponding service strategy if it is supported by the dApp. |
```

Integrating With Wallet Discovery

Knowing all the wallets available to users on a blockchain can be challenging. FCL's Discovery mechanism relieves much of the burden of integrating with Flow compatible wallets and let's developers focus on building their dApp and providing as many options as possible to their users.

There are two ways an app can use Wallet Discovery:

1. The UI version which can be configured for display via iFrame, Popup, or Tab.
2. The API version which allows you to access authentication services directly in your code via `fcl.discovery.authn` method which we'll describe below.

When using FCL Wallet Discovery for authentication, dApps are able to support FCL-compatible wallets on Flow without any custom integrations or changes needed to the dApp code.

discovery-service ServicePlugin

`FclWcServicePlugin` is a `ServicePlugin` of type `discovery-service`. `discovery-service` plugins may include additional authentication services to offer through FCL Wallet Discovery.

Once a valid `discovery-service` plugin is registered, FCL shares client supported services with Discovery to add registered and injected wallets to the UI and API.

To connect a Flow supported wallet using WalletConnect 2.0, users of your dApp will go through the authentication process and have the option to select their preferred wallet.

:exclamation: Once a WalletConnect session is established and a currentUser is authenticated, FCL will handle client pairings and sessions during FCL authn, and initiate signing requests as part of authz using fcl.mutate and user-sign using fcl.signUserMessage.

How to add your FCL compatible WalletConnect wallet to Discovery (UI/API)

1. Submit a PR to add your wallet to FCL Wallet Discovery services.json
 2. Submit your FCL compatible wallet to WalletConnect Cloud Registry
 3. Add Wallet Service to fcl-wc init options. :exclamation: testnet only.

FCL tells Wallet Discovery which services are supported by the client (installed extensions and discovery-service ServicePlugins) so only those supported will be shown in Discovery UI or returned via Discovery API.

!Wallet Discovery UI

Wallet Provider Spec

Implementation path

```
|  
|  
| :---: | :---:  
-----|  
:  
-----|  
-----|  
-----|  
-----|  
| 1 | Obtain a WalletConnect projectId  
| Register to receive a projectId from the WalletConnect Cloud Registry.  
|  
| 2 | Conform to FCL Wallet Provider Spec | Compatible wallets must  
support flowauthn, flowauthz, and flowusersign methods and wrap data in  
the appropriate FCL Response type. Services returned with the  
AuthnResponse of flowauthn should set service.endpoint to corresponding  
methods.<br/> ServiceType authz : flowauthz <br/> ServiceType user-  
signature : flowusersign |  
| 3 | Add wallet to WalletConnect Cloud Registry API (optional)  
| Submit your wallet to be included in the WalletConnect Cloud Registry  
API and Explorer  
|  
| 4 | Responses  
| All responses need to be wrapped in a PollingResponse
```

Metadata requirements

:exclamation: In order to correctly identify, improve pairing, and include deep link support for mobile, services using the WC/RPC method need to use the same universal link as their uid and url in Wallet metadata.
Wallets sourced from WalletConnect Cloud Registry automatically build the service from data and will set the service.uid to the universal link.

```
javascript
import SignClient from '@walletconnect/sign-client';

export let signClient: SignClient;

export async function createSignClient() {
  signClient = await SignClient.init({
    projectId: PROJECTID,
    relayUrl: 'wss://relay.walletconnect.com',
    metadata: {
      name: 'Awesome Wallet',
      description: 'Awesome Wallet with FCL Support for WalletConnect',
      url: 'https://deeplink.awesome-wallet.com/',
      icons: ['https://avatars.githubusercontent.com/u/37784886'],
    },
  });
}
```

Next steps

1. Read the FCL Wallet Provider Spec.
2. Check out the a WalletConnect 2.0 React POC Wallet and FCL Flow dApp with support for WalletConnect v2.0.
3. Read and review the WalletConnect 2.0 Docs, examples and resources.

```
# index.md:

---
title: Flow Go SDK
---

<div align="center">
  <a href="/tooling/intro">
    
  </a>
  <p align="center">
    <br />
    <a href="https://github.com/onflow/flow-go-sdk">
      <strong>View on GitHub</strong>
    </a>
  </p>
</div>
```

```
<br />
<br />
<a href="../fcl-js/sdk-guidelines/">SDK Specifications</a>
<a href="https://github.com/onflow/flow-go-
sdk/blob/master/CONTRIBUTING.md">
    Contribute
</a>
<a href="https://github.com/onflow/flow-go-sdk/issues">Report a
Bug</a>
</p>
</div>
<br />
```

Overview

This reference documents all the methods available in the SDK, and explains in detail how these methods work.
SDKs are open source, and you can use them according to the licence.

The library client specifications can be found here:

```

```

Getting Started

Installing

The recommended way to install Go Flow SDK is by using Go modules.

If you already initialized your Go project, you can run the following command in your terminal:

```
sh
go get github.com/onflow/flow-go-sdk
```

It's usually good practice to pin your dependencies to a specific version.

Refer to the SDK releases page to identify the latest version.

Importing the Library

After the library has been installed you can import it.

```
go
import "github.com/onflow/flow-go-sdk"
```

Connect

```

```

The Go SDK library uses HTTP or gRPC APIs to communicate with the access nodes and it must be configured with correct access node API URL. The library provides default factories for connecting to Flow AN APIs and you can easily switch between HTTP or gRPC if you use the provided client interface.

You can check more examples for creating clients in the examples:

```

```

Basic Example:

```
go  
// common client interface  
var flowClient client.Client  
  
// initialize an http emulator client  
flowClient, err := http.NewClient(http.EmulatorHost)  
  
// initialize a gRPC emulator client  
flowClient, err = grpc.NewClient(grpc.EmulatorHost)
```

You can also initialize an HTTP client or gRPC client directly which will offer you access to network specific options, but be aware you won't be able to easily switch between those since they don't implement a common interface. This is only advisable if the implementation needs the access to those advanced options.

Advanced Example:

```
go  
// initialize http specific client  
httpClient, err := http.NewHTTPClient(http.EMULATORURL)  
  
// initialize grpc specific client  
grpcClient, err := grpc.NewGRPCClient(  
    grpc.EMULATORURL,  
    grpcOpts.WithTransportCredentials(insecure.NewCredentials()),  
)
```

Querying the Flow Network

After you have established a connection with an access node, you can query the Flow network to retrieve data about blocks, accounts, events and transactions. We will explore how to retrieve each of these entities in the sections below.

Get Blocks

```

```

Query the network for block by id, height or get the latest block.

□ Block ID is SHA3-256 hash of the entire block payload. This hash is stored as an ID field on any block response object (ie. response from GetLatestBlock).

□ Block height expresses the height of the block on the chain. The latest block height increases by one for every valid block produced.

Examples

This example depicts ways to get the latest block as well as any other block by height or ID:

```


go
func demo() {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()

    // get the latest sealed block
    isSealed := true
    latestBlock, err := flowClient.GetLatestBlock(ctx, isSealed)
    printBlock(latestBlock, err)

    // get the block by ID
    blockID := latestBlock.ID.String()
    blockByID, err := flowClient.GetBlockByID(ctx, flow.HexToID(blockID))
    printBlock(blockByID, err)

    // get block by height
    blockByHeight, err := flowClient.GetBlockByHeight(ctx, 0)
    printBlock(blockByHeight, err)
}

func printBlock(block flow.Block, err error) {
    examples.Handle(err)

    fmt.Printf("\nID: %s\n", block.ID)
    fmt.Printf("height: %d\n", block.Height)
    fmt.Printf("timestamp: %s\n\n", block.Timestamp)
}
```

Result output:

```
bash
ID: 835dc83939141097aa4297aa6cf69fc600863e3b5f9241a0d7feac1868adfa4f
height: 10
timestamp: 2021-10-06 15:06:07.105382 +0000 UTC
```

```
ID: 835dc83939141097aa4297aa6cf69fc600863e3b5f9241a0d7feac1868adfa4f
height: 10
timestamp: 2021-10-06 15:06:07.105382 +0000 UTC
```

```
ID: 7bc42fe85d32ca513769a74f97f7e1a7bad6c9407f0d934c2aa645ef9cf613c7
height: 0
timestamp: 2018-12-19 22:32:30.000000042 +0000 UTC
```

Get Account

```

```

Retrieve any account from Flow network's latest block or from a specified block height.

The GetAccount method is actually an alias for the get account at latest block method.

 Account address is a unique account identifier. Be mindful about the 0x prefix, you should use the prefix as a default representation but be careful and safely handle user inputs without the prefix.

An account includes the following data:

- **Address:** the account address.
- **Balance:** balance of the account.
- **Contracts:** list of contracts deployed to the account.
- **Keys:** list of keys associated with the account.

Examples

Example depicts ways to get an account at the latest block and at a specific block height:

```

```

```
go
func demo() {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()
```

```

// get account from the latest block
address := flow.HexToAddress("f8d6e0586b0a20c7")
account, err := flowClient.GetAccount(ctx, address)
printAccount(account, err)

// get account from the block by height 0
account, err = flowClient.GetAccountAtBlockHeight(ctx, address, 0)
printAccount(account, err)
}

func printAccount(account flow.Account, err error) {
    examples.Handle(err)

    fmt.Printf("\nAddress: %s", account.Address.String())
    fmt.Printf("\nBalance: %d", account.Balance)
    fmt.Printf("\nContracts: %d", len(account.Contracts))
    fmt.Printf("\nKeys: %d\n", len(account.Keys))
}

```

Result output:

```

bash
Address: f8d6e0586b0a20c7
Balance: 999999999999600000
Contracts: 2
Keys: 1

Address: f8d6e0586b0a20c7
Balance: 999999999999600000
Contracts: 2
Keys: 1

```

Get Transactions

Retrieve transactions from the network by providing a transaction ID. After a transaction has been submitted, you can also get the transaction result to check the status.

□ Transaction ID is a hash of the encoded transaction payload and can be calculated before submitting the transaction to the network.

⚠ The transaction ID provided must be from the current spork.

□ Transaction status represents the state of transaction in the blockchain. Status can change until it is sealed.

Status	Final	Description
UNKNOWN	✗	The transaction has not yet been seen by the network
PENDING	✗	The transaction has not yet been included in a block
FINALIZED	✗	The transaction has been included in a block
EXECUTED	✗	The transaction has been executed but the result has not yet been sealed
SEALED	✓	The transaction has been executed and the result is sealed in a block
EXPIRED	✓	The transaction reference block is outdated before being executed


```
go
func demo(txID flow.Identifier) {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()

    tx, err := flowClient.GetTransaction(ctx, txID)
    printTransaction(tx, err)

    txr, err := flowClient.GetTransactionResult(ctx, txID)
    printTransactionResult(txr, err)
}

func printTransaction(tx flow.Transaction, err error) {
    examples.Handle(err)

    fmt.Printf("\nID: %s", tx.ID().String())
    fmt.Printf("\nPayer: %s", tx.Payer.String())
    fmt.Printf("\nProposer: %s", tx.ProposalKey.Address.String())
    fmt.Printf("\nAuthorizers: %s", tx.Authorizers)
}

func printTransactionResult(txr flow.TransactionResult, err error) {
    examples.Handle(err)

    fmt.Printf("\nStatus: %s", txr.Status.String())
    fmt.Printf("\nError: %v", txr.Error)
}
```

Example output:

bash

```
ID: fb1272c57cdad79acf2fcf37576d82bf760e3008de66aa32a900c8cd16174e1c
Payer: f8d6e0586b0a20c7
Proposer: f8d6e0586b0a20c7
Authorizers: []
Status: SEALED
Error: <nil>
```

Get Events

Retrieve events by a given type in a specified block height range or through a list of block IDs.

- Event type is a string that follow a standard format:

```
A.{contract address}.{contract name}.{event name}
```

Please read more about events in the documentation. The exception to this standard are core events, and you should read more about them in this document.

- Block height range expresses the height of the start and end block in the chain.

Examples

Example depicts ways to get events within block range or by block IDs:

```


go
func demo(deployedContract flow.Account, runScriptTx flow.Transaction) {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()

    // Query for account creation events by type
    result, err := flowClient.GetEventsForHeightRange(ctx,
"flow.AccountCreated", 0, 30)
    printEvents(result, err)

    // Query for our custom event by type
    customType := fmt.Sprintf("AC.%s.EventDemo.EventDemo.Add",
deployedContract.Address.Hex())
    result, err = flowClient.GetEventsForHeightRange(ctx, customType, 0,
10)
    printEvents(result, err)
```

```

// Get events directly from transaction result
txResult, err := flowClient.GetTransactionResult(ctx,
runScriptTx.ID())
examples.Handle(err)
printEvent(txResult.Events)
}

func printEvents(result []client.BlockEvents, err error) {
examples.Handle(err)

for , block := range result {
    printEvent(block.Events)
}
}

func printEvent(events []flow.Event) {
for , event := range events {
    fmt.Printf("\n\nType: %s", event.Type)
    fmt.Printf("\nValues: %v", event.Value)
    fmt.Printf("\nTransaction ID: %s", event.TransactionID)
}
}

```

Example output:

```

bash
Type: flow.AccountCreated
Values: flow.AccountCreated(address: 0xfd43f9148d4b725d)
Transaction ID:
ba9d53c8dcb0f9c2f854f93da8467a22d053eab0c540bde0b9ca2f7ad95eb78e

Type: flow.AccountCreated
Values: flow.AccountCreated(address: 0xeb179c27144f783c)
Transaction ID:
8ab7bfef3de1cf8b2ffb36559446100bf4129a9aa88d6bc59f72a467acf0c801

...
Type: A.eb179c27144f783c.EventDemo.Add
Values: A.eb179c27144f783c.EventDemo.Add(x: 2, y: 3, sum: 5)
Transaction ID:
f3a2e33687ad23b0e02644ebbdcd74a7cd8ea7214065410a8007811d0bcd353

```

Get Collections

```



```

Retrieve a batch of transactions that have been included in the same block, known as collections.

Collections are used to improve consensus throughput by increasing the number of transactions per block and they act as a link between a block and a transaction.

- Collection ID is SHA3-256 hash of the collection payload.

Example retrieving a collection:

```
go
func demo(exampleCollectionID flow.Identifier) {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()

    // get collection by ID
    collection, err := flowClient.GetCollection(ctx, exampleCollectionID)
    printCollection(collection, err)
}

func printCollection(collection flow.Collection, err error) {
    examples.Handle(err)

    fmt.Printf("\nID: %s", collection.ID().String())
    fmt.Printf("\nTransactions: %s", collection.TransactionIDs())
}
```

Example output:

```
bash
ID: 3d7b8037381f2497d83f2f9e09422c036aae2a59d01a7693fb6003b4d0bc3595
Transactions:
[cf1184e3de4bd9a7232ca3d0b9dd2cfbf96c97888298b81a05c086451fa52ec1]
```

Execute Scripts

```

```

Scripts allow you to write arbitrary non-mutating Cadence code on the Flow blockchain and return data. You can learn more about Cadence and scripts here, but we are now only interested in executing the script code and getting back the data.

We can execute a script using the latest state of the Flow blockchain or we can choose to execute the script at a specific time in history defined by a block height or block ID.

- Block ID is SHA3-256 hash of the entire block payload, but you can get that value from the block response properties.

- Block height expresses the height of the block in the chain.

```

```

```
go  
func demo() {  
    ctx := context.Background()  
    flowClient := examples.NewFlowClient()  
  
    script := []byte(  
        access(all) fun main(a: Int): Int {  
            return a + 10  
        }  
    )  
    args := []cadence.Value{ cadence.NewInt(5) }  
    value, err := flowClient.ExecuteScriptAtLatestBlock(ctx, script,  
args)  
  
    examples.Handle(err)  
    fmt.Printf("\nValue: %s", value.String())  
  
    complexScript := []byte(  
        access(all) struct User {  
            access(all) var balance: UFix64  
            access(all) var address: Address  
            access(all) var name: String  
  
            init(name: String, address: Address, balance: UFix64) {  
                self.name = name  
                self.address = address  
                self.balance = balance  
            }  
        }  
  
        access(all) fun main(name: String): User {  
            return User(  
                name: name,  
                address: 0x1,  
                balance: 10.0  
            )  
        }  
    )  
    args = []cadence.Value{ cadence.NewString("Dete") }  
    value, err = flowClient.ExecuteScriptAtLatestBlock(ctx,  
complexScript, args)  
    printComplexScript(value, err)  
}  
  
type User struct {  
    balance uint64  
    address flow.Address  
    name string  
}
```

```

func printComplexScript(value cadence.Value, err error) {
    examples.Handle(err)
    fmt.Printf("\nString value: %s", value.String())

    s := value.(cadence.Struct)
    u := User{
        balance: s.Fields[0].ToGoValue().(uint64),
        address: s.Fields[1].ToGoValue().([flow.AddressLength]byte),
        name:     s.Fields[2].ToGoValue().(string),
    }

    fmt.Printf("\nName: %s", u.name)
    fmt.Printf("\nAddress: %s", u.address.String())
    fmt.Printf("\nBalance: %d", u.balance)
}

```

Example output:

```

bash
Value: 15
String value:
s.34a17571e1505cf6770e6ef16ca387e345e9d54d71909f23a7ec0d671cd2faf5.User(b
alance: 10.00000000, address: 0x1, name: "Dete")
Name: Dete
Address: 0000000000000001
Balance: 1000000000

```

Mutate Flow Network

Flow, like most blockchains, allows anybody to submit a transaction that mutates the shared global chain state. A transaction is an object that holds a payload, which describes the state mutation, and one or more authorizations that permit the transaction to mutate the state owned by specific accounts.

Transaction data is composed and signed with help of the SDK. The signed payload of transaction then gets submitted to the access node API. If a transaction is invalid or the correct number of authorizing signatures are not provided, it gets rejected.

Executing a transaction requires couple of steps:

- Building transaction.
- Signing transaction.
- Sending transaction.

Transactions

A transaction is nothing more than a signed set of data that includes script code which are instructions on how to mutate the network state and

properties that define and limit its execution. All these properties are explained below.

□ Script field is the portion of the transaction that describes the state mutation logic. On Flow, transaction logic is written in Cadence. Here is an example transaction script:

```
transaction(greeting: String) {  
    execute {  
        log(greeting.concat(", World!"))  
    }  
}
```

□ Arguments. A transaction can accept zero or more arguments that are passed into the Cadence script. The arguments on the transaction must match the number and order declared in the Cadence script. Sample script from above accepts a single String argument.

□ Proposal key must be provided to act as a sequence number and prevent reply and other potential attacks.

Each account key maintains a separate transaction sequence counter; the key that lends its sequence number to a transaction is called the proposal key.

A proposal key contains three fields:

- Account address
- Key index
- Sequence number

A transaction is only valid if its declared sequence number matches the current on-chain sequence number for that key. The sequence number increments by one after the transaction is executed.

□ Payer is the account that pays the fees for the transaction. A transaction must specify exactly one payer. The payer is only responsible for paying the network and gas fees; the transaction is not authorized to access resources or code stored in the payer account.

□ Authorizers are accounts that authorize a transaction to read and mutate their resources. A transaction can specify zero or more authorizers, depending on how many accounts the transaction needs to access.

The number of authorizers on the transaction must match the number of &Account parameters declared in the prepare statement of the Cadence script.

Example transaction with multiple authorizers:

```
transaction {
    prepare(authorizer1: &Account, authorizer2: &Account) { }
}
```

Gas Limit

◻ Gas limit is the limit on the amount of computation a transaction requires, and it will abort if it exceeds its gas limit. Cadence uses metering to measure the number of operations per transaction. You can read more about it in the Cadence documentation.

The gas limit depends on the complexity of the transaction script. Until dedicated gas estimation tooling exists, it's best to use the emulator to test complex transactions and determine a safe limit.

Reference Block

◻ Reference block specifies an expiration window (measured in blocks) during which a transaction is considered valid by the network. A transaction will be rejected if it is submitted past its expiry block. Flow calculates transaction expiry using the reference block field on a transaction. A transaction expires after 600 blocks are committed on top of the reference block, which takes about 10 minutes at average Mainnet block rates.

Build Transactions

```

```

Building a transaction involves setting the required properties explained above and producing a transaction object.

Here we define a simple transaction script that will be used to execute on the network and serve as a good learning example.

```
transaction(greeting: String) {

    let guest: Address

    prepare(authorizer: &Account) {
        self.guest = authorizer.address
    }

    execute {
        log(greeting.concat(",")).concat(self.guest.toString()))
    }
}
```

```


go
import (
    "context"
    "os"
    "github.com/onflow/flow-go-sdk"
    "github.com/onflow/flow-go-sdk/client"
)
func main() {

    greeting, err := os.ReadFile("Greeting2.cdc")
    if err != nil {
        panic("failed to load Cadence script")
    }

    proposerAddress := flow.HexToAddress("9a0766d93b6608b7")
    proposerKeyIndex := 3

    payerAddress := flow.HexToAddress("631e88ae7f1d7c20")
    authorizerAddress := flow.HexToAddress("7aad92e5a0715d21")

    var accessAPIHost string

    // Establish a connection with an access node
    flowClient := examples.NewFlowClient()

    // Get the latest sealed block to use as a reference block
    latestBlock, err :=
        flowClient.GetLatestBlockHeader(context.Background(), true)
    if err != nil {
        panic("failed to fetch latest block")
    }

    // Get the latest account info for this address
    proposerAccount, err :=
        flowClient.GetAccountAtLatestBlock(context.Background(), proposerAddress)
    if err != nil {
        panic("failed to fetch proposer account")
    }

    // Get the latest sequence number for this key
    sequenceNumber := proposerAccount.Keys[proposerKeyIndex].SequenceNumber

    tx := flow.NewTransaction().
        SetScript(greeting).
        SetComputeLimit(100).
        SetReferenceBlockID(latestBlock.ID).
        SetProposalKey(proposerAddress, proposerKeyIndex, sequenceNumber).
        SetPayer(payerAddress).
```

```

    AddAuthorizer(authorizerAddress)

    // Add arguments last

    hello := cadence.NewString("Hello")

    err = tx.AddArgument(hello)
    if err != nil {
        panic("invalid argument")
    }
}

```

After you have successfully built a transaction the next step in the process is to sign it.

Sign Transactions



Flow introduces new concepts that allow for more flexibility when creating and signing transactions.

Before trying the examples below, we recommend that you read through the transaction signature documentation the next step in the process is to sign it. Flow transactions have envelope and payload signatures, and you should learn about each in the signature documentation.

Quick example of building a transaction:

```

go
import (
    "github.com/onflow/flow-go-sdk"
    "github.com/onflow/flow-go-sdk/crypto"
)

var (
    myAddress    flow.Address
    myAccountKey flow.AccountKey
    myPrivateKey crypto.PrivateKey
)

tx := flow.NewTransaction().
    SetScript([]byte("transaction { execute { log(\"Hello, World!\") } }")).
    SetComputeLimit(100).
    SetProposalKey(myAddress, myAccountKey.Index,
myAccountKey.SequenceNumber).
    SetPayer(myAddress)

```

Transaction signing is done through the `crypto.Signer` interface. The simplest (and least secure) implementation of `crypto.Signer` is `crypto.InMemorySigner`.

Signatures can be generated more securely using keys stored in a hardware device such as an HSM. The `crypto.Signer` interface is intended to be flexible enough to support a variety of signer implementations and is not limited to in-memory implementations.

Simple signature example:

```
go
// construct a signer from your private key and configured hash algorithm
mySigner, err := crypto.NewInMemorySigner(myPrivateKey,
myAccountKey.HashAlgo)
if err != nil {
    panic("failed to create a signer")
}

err = tx.SignEnvelope(myAddress, myAccountKey.Index, mySigner)
if err != nil {
    panic("failed to sign transaction")
}
```

Flow supports great flexibility when it comes to transaction signing, we can define multiple authorizers (multi-sig transactions) and have different payer account than proposer. We will explore advanced signing scenarios below.

Single party, single signature

- Proposer, payer and authorizer are the same account (0x01).
- Only the envelope must be signed.
- Proposal key must have full signing weight.

Account	Key ID	Weight
-----	-----	-----
0x01	1	1000

```

```

```
go
account1,   := c.GetAccount(ctx, flow.HexToAddress("01"))

key1 := account1.Keys[0]

// create signer from securely-stored private key
key1Signer := getSignerForKey1()

referenceBlock,   := flow.GetLatestBlock(ctx, true)
tx := flow.NewTransaction().
```

```

SetScript([]byte(
    transaction {
        prepare(signer: &Account) { log(signer.address) }
    }
)).
SetComputeLimit(100).
SetProposalKey(account1.Address, key1.Index, key1.SequenceNumber).
SetReferenceBlockID(referenceBlock.ID).
SetPayer(account1.Address).
AddAuthorizer(account1.Address)

// account 1 signs the envelope with key 1
err := tx.SignEnvelope(account1.Address, key1.Index, key1Signer)

```

Single party, multiple signatures

- Proposer, payer and authorizer are the same account (0x01).
- Only the envelope must be signed.
- Each key has weight 500, so two signatures are required.

Account	Key ID	Weight
-----	-----	-----
0x01	1	500
0x01	2	500


```

go
account1,   := c.GetAccount(ctx, flow.HexToAddress("01"))

key1 := account1.Keys[0]
key2 := account1.Keys[1]

// create signers from securely-stored private keys
key1Signer := getSignerForKey1()
key2Signer := getSignerForKey2()

referenceBlock,   := flow.GetLatestBlock(ctx, true)
tx := flow.NewTransaction().
    SetScript([]byte(
        transaction {
            prepare(signer: &Account) { log(signer.address) }
        }
)).
    SetComputeLimit(100).
    SetProposalKey(account1.Address, key1.Index, key1.SequenceNumber).
    SetReferenceBlockID(referenceBlock.ID).
    SetPayer(account1.Address).
    AddAuthorizer(account1.Address)

// account 1 signs the envelope with key 1

```

```

err := tx.SignEnvelope(account1.Address, key1.Index, key1Signer)

// account 1 signs the envelope with key 2
err = tx.SignEnvelope(account1.Address, key2.Index, key2Signer)

Multiple parties

- Proposer and authorizer are the same account (0x01).
- Payer is a separate account (0x02).
- Account 0x01 signs the payload.
- Account 0x02 signs the envelope.
  - Account 0x02 must sign last since it is the payer.

| Account | Key ID | Weight |
| ----- | ----- | ----- |
| 0x01   | 1      | 1000   |
| 0x02   | 3      | 1000   |



```

```

go
account1,   := c.GetAccount(ctx, flow.HexToAddress("01"))
account2,   := c.GetAccount(ctx, flow.HexToAddress("02"))

key1 := account1.Keys[0]
key3 := account2.Keys[0]

// create signers from securely-stored private keys
key1Signer := getSignerForKey1()
key3Signer := getSignerForKey3()

referenceBlock,   := flow.GetLatestBlock(ctx, true)
tx := flow.NewTransaction().
    SetScript([]byte(
        transaction {
            prepare(signer: &Account) { log(signer.address) }
        }
    )).
    SetComputeLimit(100).
    SetProposalKey(account1.Address, key1.Index, key1.SequenceNumber).
    SetReferenceBlockID(referenceBlock.ID).
    SetPayer(account2.Address).
    AddAuthorizer(account1.Address)

// account 1 signs the payload with key 1
err := tx.SignPayload(account1.Address, key1.Index, key1Signer)

// account 2 signs the envelope with key 3
// note: payer always signs last
err = tx.SignEnvelope(account2.Address, key3.Index, key3Signer)

```

Multiple parties, two authorizers

- Proposer and authorizer are the same account (0x01).
- Payer is a separate account (0x02).
- Account 0x01 signs the payload.
- Account 0x02 signs the envelope.
 - Account 0x02 must sign last since it is the payer.
- Account 0x02 is also an authorizer to show how to include two &Account objects into an transaction

Account	Key ID	Weight
0x01	1	1000
0x02	3	1000

```

```

```
go  
account1,   := c.GetAccount(ctx, flow.HexToAddress("01"))  
account2,   := c.GetAccount(ctx, flow.HexToAddress("02"))  
  
key1 := account1.Keys[0]  
key3 := account2.Keys[0]  
  
// create signers from securely-stored private keys  
key1Signer := getSignerForKey1()  
key3Signer := getSignerForKey3()  
  
referenceBlock,   := flow.GetLatestBlock(ctx, true)  
tx := flow.NewTransaction().  
    SetScript([]byte(  
        transaction {  
            prepare(signer1: &Account, signer2: &Account) {  
                log(signer.address)  
                log(signer2.address)  
            }  
        }  
    )).  
    SetComputeLimit(100).  
    SetProposalKey(account1.Address, key1.Index, key1.SequenceNumber).  
    SetReferenceBlockID(referenceBlock.ID).  
    SetPayer(account2.Address).  
    AddAuthorizer(account1.Address).  
    AddAuthorizer(account2.Address)  
  
    // account 1 signs the payload with key 1  
    err := tx.SignPayload(account1.Address, key1.Index, key1Signer)  
  
    // account 2 signs the envelope with key 3  
    // note: payer always signs last  
    err = tx.SignEnvelope(account2.Address, key3.Index, key3Signer)
```

Multiple parties, multiple signatures

- Proposer and authorizer are the same account (0x01).
- Payer is a separate account (0x02).
- Account 0x01 signs the payload.
- Account 0x02 signs the envelope.
 - Account 0x02 must sign last since it is the payer.
- Both accounts must sign twice (once with each of their keys).

Account	Key ID	Weight
0x01	1	500
0x01	2	500
0x02	3	500
0x02	4	500


```
go
account1,   := c.GetAccount(ctx, flow.HexToAddress("01"))
account2,   := c.GetAccount(ctx, flow.HexToAddress("02"))

key1 := account1.Keys[0]
key2 := account1.Keys[1]
key3 := account2.Keys[0]
key4 := account2.Keys[1]

// create signers from securely-stored private keys
key1Signer := getSignerForKey1()
key2Signer := getSignerForKey1()
key3Signer := getSignerForKey3()
key4Signer := getSignerForKey4()

referenceBlock,   := flow.GetLatestBlock(ctx, true)
tx := flow.NewTransaction().
    SetScript([]byte(
        transaction {
            prepare(signer: &Account) { log(signer.address) }
        }
    )).
    SetComputeLimit(100).
    SetProposalKey(account1.Address, key1.Index, key1.SequenceNumber).
    SetReferenceBlockID(referenceBlock.ID).
    SetPayer(account2.Address).
    AddAuthorizer(account1.Address)

// account 1 signs the payload with key 1
err := tx.SignPayload(account1.Address, key1.Index, key1Signer)

// account 1 signs the payload with key 2
```

```
err = tx.SignPayload(account1.Address, key2.Index, key2Signer)

// account 2 signs the envelope with key 3
// note: payer always signs last
err = tx.SignEnvelope(account2.Address, key3.Index, key3Signer)

// account 2 signs the envelope with key 4
// note: payer always signs last
err = tx.SignEnvelope(account2.Address, key4.Index, key4Signer)
```

Send Transactions

```

```

After a transaction has been built and signed, it can be sent to the Flow blockchain where it will be executed. If sending was successful you can then retrieve the transaction result.

```

```

```
go
func demo(tx flow.Transaction) {
    ctx := context.Background()
    flowClient := examples.NewFlowClient()

    err := flowClient.SendTransaction(ctx, tx)
    if err != nil {
        fmt.Println("error sending transaction", err)
    }
}
```

Create Accounts

```

```

On Flow, account creation happens inside a transaction. Because the network allows for a many-to-many relationship between public keys and accounts, it's not possible to derive a new account address from a public key offline.

The Flow VM uses a deterministic address generation algorithm to assign account addresses on chain. You can find more details about address generation in the [accounts & keys documentation](#).

Public Key

Flow uses ECDSA key pairs to control access to user accounts. Each key pair can be used in combination with the SHA2-256 or SHA3-256 hashing algorithms.

⚠ You'll need to authorize at least one public key to control your new account.

Flow represents ECDSA public keys in raw form without additional metadata. Each key is a single byte slice containing a concatenation of its X and Y components in big-endian byte form.

A Flow account can contain zero (not possible to control) or more public keys, referred to as account keys. Read more about accounts in the documentation.

An account key contains the following data:

- Raw public key (described above)
- Signature algorithm
- Hash algorithm
- Weight (integer between 0-1000)

Account creation happens inside a transaction, which means that somebody must pay to submit that transaction to the network. We'll call this person the account creator. Make sure you have read sending a transaction section first.

```
go
var (
    creatorAddress      flow.Address
    creatorAccountKey  flow.AccountKey
    creatorSigner       crypto.Signer
)

var accessAPIHost string

// Establish a connection with an access node
flowClient := examples.NewFlowClient()

// Use the templates package to create a new account creation transaction
tx := templates.CreateAccount([]flow.AccountKey{accountKey}, nil,
    creatorAddress)

// Set the transaction payer and proposal key
tx.SetPayer(creatorAddress)
tx.SetProposalKey(
    creatorAddress,
    creatorAccountKey.Index,
    creatorAccountKey.SequenceNumber,
)

// Get the latest sealed block to use as a reference block
latestBlock, err := flowClient.GetLatestBlockHeader(context.Background(), true)
```

```

if err != nil {
    panic("failed to fetch latest block")
}

tx.SetReferenceBlockID(latestBlock.ID)

// Sign and submit the transaction
err = tx.SignEnvelope(creatorAddress, creatorAccountKey.Index,
creatorSigner)
if err != nil {
    panic("failed to sign transaction envelope")
}

err = flowClient.SendTransaction(context.Background(), tx)
if err != nil {
    panic("failed to send transaction to network")
}

```

After the account creation transaction has been submitted you can retrieve the new account address by getting the transaction result.

The new account address will be emitted in a system-level `flow.AccountCreated` event.

```

go
result, err := flowClient.GetTransactionResult(ctx, tx.ID())
if err != nil {
    panic("failed to get transaction result")
}

var newAddress flow.Address

if result.Status != flow.TransactionStatusSealed {
    panic("address not known until transaction is sealed")
}

for , event := range result.Events {
    if event.Type == flow.EventAccountCreated {
        newAddress = flow.AccountCreatedEvent(event).Address()
        break
    }
}

```

Generate Keys

```



```

Flow uses ECDSA signatures to control access to user accounts. Each key pair can be used in combination with the SHA2-256 or SHA3-256 hashing algorithms.

Here's how to generate an ECDSA private key for the P-256 (secp256r1) curve.

```
go
import "github.com/onflow/flow-go-sdk/crypto"

// deterministic seed phrase
// note: this is only an example, please use a secure random generator
// for the key seed
seed := []byte("elephant ears space cowboy octopus rodeo potato cannon
pineapple")

privateKey, err := crypto.GeneratePrivateKey(crypto.ECDsap256, seed)

// the private key can then be encoded as bytes (i.e. for storage)
encPrivateKey := privateKey.Encode()
// the private key has an accompanying public key
publicKey := privateKey.PublicKey()
```

The example above uses an ECDSA key pair on the P-256 (secp256r1) elliptic curve. Flow also supports the secp256k1 curve used by Bitcoin and Ethereum. Read more about supported algorithms [here](#).

Transferring Flow

This is an example of how to construct a FLOW token transfer transaction with the Flow Go SDK.

Cadence Script

The following Cadence script will transfer FLOW tokens from a sender to a recipient.

Note: this transaction is only compatible with Flow Mainnet.

```
cadence
// This transaction is a template for a transaction that
// could be used by anyone to send tokens to another account
// that has been set up to receive tokens.
//
// The withdraw amount and the account from getAccount
// would be the parameters to the transaction

import "FungibleToken"
import "FlowToken"

transaction(amount: UFix64, to: Address) {
    // The Vault resource that holds the tokens that are being
    transferred
    let sentVault: @{{FungibleToken.Vault}}
```

```

    prepare(signer: auth(BorrowValue) &Account) {

        // Get a reference to the signer's stored vault
        let vaultRef = signer.storage.borrow<auth(FungibleToken.Withdraw)
&FlowToken.Vault>(from: /storage/flowTokenVault)
            ?? panic("Could not borrow reference to the owner's
Vault!")

        // Withdraw tokens from the signer's stored vault
        self.sentVault <- vaultRef.withdraw(amount: amount)
    }

    execute {

        // Get a reference to the recipient's Receiver
        let receiverRef = getAccount(to)

.capabilities.borrow<&{FungibleToken.Receiver}>(/public/flowTokenReceiver
)
            ?? panic("Could not borrow receiver reference to the
recipient's Vault")

        // Deposit the withdrawn tokens in the recipient's receiver
        receiverRef.deposit(from: <-self.sentVault)
    }
}

```

Build the Transaction

```

go
import (
    "github.com/onflow/cadence"
    "github.com/onflow/flow-go-sdk"
)

// Replace with script above
const transferScript string = TOKENTRANSFERCADENCESCRIPT

var (
    senderAddress    flow.Address
    senderAccountKey flow.AccountKey
    senderPrivateKey crypto.PrivateKey
)

func main() {
    tx := flow.NewTransaction().
        SetScript([]byte(transferScript)).
        SetComputeLimit(100).
        SetPayer(senderAddress).
        SetAuthorizer(senderAddress).
        SetProposalKey(senderAddress, senderAccountKey.Index,
senderAccountKey.SequenceNumber)
}

```

```

amount, err := cadence.NewUFix64("123.4")
if err != nil {
    panic(err)
}

recipient := cadence.NewAddress(flow.HexToAddress("0xabc..."))

err = tx.AddArgument(amount)
if err != nil {
    panic(err)
}

err = tx.AddArgument(recipient)
if err != nil {
    panic(err)
}
}

```

migration-v0.25.0.md:

Migration Guide v0.25.0

The Go SDK version 0.25.0 introduced breaking changes in the API and package naming.

Changes were required to make the implementation of the new HTTP access node API available.

We will list all the changes and provide examples on how to migrate.

- Renamed package: client -> access: the client package was renamed to access which now includes both grpc package containing previously only gRPC implementation and also http package containing the new HTTP API implementation.
 - Removed package: convert: the convert package was removed and all its functions were moved to each of the corresponding grpc or http packages. The methods were also changed to not be exported, so you can no longer use them outside the convert package.
 - New clients: new clients were added each implementing the functions from the client interface and exposing a factory for creating them.
 - New Client Interface: new client interface was created which is now network agnostic, meaning it doesn't expose additional options in the API that were used to pass gRPC specific options. You can still pass those options but you must use the network specific client as shown in the example below.
- The interface also changed some functions:
- GetCollectionByID renamed to GetCollection
 - Close() error was added

Migration

Creating a Client

Creating a client for communicating with the access node has changed since it's now possible to pick and choose between HTTP and gRPC communication protocols.

Previous versions:

```
go
// initialize a gRPC emulator client
flowClient, err := client.New("127.0.0.1:3569", grpc.WithInsecure())
```

Version 0.25.0:

```
go
// common client interface
var flowClient access.Client

// initialize an http emulator client
flowClient, err := http.NewClient(http.EmulatorHost)

// initialize a gRPC emulator client
flowClient, err = grpc.NewClient(grpc.EmulatorHost)
```

Using the gRPC Client with Options

Using the client is in most cases the same except for the advance case of passing additional options to the gRPC client which is no longer possible in the base client, you must use a network specific client as shown in the advanced example:

Previous versions:

```
go
// initialize a gRPC emulator client
flowClient, err := client.New("127.0.0.1:3569", grpc.WithInsecure())
latestBlock, err := flowClient.GetLatestBlock(ctx, true,
MaxCallSendMsgSize(100))
```

Version 0.25.0:

```
go
// initialize a grpc network specific client
flowClient, err := NewBaseClient(
    grpc.EmulatorHost,
    grpc.WithTransportCredentials(insecure.NewCredentials()),
)
latestBlock, err := flowClient.GetLatestBlock(ctx, true,
MaxCallSendMsgSize(100))
```

index.md:

```
---
title: Flow Emulator
description: A development tool that looks, acts and talks like Flow
sidebarposition: 3
---
```

The Flow Emulator is a lightweight tool that emulates the behaviour of the real Flow network.

The emulator exposes a gRPC server that implements the Flow Access API, which is designed to have near feature parity with the real network API.

Running the emulator with the Flow CLI

The emulator is bundled with the Flow CLI, a command-line interface for working with Flow.

Installation

Follow these steps to install the Flow CLI on macOS, Linux, and Windows.

Usage

To learn more about using the Emulator, have a look at the README of the repository.

```
# boilerplate.md:
```

```
---
title: Cadence Boilerplate Generation
sidebarlabel: Cadence Boilerplate
description: Cadence Boilerplate Generation via the CLI
sidebarposition: 16
---
```

Introduction

Flow CLI now includes a feature to automatically generate boilerplate code for contracts, transactions, and scripts. This feature enhances the development experience by simplifying the initial setup of various components in Flow.

```
shell
> flow generate
Usage:
  flow generate [command]
```

```
Aliases:
  generate, g
```

```
Available Commands:
  contract      Generate a new contract
```

```
script      Generate a new script
transaction Generate a new transaction
```

Generate Contract

To create a new contract with basic structure, use the `contract` command. It creates a new Cadence file with a template contract definition.

```
shell
flow generate contract [ContractName]
```

Usage Example

```
shell
> flow generate contract HelloWorld
```

This command creates a file `cadence/contracts/HelloWorld.cdc` with the following content:

```
cadence
access(all) contract HelloWorld {
    init() {}
}
```

Generate Transaction

For initializing a transaction, use the `transaction` command. It sets up a new Cadence file with a template transaction structure.

```
shell
flow generate transaction [TransactionName]
```

Usage Example

```
shell
> flow generate transaction SayHello
```

This command creates a file `cadence/transactions/SayHello.cdc` with the following content:

```
cadence
transaction() {
    prepare() {}

    execute {}
}
```

Generate Script

Similarly, to start a new script, the `script` command generates a Cadence file with a basic script structure.

```
shell
flow generate script [ScriptName]
```

Usage Example

```
shell
> flow generate script ReadHello
```

This command creates a file `cadence/scripts/ReadHello.cdc` with the following content:

```
cadence
access(all) fun main() {}
```

Optional --dir Flag

The `--dir` flag is an optional feature in the Flow CLI generate commands, allowing you to specify a custom directory for the generated contract, transaction, or script files. If this flag is not provided, the CLI adheres to the recommended project setup:

- Contracts are generated in the `cadence/contracts` directory.
 - Transactions are generated in the `cadence/transactions` directory.
 - Scripts are generated in the `cadence/scripts` directory.
-
- Usage: `--dir=<directoryname>`
 - Example: `flow generate contract HelloWorld --dir=customcontracts`

Use the `--dir` flag only if your project requires a different organizational structure than the default.

```
# data-collection.md:
---
title: Data Collection
description: Data collected from Flow CLI usage
sidebarposition: 17
---
```

Flow CLI tracks `flow` command usage count using Mixpanel.

Data collection is enabled by default. Users can opt out of our data collection through running `flow settings metrics disable`. To opt back in, users can run `flow settings metrics enable`.

Why do we collect data about flow cli usage?

Collecting aggregate command count allow us to prioritise features and fixes based on how users use flow cli.

What data do we collect?

We only collect the number of times a command is executed.

We don't keep track of the values of arguments, flags used and the values of the flags used. We also don't associate any commands to any particular user.

The only property that we collect from our users are their preferences for opting in / out of data collection.

The analytics user ID is specific to Mixpanel and does not permit Flow CLI maintainers to e.g. track you across websites you visit.

Further details regarding the data collected can be found under Mixpanel's data collection page in Ingestion API section of <https://help.mixpanel.com/hc/en-us/articles/115004613766-Default-Properties-Collected-by-Mixpanel>.

Please note that although Mixpanel's page above mentions that geolocation properties are recorded by default, we have turned off geolocation data reporting to Mixpanel.

```
# dependency-manager.md:
```

```
---
```

```
title: Dependency Manager
sidebarlabel: Dependency Manager
description: Dependency Manager for the Flow Blockchain.
sidebarposition: 11
---
```

The Dependency Manager in the Flow CLI aids in speeding up and managing the development process when you use contracts from outside your project. It eliminates the manual process of copying, pasting, and updating contracts you are using or building upon. This could include core contracts or any other ecosystem contracts that you use.

For example, suppose you wanted to build a new application using the FlowToken contract. First, you would have to find the contract on the network you want to use as the source of truth and then copy it into your local project, adding it to your flow.json. You would then repeat this process for each import (dependency) it relies on (e.g., the NonFungibleToken contract) to get it working. The Dependency Manager streamlines this process with a few simple commands.

add

If you know the address and name of the contract you want to install (this can usually be found easily in the Contract Browser), you can install the dependency and all its dependencies with a single CLI command, as shown below (using FlowToken as an example):

```
flow dependencies add testnet://7e60df042a9c0868.FlowToken
```

For core contracts, you can add them to your project using a simplified syntax using only the contract name (learn more about core contracts here), for example:

```
flow dependencies add FlowToken
```

The command will default to using Flow Mainnet as the source network (i.e. this command is functionally equivalent to `flow dependencies add mainnet://1654653399040a61.FlowToken`).

> Note: You can also use the shorthand deps

In this command, the string that will be used as the source in the `flow.json` after installation is `testnet://7e60df042a9c0868.FlowToken`. This can be broken down into three sections for formatting it yourself for another contract:

- Network: `testnet`
- Address: `7e60df042a9c0868`
- Contract Name: `FlowToken`

This is the remote source of the contract on the network that will be used as the source of truth.

```
install
```

Another way to install a dependency and its dependencies is to use the `install` command. For this, you can add the dependency in the short format like:

```
{
  "dependencies": {
    "FlowToken": "emulator://0ae53cb6e3f42a79.FlowToken"
  }
}
```

Or the extended format like this:

```
{
  "dependencies": {
    "FlowToken": {
      "source": "testnet://7e60df042a9c0868.FlowToken",
      "aliases": {
        ...
      }
    }
  }
}
```

```
        "emulator": "0ae53cb6e3f42a79"
    }
}
}
}
```

Now you can run the following command from your terminal:

```
flow dependencies install
```

This will look at all the dependencies you have in your flow.json, install them, and all their dependencies.

Other Things to Note

- After installation, you will have a local folder named imports that you should add to .gitignore. This folder is where your dependencies will be stored locally.
- If your contracts change on the network, the Dependency Manager will ask if you want to update the local dependencies in your imports folder. The hash saved in the dependency object is used for this check, so don't remove it.
- Dependencies will function just like contracts. For instance, you can add them to deployments in your flow.json and run flow project deploy, as well as import them in your scripts, transactions, and contracts just as you would with a contract you added yourself (e.g., import "FlowToken").
- Core contract aliases will be automatically added for you across all networks.

discover

The discover command is used to interactively discover and install core contracts for your project. Core contracts are a standard set of smart contracts maintained by the Flow Foundation that are commonly used across the Flow ecosystem (learn more about core contracts [here](#)).

To use the discover command, run the following command in your project directory:

```
flow dependencies discover
```

You will then be presented with a list of available core contracts to install, for example:

```
shell
Select any core contracts you would like to install or skip to continue.
```

Use arrow keys to navigate, space to select, enter to confirm or skip, q to quit:

```
> [ ] FlowEpoch
[ ] FlowIDTableStaking
[ ] FlowClusterQC
[ ] FlowDKG
[ ] FlowServiceAccount
[ ] NodeVersionBeacon
[ ] RandomBeaconHistory
[ ] FlowStorageFees
[ ] FlowFees
[ ] FungibleTokenSwitchboard
[ ] EVM
```

After selecting any contracts you would like to install, you can confirm your selection by pressing enter. The selected contracts will be added to your flow.json file and accessible in your project.

```
# flix.md:
```

```
---
title: Flow Interaction Templates (FLIX)
sidebarlabel: Flow Interaction Templates (FLIX)
description: Flow Interaction Templates (FLIX) via the CLI
sidebarposition: 15
---
```

FLIX helps developers reuse existing Cadence transactions and scripts to easily integrate with existing Cadence smart contracts. Get more information about Flow Interaction Templates

Introduction

The Flow CLI provides a flix command with a few sub commands execute and package. Get familiar with Flow Interaction Templates (FLIX). FLIX are a standard for distributing Cadence scripts and transactions, and metadata in a way that is consumable by tooling and wallets. FLIX can be audited for correctness and safety by auditors in the ecosystem.

```
shell
>flow flix
execute, generate, package
```

Usage:

```
flow flix [command]
```

Available Commands:

```
execute      execute FLIX template with a given id, name, local
filename, or url
generate     generate FLIX json template given local Cadence filename
package      package file for FLIX template fcl-js is default
```

Execute

The Flow CLI provides a `flix` command to execute FLIX. The Cadence being execute in the FLIX can be a transaction or script.

```
shell
flow flix execute <query> [<argument> <argument>...] [flags]
```

:::warning

A FLIX template might only support testnet and/or mainnet. Generally, emulator is not supported. This can be the case if the FLIX template relies on contract dependencies.

:::

Queries can be a FLIX id, name, url or path to a local FLIX file.

Execute Usage

```
shell
Execute a FLIX transaction by name on Testnet
flow flix execute transfer-flow 5.0 "0x123" --network testnet --signer
"testnet-account"
```

```
shell
Execute a FLIX script by id on Testnet
flow flix execute
bd10ab0bf472e6b58ecc0398e9b3d1bd58a4205f14a7099c52c0640d9589295f --
network testnet
```

```
shell
Execute a local FLIX script by path on Testnet
flow flix execute ./multiply.template.json 2 3 --network testnet
```

The Flow CLI provides a `flix` command to package up generated plain and simple JavaScript. This JavaScript uses FCL (Flow Client Library) to call the cadence the Flow Interaction Templates (FLIX) is based on.

:::info

Currently, `flix package` command only supports generating FCL (Flow Client Library) specific JavaScript and TypeScript, there are plans to support other languages like golang.

:::

```
shell
flow flix package <query> [flags]
```

Generate

Generate FLIX json file. This command will take in a Cadence file and produce a FLIX json file. There are two ways to provide metadata to populate the FLIX json structure.

- Use --pre-fill flag to pass in a pre populated FLIX json structure
- Use --exclude-networks flag to specify excluded networks when generating a FLIX templates. Example, --exclude-networks testnet,mainnet

:::warning

When generating a FLIX template, make sure all contract dependencies have been deployed to the supported networks. Add any aliases to your flow.json that will be needed to populate dependencies. Verify all dependencies have been populated after generating.

:::

Generate Usage

```
shell
Generate FLIX json file using cadence transaction or script, this example
is not using a prefilled json file so will not have associated message
metadata
flow flix generate cadence/transactions/update-helloworld.cdc --save
cadence/templates/update-helloworld.template.json
```

Example of Cadence simple, no metadata associated
cadence

```
import "HelloWorld"
access(all) fun main(): String {
    return HelloWorld.greeting
}
```

Cadence Doc Pragma:

It's recommended to use pragma to set the metadata for the script or transaction. More information on Cadence Doc Pragma FLIP

A pragma is short for "pragmatic information", it's special instructions to convey information to a processor in this case the utility that generates FLIX.
cadence

```

import "HelloWorld"

#interaction (
    version: "1.1.0",
    title: "Update Greeting",
    description: "Update the greeting on the HelloWorld contract",
    language: "en-US",
)
transaction(greeting: String) {

    prepare(acct: &Account) {
        log(acct.address)
    }

    execute {
        HelloWorld.updateGreeting(newGreeting: greeting)
    }
}

```

:::info
 Cadence v0.42.7 supports additional Cadence pragma functionality that FLIX utility can use to generate FLIX. It will support parameters "title" and "description".
 :::

The resulting json metadata is extracted from Cadence Doc Pragma json

```
{
    "ftype": "InteractionTemplate",
    "fversion": "1.1.0",
    "id": "",
    "data": {
        "type": "transaction",
        "interface": "",
        "messages": [
            {
                "key": "title",
                "i18n": [
                    {
                        "tag": "en-US",
                        "translation": "Update Greeting"
                    }
                ]
            },
            {
                "key": "description",
                "i18n": [
                    {
                        "tag": "en-US",

```

```

        "translation": "Update the greeting on the
HelloWorld contract"
    }
]
}
],
"cadence": {},
"dependencies": [],
"parameters": [
{
    "label": "greeting",
    "index": 0,
    "type": "String",
    "messages": []
}
]
}
}
}

```

Example of using a pre-filled FLIX json file. No need to use Cadence pragma when using a pre-filled FLIX json file. This method separates FLIX specific information from the transaction or script Cadence. Use the flow flix generate command:

```

shell
flow flix generate cadence/scripts/read-helloworld.cdc --pre-fill
cadence/templates/read-helloworld.prefill.json --save
cadence/templates/read-helloworld.template.json

```

Using a pre-filled FLIX template, the cadence can be simple but no metadata accompanies it.

```

cadence
import "HelloWorld"
access(all) fun main(): String {
    return HelloWorld.greeting
}

```

Example of json prefill file with message metadata:

```

json
{
    "ftype": "InteractionTemplate",
    "fversion": "1.1.0",
    "id": "",
    "data": {
        "type": "script",
        "interface": "",
        "messages": [
            {
                "key": "title",
                "i18n": [

```

```

        {
            "tag": "en-US",
            "translation": "Get Greeting"
        }
    ]
},
{
    "key": "description",
    "i18n": [
        {
            "tag": "en-US",
            "translation": "Call HelloWorld contract to get
greeting"
        }
    ]
}
]
}
}

```

The resulting FLIX json file after generation:

```

json
{
    "ftype": "InteractionTemplate",
    "fversion": "1.1.0",
    "id": "fd9abd34f51741401473eb1cf676b105fed28b50b86220a1619e50d4f80b0be1",
    "data": {
        "type": "script",
        "interface": "",
        "messages": [
            {
                "key": "title",
                "i18n": [
                    {
                        "tag": "en-US",
                        "translation": "Get Greeting"
                    }
                ]
            },
            {
                "key": "description",
                "i18n": [
                    {
                        "tag": "en-US",
                        "translation": "Call HelloWorld contract to get
greeting"
                    }
                ]
            }
        ],
    }
},
]
```

```

"cadence": {
    "body": "import \"HelloWorld\"\naccess(all) fun main(): String {\n    return HelloWorld.greeting\n}\n",
    "networkpins": [
        {
            "network": "testnet",
            "pinself": "41c4c25562d467c534dc92baba92e0c9ab207628731ee4eb4e883425abda692c"
        }
    ],
    "dependencies": [
        {
            "contracts": [
                {
                    "contract": "HelloWorld",
                    "networks": [
                        {
                            "network": "testnet",
                            "address": "0xe15193734357cf5c",
                            "dependencypinblockheight": 137864533,
                            "dependencypin": {
                                "pin": "aad46badcab3caaeb4f0435625f43e15bb4c15b1d55c74a89e6f04850c745858",
                                "pinself": "a06b3cd29330a3c22df3ac2383653e89c249c5e773fd4bbe73c45ea10294b97",
                                "pincontractname": "HelloWorld",
                                "pincontractaddress": "0xe15193734357cf5c",
                                "imports": []
                            }
                        }
                    ]
                }
            ]
        },
        "parameters": null
    }
}

```

Package

Queries can be a FLIX url or path to a local FLIX file. This command leverages FCL which will execute FLIX cadence code. Package files can be generated in JavaScript or TypeScript.

:::warning

Currently package doesn't support id, name flix query.

```
:::
```

Package Usage

```
shell
Generate packaged code that leverages FCL to call the Cadence transaction
code, --save flag will save the output to a specific file
flow flix package transfer-flow --save ./package/transfer-flow.js
```

```
shell
Generate package code for a FLIX script using id, since there is no
saving file, the result will display in terminal
flow flix package
bd10ab0bf472e6b58ecc0398e9b3d1bd58a4205f14a7099c52c0640d9589295f
```

```
shell
Generate package code using local template file to save in a local file
flow flix package ./multiply.template.json --save ./multiply.js
```

```
shell
Generate package code using local template file to save in a local
typescript file
flow flix package ./multiply.template.json --lang ts --save ./multiply.ts
```

Example Package Output

```
shell
flow flix package https://flix.flow.com/v1/templates\?name\=transfer-flow
```

javascript

```
/ 
  This binding file was auto generated based on FLIX template v1.0.0.
  Changes to this file might get overwritten.
  Note fcl version 1.3.0 or higher is required to use templates.
/
```

```
import as fcl from "@onflow/fcl"
const flixTemplate = "https://flix.flow.com/v1/templates?name=transfer-
flow"
```

```
/
Transfer tokens from one account to another
@param {Object} Parameters - parameters for the cadence
@param {string} Parameters.amount - The amount of FLOW tokens to send:
UFix64
@param {string} Parameters.to - The Flow account the tokens will go to:
Address
```

```
@returns {Promise<string>} - returns a promise which resolves to the
transaction id
/
export async function transferTokens({amount, to}) {
  const transactionId = await fcl.mutate({
    template: flixTemplate,
    args: (arg, t) => [arg(amount, t.UFix64), arg(to, t.Address)]
  });

  return transactionId
}

shell
Generate TypeScript version of package file
flow flix package https://flix.flow.com/v1/templates?name=transfer-flow -
-lang ts
```

```
typescript
/
  This binding file was auto generated based on FLIX template v1.1.0.
  Changes to this file might get overwritten.
  Note fcl version 1.9.0 or higher is required to use templates.
/

import as fcl from "@onflow/fcl"
const flixTemplate = "https://flix.flow.com/v1/templates?name=transfer-
flow"

interface TransferTokensParams {
  amount: string; // The amount of FLOW tokens to send
  to: string; // The Flow account the tokens will go to
}

/
  transferTokens: Transfer tokens from one account to another
  @param string amount - The amount of FLOW tokens to send
  @param string to - The Flow account the tokens will go to
  @returns {Promise<string>} - Returns a promise that resolves to the
transaction ID
/
export async function transferTokens({amount, to}: TransferTokensParams): Promise<string> {
  const transactionId = await fcl.mutate({
    template: flixTemplate,
    args: (arg, t) => [arg(amount, t.UFix64), arg(to, t.Address)]
  });

  return transactionId
}
```

```
::::warning

Notice that fcl v1.9.0 is needed to use FLIX v1.1 templates

:::

Resources

To find out more about FLIX, see the read the FLIP.

For a list of all templates, check out the FLIX template repository.

To generate a FLIX, see the FLIX CLI readme.

Arguments
- Name: argument
- Valid input: valid FLIX

Input argument value matching corresponding types in the source code and
passed in the same order.
You can pass a nil value to optional arguments by executing the flow FLIX
execute script like this: flow flix execute template.json nil.

Flags

Arguments JSON

- Flag: --args-json
- Valid inputs: arguments in JSON-Cadence form.
- Example: flow flix execute template.script.json '[{"type": "String",
"value": "Hello World"}]'

Arguments passed to the Cadence script in the Cadence JSON format.
Cadence JSON format contains type and value keys and is
documented here.

Pre Fill

- Flag: --pre-fill
- Valid inputs: a json file in the FLIX json structure FLIX json format

Block Height

- Flag: --block-height
- Valid inputs: a block height number

Block ID

- Flag: --block-id
- Valid inputs: a block ID
```

Signer

- Flag: `--signer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used to sign the transaction.

Proposer

- Flag: `--proposer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used as proposer in the transaction.

Payer

- Flag: `--payer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used as payer in the transaction.

Authorizer

- Flag: `--authorizer`
- Valid inputs: the name of a single or multiple comma-separated accounts defined in the configuration (`flow.json`)

Specify the name of the account(s) that will be used as authorizer(s) in the transaction. If you want to provide multiple authorizers separate them using commas (e.g. `alice,bob`)

Gas Limit

- Flag: `--gas-limit`
- Valid inputs: an integer greater than zero.
- Default: 1000

Specify the gas limit for this transaction.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: `127.0.0.1:3569` (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge
several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow
connections.

index.md:

```
---
sidebarposition: 2
title: Flow CLI
sidebarlabel: Flow CLI
---
```

The Flow Command Line Interface (CLI) is a powerful tool that enables developers to seamlessly interact with the Flow blockchain across various environments, including testnet, mainnet, and local development using the Flow Emulator. Designed for ease of use, the Flow CLI simplifies common blockchain tasks such as managing accounts and contract dependencies, sending transactions, querying chain state, deploying smart contracts, and much more.

With Flow CLI, developers can:

- Initialize Projects: Quickly set up new Flow projects using the `flow init` command, which creates the necessary files and directories, sets up your project configuration, and installs any core contract dependencies.
- Manage Contract Dependencies: Use the Dependency Manager to install and manage smart contract dependencies effortlessly, simplifying the integration of external contracts into your project.
- Manage Accounts: Create and manage Flow accounts, configure keys, and handle account-related operations.
- Send Transactions: Build, sign, and submit transactions to the Flow network, allowing for contract interaction and fund transfers.
- Query Chain State: Retrieve data from the Flow blockchain, including account balances, event logs, and the status of specific transactions.
- Deploy Smart Contracts: Easily deploy and update Cadence smart contracts on any Flow environment (emulator, testnet, or mainnet).

- Use the Emulator: Set up a local Flow blockchain instance with the Flow emulator to test and debug smart contracts in a development environment before deploying them on the network.
- Interact with the Flow Access API: Automate complex workflows using configuration files and command-line scripting, which allows for greater flexibility in continuous integration (CI) or custom development tools.
- Access Flow's Tooling Ecosystem: Integrate Flow CLI with other developer tools like the Cadence Extension for VSCode to enhance your development experience.

The Flow CLI is essential for developers looking to build, test, and maintain decentralized applications on the Flow blockchain efficiently, offering a feature-rich, user-friendly interface for both beginners and experienced blockchain developers.

Installation

Follow these steps to install the Flow CLI on macOS, Linux, and Windows.

Create Your First Project

To get started with creating your first Flow project and to learn more about how to use the Flow CLI super commands, please refer to the Super Commands documentation. These commands simplify the setup and development process, allowing you to focus on building your application without worrying about the underlying configurations.

```
# install.md:
```

```
---
```

```
title: Install Instructions
description: How to install the Flow command-line interface (CLI)
sidebarposition: 1
---
```

The Flow CLI can be installed on macOS, Windows (7 or greater) and most Linux systems.

> Note: If you need to install the pre-release version of the Flow CLI supporting Cadence 1.0, please refer to the Cadence 1.0 migration guide instructions.

macOS

Homebrew

```
sh
brew install flow-cli
```

From a pre-built binary

This installation method only works on x86-64.

This script downloads and installs the appropriate binary for your system:

```
sh  
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.sh)"
```

To update, simply re-run the installation command above.

It is currently not possible to install earlier versions of the Flow CLI with Homebrew.

Linux

From a pre-built binary

This installation method only works on x86-64.

This script downloads and installs the appropriate binary for your system:

```
sh  
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.sh)"
```

To update, simply re-run the installation command above.

Install a specific version

To install a specific version of Flow CLI newer than v0.42.0, append the version tag to the command (e.g. the command below installs CLI version v0.44.0).

```
sh  
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.sh)" -- v0.44.0
```

To install a version older than v0.42.0, refer to [Installing versions before 0.42.0](#) below.

Windows

From a pre-built binary

This installation method only works on Windows 10, 8.1, or 7 (SP1, with PowerShell 3.0), on x86-64.

1. Open PowerShell ([Instructions](#))
2. In PowerShell, run:

```
powershell  
    iex "& { $(irm 'https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.ps1') }"
```

To update, simply re-run the installation command above.

Upgrade the Flow CLI

macOS

Homebrew

```
sh  
brew upgrade flow-cli
```

From a pre-built binary

This update method only works on x86-64.

This script downloads and updates the appropriate binary for your system:

```
sh  
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.sh)"
```

Linux

From a pre-built binary

This update method only works on x86-64.

This script downloads and updates the appropriate binary for your system:

```
sh  
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.sh)"
```

Windows

From a pre-built binary

This update method only works on Windows 10, 8.1, or 7 (SP1, with PowerShell 3.0), on x86-64.

1. Open PowerShell (Instructions)
2. In PowerShell, run:

```
powershell  
    iex "& { $(irm 'https://raw.githubusercontent.com/onflow/flow-  
cli/master/install.ps1') }"
```

Uninstalling Flow CLI

To remove the flow CLI you can run the following command if it was previously installed using a pre-built binary.

- macOS: rm /usr/local/bin/flow
- Linux: rm /.local/bin/flow
- Windows: rm /Users/{user}/AppData/Flow/flow.exe

If you installed it using Homebrew you can remove it using: brew uninstall flow-cli.

Installing versions before 0.42.0

If you want to install versions before v0.42.0 you have to use a different install command.

Linux/macOS

```
https://raw.githubusercontent.com/onflow/flow-cli/v0.41.3/install.ps1
```

```
sh -ci "$(curl -fsSL https://raw.githubusercontent.com/onflow/flow-
cli/v0.41.3/install.sh)" -- v0.41.2
```

Windows

```
iex "& { $(irm 'https://raw.githubusercontent.com/onflow/flow-
cli/master/install.ps1') }"
```

```
# lint.md:
```

```
---
title: Cadence Linter
description: A static-analysis tool for finding potential issues in
Cadence code
sidebarposition: 14
---
```

The Cadence Linter is a static-analysis tool for finding potential issues in Cadence code. It is available in the Flow CLI & is designed to help developers write better code by identifying common mistakes and potential issues before they become problems.

The linter will also check your code for any syntax or semantic errors, and provide suggestions for how to fix them.

```
shell
flow cadence lint [files]
```

Example Usage

```
shell
flow cadence lint /.cdc
```

Example Output

```
shell
test.cdc:27:6: semantic-error: cannot find variable in this scope: abc
test.cdc:35:6: removal-hint: unnecessary force operator
2 problems (1 error, 1 warning)
```

```
:::info
The Cadence Linter is also available in the Cadence VSCode extension,
which provides real-time feedback as you write your code.
```

```
:::
```

```
# super-commands.md:
```

```
---
title: Super Commands
description: How Flow Super Commands Work
sidebarposition: 2
---
```

Flow CLI Super commands are set of commands that can be used during development of your dApp to greatly simplify the workflow. The result is you can focus on writing the contracts and the commands will take care of the rest.

Init

The initial command to start your new Flow project is `flow init`. It will ask you a few questions about how you'd like to configure your project and then create the necessary files and folders, set up the configuration file, and install any core contract dependencies you might need.

During the initialization process, `flow init` will prompt you if you want to install any core smart contracts (e.g. `NonFungibleToken`) and set them up in your project. If you choose to install core contracts, the CLI will use the Dependency Manager under the hood to automatically install any required smart contract dependencies.

> Note: If you just want the `flow.json` configured without creating any folders or files, you can run `flow init --config-only`.

Running the command:

```
> flow init $PROJECTNAME
```

Will create the following folders and files:

- /contracts folder should contain all your Cadence contracts,
- /scripts folder should contain all your Cadence scripts,
- /transactions folder should contain all your Cadence transactions,
- /tests folder should contain all your Cadence tests,
- flow.json is a configuration file for your project, which will be automatically maintained.

Using Scaffolds

Based on the purpose of your project you can select from a list of available scaffolds.

You can access the scaffolds by simply using the --scaffold flag like so:

```
> flow init $PROJECTNAME --scaffold
```

If you'd like to skip the interactive mode of selecting a scaffold, use the --scaffold-id flag with a known ID:

```
> flow init $PROJECTNAME --scaffold-id=1
```

The list of scaffolds will continuously grow, and you are welcome to contribute to that.

You can contribute by creating your own scaffold repository which can then be added to the scaffold list by following instructions here.

Testing

flow init will also have created an example test file in the /tests folder. You can run the tests by using the flow test command.

Develop

After creating the project using the flow init command you can start the emulator in the project directory by running flow emulator. After emulator is started up you can continue by running the flow develop command like so:

```
> flow dev
```

This will continuously watch for your projects Cadence files for changes and keep them in sync with the deployed contracts on the emulator.

The output will look something like:

```
[15:53:38] Syncing all the contracts...
```

```
⌚ charlie
|- MyContract contracts/charlie/MyContract.cdc

⌚ emulator-account
|- HelloWorld contracts/HelloWorld.cdc
```

After the command is started it will automatically watch any changes you make to Cadence files and make sure to continuously sync those changes on the emulator network.

If you make any mistakes it will report the errors as well.

It is recommended that you use VSCode as the IDE and run the command in the terminal window of the IDE.

The latest VSCode extension also supports resolution of the improved import syntax, more on that later.

⚠ Please note that this command only works on the emulator network. It's meant for development only and hence it doesn't allow interacting with testnet or mainnet network. After your project is completed you will be soon able to migrate it using a migration super command. Also, please note the command requires a running emulator which you have to start. If you restart the emulator the command needs to be restarted as well.

This command is meant to be used during development, and it updates the contracts by removing and redeploying them, which means that if you manually interacted with those contracts and stored resources in accounts storage that stored items might no longer be valid after contract is updated. Our advise is to first focus on development and use automated tests to assert correct functionality and interact with contracts manually after this cycle is complete. Also note that this is still a very experimental feature, so it might undergo a lot of changes and improvements as we learn from the usage.

Deploying Contracts

When adding the contracts to the /contracts folder it will automatically deploy them to the default account, which is also created for you at startup of running flow dev.

If you want to add the contracts to a separate account all you have to do is create a folder inside the /contracts folder and add the contract there, that will first automatically create the account with the same name as the folder name and then deploy all the contracts inside that folder to that newly created account.

Example:

If I want to have a contract named A.cdc deployed to a default account and a contract named B.cdc deployed to account called Bob my folder structure inside contracts folder will look like:

```
/contracts
  A.cdc
```

```
bob/  
B.cdc
```

Import Schema

You can simply import your contracts by name. We have introduced a new way to import your contracts. This will simplify your workflow.

The new import schema format looks like:

```
import "{name of the contract}"
```

Example:

```
import "HelloWorld"
```

This will automatically import the contract you have created in your project with the same name and save the configuration in flow.json. It doesn't matter if the contract has been deployed on a non-default account.

[Learn More](#)

To learn more about next steps following the initial setup, check out the following links:

- Dependency Manager: Lets you install and manage your contract dependencies with CLI commands.
- Manage Configuration: Learn how to manage your project configuration file.

```
# _template.md:  
  
---  
title: -title-  
sidebarlabel:  
description: -description-  
---  
  
\{short description\}  
  
shell  
{command}  
  
\{optional warning\}
```

[Example Usage](#)

```
shell  
{usage example with response}
```

[Arguments](#)

```
\{Argument 1\}

- Name: \{argument\}
- Valid Input: \{input\}

\{argument general description\}
```

Arguments

Address

- Name: address
- Valid Input: Flow account address

Flow account address (prefixed with 0x or not).

Flags

```
\{Option 1\}
```

- Flag: \{flag value\}
- Valid inputs: \{input description\}

```
\{flag general description\}
```

Signer

- Flag: --signer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used to sign the transaction.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the commands.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify in which format you want to display the result.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: valid filename

Specify the filename where you want the result to be saved.

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see while command execution.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using -f flag multiple times.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# account-add-contract.md:
```

```
---
```

```
title: Deploy a Contract
```

```
sidebarposition: 3
```

```
---
```

Deploy a new contract to a Flow account using the Flow CLI.

```
shell
```

```
flow accounts add-contract <filename> [<argument> <argument>...] [flags]
```

⚠️ Deprecation notice: using name argument in adding contract command will be deprecated soon.

```
shell
```

```
flow accounts add-contract <name> <filename> [<argument> <argument>...] [flags]
```

Example Usage

```
shell
```

```
> flow accounts add-contract ./FungibleToken.cdc
```

```
Contract 'FungibleToken' deployed to the account 0xf8d6e0586b0a20c7
```

```
Address      0xf8d6e0586b0a20c7
```

```
Balance      9999999999.70000000
```

```
Keys        1
```

```
Key 0 Public Key
```

```
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
```

```
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
```

```
    Weight          1000
```

```
    Signature Algorithm  ECDSAP256
```

```
    Hash Algorithm     SHA3256
```

```
    Revoked           false
```

```
    Sequence Number    6
```

```
    Index              0
```

```
Contracts Deployed: 1
```

```
Contract: 'FungibleToken'
```

Testnet Example

```
> flow accounts add-contract ./FungibleToken.cdc --signer alice --network testnet

Contract 'FungibleToken' deployed to the account 0xf8d6e0586b0a20c7

Address      0xf8d6e0586b0a20c7
Balance      99999999999.70000000
Keys         1

Key 0 Public Key
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
  Weight          1000
  Signature Algorithm   ECDSAP256
  Hash Algorithm       SHA3256
  Revoked            false
  Sequence Number     6
  Index               0

Contracts Deployed: 1
Contract: 'FungibleToken'
```

Make sure alice account is defined in flow.json

Arguments

Name

- Name: name
- Valid inputs: any string value.

Name of the contract as it is defined in the contract source code.

⚠️ Deprecation notice: use filename argument only, no need to use name argument.

Filename

- Name: filename
- Valid inputs: a path in the current filesystem.

Path to the file containing the contract source code.

Arguments

- Name: argument
- Valid inputs: valid cadence values matching argument type in transaction code.

Input arguments values matching corresponding types in the source code and passed in the same order.

Example:

shell

```
> flow accounts add-contract ./contract.cdc Hello 2
```

Transaction code:

```
access(all) contract HelloWorld {  
    init(a:String, b:Int) {  
    }  
}
```

Flags

Signer

- Flag: `--signer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used to sign the transaction.

Arguments JSON

- Flag: `--args-json`
- Valid inputs: arguments in JSON-Cadence form.
- Example: `flow accounts add-contract ./tx.cdc '[{"type": "String", "value": "Hello"}]'`

Arguments passed to the Cadence transaction in Cadence JSON format. Cadence JSON format contains type and value keys and is documented here.

Include Fields

- Flag: `--include`
- Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`

- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json).
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: --config-path

- Short Flag: -f
- Valid inputs: a path in the current filesystem.
- Default: flow.json

Specify the path to the flow.json configuration file.
You can use the -f flag multiple times to merge
several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow
connections.

account-fund.md:

```
---
title: Funding a Testnet Account
description: How to fund a Testnet Flow account from the command line
sidebarposition: 7
---
```

:::info

The Flow Testnet Faucet allows users to create accounts and receive 1,000
Testnet FLOW tokens for testing and development purposes. You can also
fund an existing Testnet accounts without needing to create one through
the site, or through the CLI.

:::

Fund a valid Testnet Flow Account using the Flow CLI.

```
shell
flow accounts fund <address>
```

Example Usage

```
> flow accounts fund 8e94eaa81771313a
```

Opening the faucet to fund 0x8e94eaa81771313a on your native browser.

If there is an issue, please use this link instead: <https://testnet-faucet.onflow.org/fund-account?address=8e94eaa81771313a>

Arguments

Address

- Name: address
- Valid Input: Flow Testnet account address.

Flow account address (prefixed with 0x or not).

account-remove-contract.md:

```
---
```

title: Remove a Contract
sidebarposition: 5

This feature is only found in the Emulator. You cannot remove a contract
on Testnet or Mainnet.

Remove an existing contract deployed to a Flow account using the Flow
CLI.

```
shell  
> flow accounts remove-contract <name>
```

Example Usage

```
shell  
> flow accounts remove-contract FungibleToken  
  
Contract 'FungibleToken' removed from account '0xf8d6e0586b0a20c7'  
  
Address      0xf8d6e0586b0a20c7  
Balance      9999999999.70000000  
Keys         1  
  
Key 0 Public Key  
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d  
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7  
  Weight          1000  
  Signature Algorithm   ECDSAP256  
  Hash Algorithm        SHA3256  
  Revoked            false  
  Sequence Number     6  
  Index               0  
  
Contracts Deployed: 0
```

Testnet Example

```
> flow accounts remove-contract FungibleToken --signer alice --network  
testnet  
  
Contract 'FungibleToken' removed from account '0xf8d6e0586b0a20c7'
```

```
Address      0xf8d6e0586b0a20c7
Balance      99999999999.70000000
Keys         1

Key 0 Public Key
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
    Weight          1000
    Signature Algorithm ECDSAP256
    Hash Algorithm     SHA3256
    Revoked           false
    Sequence Number   6
    Index              0
```

Contracts Deployed: 0

Make sure alice account is defined in flow.json

Arguments

Name

- Name: name
- Valid inputs: any string value.

Name of the contract as it is defined in the contract source code.

Flags

Signer

- Flag: --signer
- Valid inputs: the name of an account defined in the configuration (flow.json).

Specify the name of the account that will be used to sign the transaction.

Include Fields

- Flag: --include
- Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
 - Short Flag: `-f`
 - Valid inputs: a path in the current filesystem
 - Default: `flow.json`

Specify the path to the flow.json configuration file.
You can use the -f flag multiple times to merge
several configuration files.

Version Check

- Flag: `--skip-version-check`
 - Default: `false`

Skip version check during start up to speed up process for slow connections.

account-staking-info.md:

```
---
```

title: Account Staking Info
description: How to get staking info
sidebarposition: 6

```
---
```

Retrieve staking information for the account on the Flow network using Flow CLI.

```
shell
flow accounts staking-info <address>
```

Example Usage

Account Delegation Info:

ID:	7
Tokens Committed:	0.00000000
Tokens To Unstake:	0.00000000
Tokens Rewarded:	30397.81936000
Tokens Staked:	100000.00000000
Tokens Unstaked:	0.00000000
Tokens Unstaking:	0.00000000

Arguments

Address

- Name: address
 - Valid Input: Flow account address.

Flow account address (prefixed with 0x or not).

Flags

Include Fields

- Flag: --include
 - Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: `--host`
 - Valid inputs: an IP address or hostname.
 - Default: `127.0.0.1:3569` (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge
several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow
connections.

account-update-contract.md:

```
---
```

`title: Update a Contract`
`sidebarposition: 4`

```
---
```

Update an existing contract deployed to a Flow account using the Flow
CLI.

```
shell  
flow accounts update-contract <filename> [<argument> <argument>...]  
[flags]
```

⚠️ Deprecation notice: using `name` argument in `update contract` command
will be deprecated soon.

```
shell  
flow accounts update-contract <name> <filename> [<argument>  
<argument>...] [flags]
```

Example Usage

```
shell  
> flow accounts update-contract ./FungibleToken.cdc  
  
Contract 'FungibleToken' updated on account '0xf8d6e0586b0a20c7'  
  
Address      0xf8d6e0586b0a20c7  
Balance      99999999999.70000000  
Keys        1
```

```
Key 0 Public Key
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
  Weight          1000
  Signature Algorithm   ECDSAP256
  Hash Algorithm        SHA3256
  Revoked            false
  Sequence Number      6
  Index                0
```

```
Contracts Deployed: 1
Contract: 'FungibleToken'
```

Testnet Example

```
> flow accounts update-contract ./FungibleToken.cdc --signer alice --
network testnet
```

```
Contract 'FungibleToken' updated on account '0xf8d6e0586b0a20c7'
```

```
Address      0xf8d6e0586b0a20c7
Balance      99999999999.70000000
Keys         1
```

```
Key 0 Public Key
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
  Weight          1000
  Signature Algorithm   ECDSAP256
  Hash Algorithm        SHA3256
  Revoked            false
  Sequence Number      6
  Index                0
```

```
Contracts Deployed: 1
Contract: 'FungibleToken'
```

Make sure alice account is defined in flow.json

Arguments

Name
- Name: name
- Valid inputs: Any string value

Name of the contract as it is defined in the contract source code.

⚠ Deprecation notice: use filename argument only, no need to use name argument.

Filename
- Name: filename
- Valid inputs: Any filename and path valid on the system.

Filename of the file containing contract source code.

Arguments

- Name: argument
- Valid inputs: valid cadence values matching argument type in transaction code.

Input arguments values matching corresponding types in the source code and passed in the same order.

Example:

```
shell  
> flow accounts update-contract ./contract.cdc Hello 2
```

Transaction code:

```
access(all) contract HelloWorld {  
    init(a:String, b:Int) {  
    }  
}
```

Flags

Signer

- Flag: --signer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used to sign the transaction.

Show Diff

- Flag: --show-diff
- Valid inputs: true, false

Shows a diff to approve before updating between deployed contract and new contract updates.

Arguments JSON

- Flag: --args-json
- Valid inputs: arguments in JSON-Cadence form.
- Example: flow accounts update-contract ./tx.cdc '[{"type": "String", "value": "Hello"}]'

Arguments passed to the Cadence transaction in Cadence JSON format. Cadence JSON format contains type and value keys and is documented here.

Include Fields

- Flag: --include
- Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none, error, debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

`create-accounts.md:`

```
---
```

`title: Create an Account`
`description: How to create a Flow account from the command line`
`sidebarposition: 2`

The Flow CLI provides a command to submit an account creation transaction to any Flow Access API. There are two options how to create an account, you can use the interactive mode which guides you through the process and creates the account for you or by using the manual process which requires a pre-existing account on the network you chose.

Interactive Mode

Creating the account in interactive mode prompts you for an account name and network selection.

After you enter the required information the account will be created for you and saved to flow.json.
If account creation is done on testnet or mainnet the account key will be saved to a separate key file,
which will also be put in .gitignore. You can read more about key security here.

❗ Please note that the account creation process can take up to a minute so please be patient.

```
shell
flow accounts create

Enter an account name: mike
✓ Testnet
```

⚡ New account created with address 0x77e6ae4c8c2f1dd6 and name mike on Testnet network.

Here's a summary of all the actions that were taken:

- Added the new account to flow.json.
- Saved the private key to mike.pkey.
- Added mike.pkey to .gitignore.

Manual Mode

Manual mode requires you to have a pre-existing account on the network which you will have to provide as a signer.

That account must be added to flow.json for the command to work. You also have to generate a key pair, we suggest using the flow keys generate command, which you can read more about here.

```
shell
Create an account on Flow Testnet
> flow accounts create \
  --key a69c6986e846ba6d0....1397f5904cd319c3e01e96375d5777f1a47010 \
  --signer my-testnet-account

Address      0x01cf0e2f2f715450
Balance      10000000
Keys         1

Key 0 Public Key
a69c6986e846ba6d0....1397f5904cd319c3e01e96375d5777f1a47010
  Weight          1000
  Signature Algorithm   ECDSAP256
  Hash Algorithm     SHA3256

Contracts Deployed: 0
```

In the above example, the flow.json file would look something like this:

```
json
{
  "accounts": {
    "my-testnet-account": {
      "address": "a2c4941b5f3c7151",
      "key": "12c5dfde...bb2e542f1af710bd1d40b2"
    }
  }
}
```

Flags

Public Key

- Flag: `--key`
- Valid inputs: a hex-encoded public key in raw form.

Specify the public key that will be added to the new account upon creation.

Key Weight

- Flag: `--key-weight`
- Valid inputs: number between 0 and 1000
- Default: 1000

Specify the weight of the public key being added to the new account.

When opting to use this flag, you must specify a `--key-weight` flag for each public `--key` flag provided.

Public Key Signature Algorithm

- Flag: `--sig-algo`
- Valid inputs: "ECDSAP256", "ECDSAsecp256k1"
- Default: "ECDSAP256"

Specify the ECDSA signature algorithm for the provided public key. This option can only be used together with the `--key` flag.

Flow supports the secp256k1 and P-256 curves.

Public Key Hash Algorithm

- Flag: `--hash-algo`
- Valid inputs: "SHA2256", "SHA3256"
- Default: "SHA3256"

Specify the hash algorithm that will be paired with the public key upon account creation.

Signer

- Flag: `--signer`
- Valid inputs: the name of an account defined in `flow.json`.

Specify the name of the account that will be used to sign the transaction and pay the account creation fee.

Contract

- Flag: `--contract`
- Valid inputs: String with format `name:filename`, where `name` is name of the contract as it is defined in the contract source code and `filename` is the filename of the contract source code.

Specify one or more contracts to be deployed during account creation.

Include Fields

- Flag: `--include`
- Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: `127.0.0.1:3569` (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# get-accounts.md:
```

```
---
title: Get an Account
description: How to get a Flow account from the command line
sidebarposition: 1
---
```

The Flow CLI provides a command to fetch any account by its address from the Flow network.

```
shell
flow accounts get <address>
```

Example Usage

```
shell
flow accounts get 0xf8d6e0586b0a20c7
```

Example response

```
shell
Address      0xf8d6e0586b0a20c7
Balance      9999999999.70000000
Keys        1

Key 0 Public Key
640a5a359bf3536d15192f18d872d57c98a96cb871b92b70cecb0739c2d5c37b4be12548d
3526933c2cda9b0b9c69412f45ffb6b85b6840d8569d969fe84e5b7
    Weight          1000
    Signature Algorithm   ECDSAP256
    Hash Algorithm       SHA3256
    Revoked            false
    Sequence Number     6
    Index                0

Contracts Deployed: 2
Contract: 'FlowServiceAccount'
Contract: 'FlowStorageFees'
```

Arguments

Address

- Name: address
- Valid Input: Flow account address

Flow account address (prefixed with 0x or not).

Flags

Include Fields

- Flag: `--include`
- Valid inputs: contracts

Specify fields to include in the result output. Applies only to the text output.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# deploy-project-contracts.md:
```

```
---
```

```
title: Deploy a Project
description: How to deploy Flow project contracts with the CLI
sidebarposition: 3
---
```

```
shell
flow project deploy
```

This command automatically deploys your project's contracts based on the configuration defined in your `flow.json` file.

Before using this command, read about how to

configure project contracts and deployment targets.

Example Usage

```
shell
> flow project deploy --network=testnet

Deploying 2 contracts for accounts: my-testnet-account

NonFungibleToken -> 0x8910590293346ec4
KittyItems -> 0x8910590293346ec4

★ All contracts deployed successfully
```

In the example above, your `flow.json` file might look something like this:

```
json
{
  ...
  "contracts": {
    "NonFungibleToken": "./cadence/contracts/NonFungibleToken.cdc",
    "KittyItems": "./cadence/contracts/KittyItems.cdc"
  },
  "deployments": {
    "testnet": {
      "my-testnet-account": ["KittyItems", "NonFungibleToken"]
    }
  },
  ...
}
```

Here's a sketch of the contract source files:

```
cadence NonFungibleToken.cdc
access(all) contract NonFungibleToken {
  // ...
}

cadence KittyItems.cdc
import NonFungibleToken from "./NonFungibleToken.cdc"

access(all) contract KittyItems {
  // ...
}
```

Initialization Arguments

Deploying contracts that take initialization arguments can be achieved with adding those arguments to the configuration.

Each deployment can be specified as an object containing

name and args key specifying arguments to be used during the deployment. Example:

```
...
  "deployments": {
    "testnet": {
      "my-testnet-account": [
        "NonFungibleToken", {
          "name": "Foo",
          "args": [
            { "type": "String", "value": "Hello World" },
            { "type": "UInt32", "value": "10" }
          ]
        } ]
      }
    }
  ...
}
```

⚠ Warning: before proceeding, we recommend reading the Flow CLI security guidelines to learn about the best practices for private key storage.

Dependency Resolution

The deploy command attempts to resolve the import statements in all contracts being deployed.

After the dependencies are found, the CLI will deploy the contracts in a deterministic order

such that no contract is deployed until all of its dependencies are deployed.

The command will return an error if no such ordering exists due to one or more cyclic dependencies.

In the example above, Foo will always be deployed before Bar.

Address Replacement

After resolving all dependencies, the deploy command rewrites each contract so that its dependencies are imported from their target addresses rather than their source file location.

The rewritten versions are then deployed to their respective targets, leaving the original contract files unchanged.

In the example above, the KittyItems contract would be rewritten like this:

```
cadence KittyItems.cdc
```

```
import NonFungibleToken from 0xf8d6e0586b0a20c7

access(all) contract KittyItems {
    // ...
}
```

Merging Multiple Configuration Files

You can use the `-f` flag multiple times to merge several configuration files.

If there is an overlap in any of the fields in the configuration between two or more configuration files, the value of the overlapped field in the resulting configuration will come from the configuration file that is on the further right order in the list of configuration files specified in the `-f` flag

Let's look at an example of deploy commands with multiple configuration files below

```
cadence flow.json
{
    "accounts": {
        "admin-account": {
            "address": "f8d6e0586b0a20c7",
            "key": "21c5dfdeb0ff03a7a73ef39788563b62c89adea67bbb21ab95e5f710bd1d40b7"
        },
        "test-account": {
            "address": "f8d6e0586b0a20c8",
            "key": "52d5dfdeb0ff03a7a73ef39788563b62c89adea67bbb21ab95e5f710bd1d51c9"
        }
    }
}

cadence private.json
{
    "accounts": {
        "admin-account": {
            "address": "f1d6e0586b0a20c7",
            "key": "3335dfdeb0ff03a7a73ef39788563b62c89adea67bbb21ab95e5f710bd1d40b7"
        }
    }
}
```

In the example above, when we try to use the `deploy` command with multiple configuration files and there is an overlap in the `admin-account` account in `accounts` field of the configuration, the resulting configuration will be like this

```
> flow project deploy -f flow.json -f private.json
```

```
{
  "accounts": {
    "admin-account": {
      "address": "f1d6e0586b0a20c7",
      "key": "3335dfdeb0ff03a7a73ef39788563b62c89adea67bbb21ab95e5f710bd1d40b7"
    },
    "test-account": {
      "address": "f8d6e0586b0a20c8",
      "key": "52d5dfdeb0ff03a7a73ef39788563b62c89adea67bbb21ab95e5f710bd1d51c9"
    }
  }
}
```

Flags

Allow Updates

- Flag: `--update`
- Valid inputs: true, false
- Default: false

Indicate whether to overwrite and upgrade existing contracts. Only contracts with difference with existing contracts will be overwritten.

Show Update Diff

- Flag: `--show-diff`
- Valid inputs: true, false
- Default: false

Shows a diff to approve before updating between deployed contract and new contract updates.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`

- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: --config-path

- Short Flag: -f
- Valid inputs: a path in the current filesystem.
- Default: flow.json

Specify the path to the flow.json configuration file.
You can use the -f flag multiple times to merge
several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow
connections.

emulator-snapshot.md:

```
---
```

title: Create Emulator Snapshot
description: How to start create emulator snapshot from the command line
sidebarposition: 4

The Flow CLI provides a command to create emulator snapshots, which are
points in blockchain
history you can later jump to and reset the state to that moment. This
can be useful for testing where you
establish a begining state, run tests and after revert back to the
initial state.

The command syntax is:

```
shell  
flow emulator snapshot create|load|list {name}
```

Example Usage

Create a new snapshot
Create a new emulator snapshot at the current block with a name of
myInitialState.
shell
> flow emulator snapshot create myInitialState

Load an existing snapshot
To jump to a previously created snapshot we use the load command in
combination with the name.
shell
> flow emulator snapshot load myInitialState

List all existing snapshots

To list all the existing snapshots we previously created and can load to we run the following command:

```
shell  
> flow emulator list
```

To learn more about using the Emulator, have a look at the README of the repository.

Flags

Emulator Flags

You can specify any emulator flags found here and they will be applied to the emulator service.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using `-f` flag multiple times.

Version Check

- Flag: `--skip-version-check`
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# project-contracts.md:
```

```
---
```

```
title: Add Project Contracts
description: How to define the Cadence contracts for Flow project
sidebarposition: 2
---
```

Add a Contract

To add a contract to your project, update the "contracts" section of your `flow.json` file.

Contracts are specified as key-value pairs, where the key is the contract name, and the value is the location of the Cadence source code.

For example, the configuration below will register the contract `Foo` from the `FooContract.cdc` file.

```
json
{
  "contracts": {
    "Foo": "./cadence/contracts/FooContract.cdc"
  }
}
```

Define Contract Deployment Targets

Once a contract is added, it can then be assigned to one or more deployment targets.

A deployment target is an account to which the contract will be deployed. In a typical project, a contract has one deployment target per network (e.g. Emulator, Testnet, Mainnet).

Deployment targets are defined in the "deployments" section of your flow.json file.

Targets are grouped by their network, where each network is a mapping from target account to contract list.

Multiple contracts can be deployed to the same target account.

For example, here's how we'd deploy contracts Foo and Bar to the account my-testnet-account:

```
json
{
  "contracts": {
    "Foo": "./cadence/contracts/FooContract.cdc",
    "Bar": "./cadence/contracts/BarContract.cdc"
  },
  "deployments": {
    "testnet": {
      "my-testnet-account": ["Foo", "Bar"]
    }
  }
}
```

```
# start-emulator.md:
```

```
---
title: Start Emulator
description: How to start Flow emulator from the command line
sidebarposition: 1
---
```

The Flow CLI provides a command to start an emulator. The Flow Emulator is a lightweight tool that emulates the behaviour of the real Flow network.

```
shell
flow emulator
```

⚠ The emulator command expects configuration to be initialized. See flow init command.

Example Usage

```
shell
> flow emulator

INFO[0000] ⚡ Using service account 0xf8d6e0586b0a20c7
serviceAddress=f8d6e0586b0a20c7 ...
...
```

To learn more about using the Emulator, have a look at the README of the repository.

Flags

Emulator Flags

You can specify any emulator flags found here and they will be applied to the emulator service.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using -f flag multiple times.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# configuration.md:

---
title: Configuration
description: What is Flow CLI Configuration
sidebarposition: 2
---
```

Flow CLI uses a state called configuration which is stored in a file (usually flow.json).

Flow configuration (flow.json) file will contain the following properties:

- A networks list pre-populated with the Flow emulator, testnet and mainnet connection configuration.
- An accounts list pre-populated with the Flow Emulator service account.
- A deployments empty object where all deployment targets can be defined.
- A contracts empty object where you define contracts you wish to deploy.

Example Project Configuration

```
json
{
  "networks": {
    "emulator": "127.0.0.1:3569",
    "mainnet": "access.mainnet.nodes.onflow.org:9000",
    "testnet": "access.devnet.nodes.onflow.org:9000"
  },
  "accounts": {
    "emulator-account": {
      "address": "f8d6e0586b0a20c7",
      "key": "ae1b44c0f5e8f6992ef2348898a35e50a8b0b9684000da8b1dade1b3bcd6ebbe",
    }
  },
  "deployments": {},
  "contracts": {}
}
```

Configuration

Below is an example of a configuration file for a complete Flow project. We'll walk through each property one by one.

```
json
{
  "contracts": {
    "NonFungibleToken": "./cadence/contracts/NonFungibleToken.cdc",
    "Kibble": "./cadence/contracts/Kibble.cdc",
    "KittyItems": "./cadence/contracts/KittyItems.cdc",
    "KittyItemsMarket": "./cadence/contracts/KittyItemsMarket.cdc",
    "FungibleToken": {
      "source": "./cadence/contracts/FungibleToken.cdc",
      "aliases": {
        "testnet": "9a0766d93b6608b7",
        "emulator": "ee82856bf20e2aa6"
      }
    }
  },
}
```

```

"deployments": {
  "testnet": {
    "admin-account": ["NonFungibleToken"],
    "user-account": ["Kibble", "KittyItems", "KittyItemsMarket"]
  },
  "emulator": {
    "emulator-account": [
      "NonFungibleToken",
      "Kibble",
      "KittyItems",
      "KittyItemsMarket"
    ]
  }
},
"accounts": {
  "admin-account": {
    "address": "3ae53cb6e3f42a79",
    "key": "12332967fd2bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111"
  },
  "user-account": {
    "address": "e2a8b7f23e8b548f",
    "key": "22232967fd2bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111"
  },
  "emulator-account": {
    "address": "f8d6e0586b0a20c7",
    "key": "2eae2f31cb5b756151fa11d82949c634b8f28796a711d7eb1e52cc301ed1111",
  }
},
"networks": {
  "emulator": "127.0.0.1:3569",
  "mainnet": "access.mainnet.nodes.onflow.org:9000",
  "testnet": "access.devnet.nodes.onflow.org:9000",
  "testnetSecure": {
    "Host": "access-001.devnet30.nodes.onflow.org:9001",
    "NetworkKey": "ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daebf836f7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d"
  }
}
}

```

Contracts

Contracts are specified as key-value pairs, where the key is the contract name, and the value is the location of the Cadence source code.

The advanced format allows us to specify aliases for each network.

Simple Format

```
json
...
"contracts": {
    "NonFungibleToken": "./cadence/contracts/NonFungibleToken.cdc"
}
...
...
```

Advanced Format

Using advanced format we can define aliases. Aliases define an address where the contract is already deployed for that specific network. In the example scenario below the contract FungibleToken would be imported from the address 9a0766d93b6608b7 when deploying to testnet network and address ee82856bf20e2aa6 when deploying to the Flow emulator. We can specify aliases for each network we have defined. When deploying to testnet it is always a good idea to specify aliases for all the common contracts that have already been deployed to the testnet.

⚠ If we use an alias for the contract we should not specify it in the deployment section for that network.

```
json
...
"FungibleToken": {
    "source": "./cadence/contracts/FungibleToken.cdc",
    "aliases": {
        "testnet": "9a0766d93b6608b7",
        "emulator": "ee82856bf20e2aa6"
    }
}
...
```

Format used to specify advanced contracts is:

```
json
"CONTRACT NAME": {
    "source": "CONTRACT SOURCE FILE LOCATION",
    "aliases": {
        "NETWORK NAME": "ADDRESS ON SPECIFIED NETWORK WITH DEPLOYED
CONTRACT"
        ...
    }
}
```

Accounts

The accounts section is used to define account properties such as keys and addresses.

Each account must include a name, which is then referenced throughout the configuration file.

Simple Format

When using the simple format, simply specify the address for the account, and a single hex-encoded private key.

```
json
...
"accounts": {
    "admin-account": {
        "address": "3ae53cb6e3f42a79",
        "key": "12332967fd2bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111"
    }
}
...
...
```

Advanced format

The advanced format allows us to define more properties for the account. We can define the signature algorithm and hashing algorithm, as well as custom key formats.

Please note that we can use service for address in case the account is used on emulator network as this is a special value that is defined on the run time to the default service address on the emulator network.

Example for advanced hex format:

```
json
...
"accounts": {
    "admin-account": {
        "address": "service",
        "key": {
            "type": "hex",
            "index": 0,
            "signatureAlgorithm": "ECDSAP256",
            "hashAlgorithm": "SHA3256",
            "privateKey": "12332967fd2bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111"
        }
    }
}
```

...

You can also use BIP44 to derive keys from a mnemonic. For more details please see the FLIP

Example for BIP44 format:

json

...

```
"accounts": {  
    "admin-account": {  
        "address": "service",  
        "key": {  
            "type": "bip44",  
            "index": 0,  
            "signatureAlgorithm": "ECDSAP256",  
            "hashAlgorithm": "SHA3256",  
            "mnemonic": "skull design wagon top faith actor valley crystal  
subject volcano access join",  
            "derivationPath": "m/44'/539'/0'/0/0"  
        }  
    }  
}
```

...

Note: Default value for derivationPath is m/44'/539'/0'/0/0 if omitted.

You can also use a key management system (KMS) to sign the transactions. Currently, we only support Google KMS.

Example for Google KMS format:

json

...

```
"accounts": {  
    "admin-account": {  
        "address": "service",  
        "key": {  
            "type": "google-kms",  
            "index": 0,  
            "signatureAlgorithm": "ECDSAP256",  
            "hashAlgorithm": "SHA3256",  
            "resourceID":  
"projects/flow/locations/us/keyRings/foo/bar/cryptoKeyVersions/1"  
        }  
    }  
}
```

...

You can store the account key to a separate file and provide the file location as part of the key configuration.

Example for separate key file:

```
json
...
"accounts": {
    "admin-account": {
        "address": "service",
        "key": {
            "type": "file",
            "location": "./test.key"
        }
    }
}
...
```

Inside the test.key file you should only put the hex key content (e.g. 12332967fd2bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111)

Deployments

The deployments section defines where the project deploy command will deploy specified contracts.

This configuration property acts as the glue that ties together accounts, contracts and networks, all of which are referenced by name.

In the deployments section we specify the network, account name and list of contracts to be deployed to that account.

Format specifying the deployment is:

```
json
...
"deployments": {
    "NETWORK": {
        "ACCOUNT NAME": ["CONTRACT NAME"]
    }
}
```

...

```
json
...
"deployments": {
    "emulator": {
        "emulator-account": [
            "NonFungibleToken",
            "Kibble",
            "KittyItems",
            "KittyItemsMarket"
        ]
    }
}
```

```

        ]
    },
    "testnet": {
        "admin-account": ["NonFungibleToken"],
        "user-account": [
            "Kibble",
            "KittyItems",
            "KittyItemsMarket"
        ]
    }
}
...

```

Networks

Use this section to define networks and connection parameters for that specific network.

Format for networks is:

```

json
...
"networks": {
    "NETWORK NAME": "ADDRESS"
}
...

```



```

json
...
"networks": {
    "NETWORK NAME": {
        "host": "ADDRESS",
        "key": "ACCESS NODE NETWORK KEY"
    }
}
...

```



```

json
...
"networks": {
    "emulator": "127.0.0.1:3569",
    "mainnet": "access.mainnet.nodes.onflow.org:9000",
    "testnet": "access.devnet.nodes.onflow.org:9000",
    "testnetSecure": {
        "host": "access-001.devnet30.nodes.onflow.org:9001",
        "key": "ba69f7d2e82b9edf25b103c195cd371cf0cc047ef8884a9bbe331e62982d46daebf836f
7445a2ac16741013b192959d8ad26998aff12f2adc67a99e1eb2988d"
    },
}

```

```
}
```

...

Emulators

The default emulator CLI is automatically configured with name being "default" and values of serviceAccount: "emulator-account" and port: "3569". The default emulator configuration will not show up on flow.json.

To customize emulator values, add emulator section like the example below:

```
json
...
"emulators": {
    "custom-emulator": {
        "port": 3600,
        "serviceAccount": "emulator-account"
    }
}
...
# initialize-configuration.md:

---
title: Initialize Configuration
description: How to initialize Flow configuration using CLI
sidebarposition: 1
---

Flow CLI uses a state to operate which is called configuration (usually flow.json file).
Before using commands that require this configuration we must initialize the project by using the init command. Read more about state configuration here.

shell
flow init
```

Example Usage

```
shell
> flow init

Configuration initialized
Service account: 0xf8d6e0586b0a20c7
```

```
Start emulator by running: 'flow emulator'  
Reset configuration using: 'flow init --reset'
```

Error Handling

Existing configuration will cause the error below.
You should initialize in an empty folder or reset configuration using --reset flag
or by removing the configuration file first.
shell
X Command Error: configuration already exists at: flow.json, if you want to reset configuration use the reset flag

Global Configuration

Flow supports global configuration which is a flow.json file saved in your home directory and loaded as the first configuration file wherever you execute the CLI command.

Please be aware that global configuration has the lowest priority and is overwritten by any other configuration file if they exist (if flow.json exist in your current directory it will overwrite properties in global configuration, but only those which overlap).

You can generate a global configuration using --global flag.

Command example: flow init --global.

Global flow configuration is saved as:
- MacOs: /flow.json
- Linux: /flow.json
- Windows: C:\Users\\$USER\flow.json

Flags

Reset

- Flag: --reset

Using this flag will reset the existing configuration and create a new one.

Global

- Flag: --global

Using this flag will create a global Flow configuration.

Service Private Key

- Flag: `--service-private-key`
- Valid inputs: a hex-encoded private key in raw form.

Private key used on the default service account.

Service Key Signature Algorithm

- Flag: `--service-sig-algo`
- Valid inputs: "ECDSAP256", "ECDSAssecp256k1"
- Default: "ECDSAP256"

Specify the ECDSA signature algorithm for the provided public key.

Flow supports the secp256k1 and P-256 curves.

Service Key Hash Algorithm

- Flag: `--service-hash-algo`
- Valid inputs: "SHA2256", "SHA3256"
- Default: "SHA3256"

Specify the hashing algorithm that will be paired with the public key upon account creation.

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see while command execution.

```
# manage-configuration.md:
```

```
---
```

```
title: Manage Configuration
description: How to configure the Flow CLI
sidebarposition: 3
---
```

Configuration should be managed by using config add and config remove commands. Using add command will also validate values that will be added to the configuration.

```
shell
flow config add <account|contract|network|deployment>
flow config remove <account|contract|network|deployment>
```

Example Usage

```
shell
flow config add account

Name: Admin
Address: f8d6e0586b0a20c7
✓ ECDSAP256
✓ SHA3256
Private key: e382a0e494...9285809356
Key index (Default: 0): 0
```

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using -f flag multiple times.

```
# security.md:

---
title: Security
description: How to securely use CLI
sidebarposition: 4
---
```

The managing of accounts and private keys is intrinsically dangerous. We must take extra precautions to not expose private key data when using the CLI.

The Flow CLI provides several options to secure private account data.

⚠ Warning: please be careful when using private keys in configuration files.
Never commit private key data to source control.
If private key data must be kept in text, we suggest using a separate file
that is not checked into source control (e.g. excluded with `.gitignore`).

Private Account Configuration File
Storing an account key to a separate file which is not checked into source control (e.g. excluded with `.gitignore`) can be the first step towards better security.

Main configuration file
json
...
"accounts": {
 "my-testnet-account": {
 "address": "3ae53cb6e3f42a79",
 "key": {
 "type": "file",
 "location": "./my-testnet-account.key"
 }
 }
}
...

Separate account key file
⚠ Put this file in `.gitignore`

The `my-testnet-account.key` file only contains the hex-encoded private key.

334232967f52bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111

Private configuration file

⚠ Put this file in `.gitignore`:

```
json
// flow.testnet.json
{
  "accounts": {
    "my-testnet-account": {
      "address": "3ae53cb6e3f42a79",
      "key": "334232967f52bd75234ae9037dd4694c1f00baad63a10c35172bf65fbb8ad1111"
    }
  }
}
```

Store Configuration in Environment Variables

You can use environment variables for values that should be kept private (e.g. private keys, addresses).

See example below:

```
shell  
PRIVATEKEY=key flow project deploy
```

```
json  
// flow.json  
{  
  ...  
  "accounts": {  
    "my-testnet-account": {  
      "address": "3ae53cb6e3f42a79",  
      "key": "$PRIVATEKEY"  
    }  
  }  
  ...  
}
```

Private Dotenv File

The CLI will load environment variables defined in the .env file in the active directory, if one exists.

These variables can be substituted inside the flow.json, just like any other environment variable.

⚠ You should never commit .env to source control, especially if it contains sensitive information like a private key.

Example .env file:
bash
PRIVATEKEY=123

Composing Multiple Configuration Files

You can merge multiple configuration files like so:

```
shell  
flow project deploy -f main.json -f private.json
```

get-blocks.md:

```
title: Get Block
description: How to get a block from the command line
sidebarposition: 1
---
```

The Flow CLI provides a command to fetch any block from the Flow network.

```
shell
flow blocks get <blockid|latest|blockheight>
```

Example Usage

```
shell
flow blocks get 12884163 --host access.mainnet.nodes.onflow.org:9000 --
include transactions
```

Example response

```
shell
Block ID
  2fb7571a6ccf02f3ac42f27c14ce0a4cb119060e4fb7af36fd51894465e7002
Prent ID
  1c5a6267ba9512e141e4e90630cb326cecfbf6113818487449efeb37fc98ca18
Timestamp
  2021-03-19 17:46:15.973305066 +0000 UTC
Height
  12884163
Status
  Sealed
Total Seals
  2
Total Collections
  8
    Collection 0:
      3e694588e789a72489667a36dd73104dea4579bcd400959d47aedcccd7f930eeb
        Transaction 0:
          acc2ae1ff6deb2f4d7663d24af6ab1baf797ec264fd76a745a30792f6882093b
            Transaction 1:
              ae8bfbc85ce994899a3f942072bfd3455823b1f7652106ac102d161c17fc55c
                Transaction 2:
                  70c4d39d34e654173c5c2746e7bb3a6cdf1f5e6963538d62bad2156fc02ea1b2
                    Transaction 3:
                      2466237b5eafb469c01e2e5f929a05866de459df3bd768cde748e068c81c57bf
                        Collection 1:
                          e93f2bd988d66288c7e1ad991dec227c6c74b8039a430e43896ad94cf8fecccce
                            Transaction 0:
                              4d790300722b646e7ed3e2c52675430d7ccf2efd1d93f106b53bc348df601af6
                                Collection 2:
                                  c7d93b80ae55809b1328c686f6a8332e8e15083ab32f8b3105c4d910646f54bf
                                    Transaction 0:
                                      95c4efbb30f86029574d6acd7df04afe6108f6fd610d823dfd398c80cfa5e842
                                        Collection 3:
                                          1a4f563b48aaa38f3a7e867c89422e0bd84887de125e8f48ba147f4ee58ddf0d
                                            Transaction 0:
                                              fbcc99326336d4dbb4cbc01a3b9b85cfcdcdc071b3d0e01ee88ecd144444600b
                                                Collection 4:
                                                  01000c7773cc3c22cba6d8917a2486dc7a1a1842dd7fb7c0e87e63c22bb14abe
```

```
    Transaction 0:  
a75097639b434044de0122d3a28620e093f277fa715001e80a035568e118c59f  
    Collection 5:  
      6f2b08f9673545a2e61e954feb8d55d2a3ef2b3cef7a8d2f8de527bc42d92c28  
        Transaction 0:  
8ea63d397bd07a25db3f06fb9785dbf09bc652159f68a84c55ea2be606ada1e9  
    Collection 6:  
      13b5c48252930824a8c6e846470763582cacdacb772c1e9c584adefced6724b2  
        Transaction 0:  
8ba57a92311367189a89a59bcb3c32192387fefca9bde493e087bc0d479186a8  
        Transaction 1:  
8ab1d99702ccf31b6f4b3acd2580ddd440f08bc07acab4884337c0c593a8f69  
    Collection 7:  
      bf90fdd2761b8f37565af60fc38165dd09edf0671fdd35b37f718a7eb45e804f  
        Transaction 0:  
b92a14c0802183719efed00363d31076d7e50f41a6207781cf34d39c822bbacb
```

Arguments

Query

- Name: <blockid|latest|blockheight>
- Valid Input: Block ID, latest or block height

Specify the block to retrieve by block ID or block height.

Arguments

Address

- Name: address
- Valid Input: Flow account address

Flow account address (prefixed with 0x or not).

Flags

Events

- Flag: --events
- Valid inputs: Valid event name

List events of this type for the block.

Include

- Flag: --include
- Valid inputs: transactions

Include additional values in the response.

Signer

- Flag: `--signer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used to sign the transaction.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: `127.0.0.1:3569` (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

`get-collections.md:`

```
---
title: Get Collection
description: How to get a collection from the command line
sidebarposition: 3
---
```

The Flow CLI provides a command to fetch any collection from the Flow network.

```
shell
flow collections get <collectionid>
```

Example Usage

```
shell
flow collections get
3e694588e789a72489667a36dd73104dea4579bcd400959d47aedcc7f930eeb \
--host access.mainnet.nodes.onflow.org:9000
```

Example response

```
shell
Collection ID
3e694588e789a72489667a36dd73104dea4579bcd400959d47aedcc7f930eeb:
acc2ae1ff6deb2f4d7663d24af6ab1baf797ec264fd76a745a30792f6882093b
ae8bfbc85ce994899a3f942072bfd3455823b1f7652106ac102d161c17fc55c
70c4d39d34e654173c5c2746e7bb3a6cdf1f5e6963538d62bad2156fc02ea1b2
2466237b5eafb469c01e2e5f929a05866de459df3bd768cde748e068c81c57bf
```

Arguments

Collection ID

- Name: collectionid
- Valid Input: SHA3-256 hash of the collection contents

Arguments

Flags

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.

You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

```
Skip version check during start up to speed up process for slow connections.
```

```
# get-events.md:
```

```
---
```

```
title: Get Events
```

```
description: How to get an event from the command line
```

```
sidebarposition: 2
```

```
---
```

Use the event command to fetch a single or multiple events in a specific range of blocks.

You can provide start and end block height range, but also specify number of the latest blocks to be used to search for specified event. Events are fetched concurrently by using multiple workers which optionally you can also control by specifying the flags.

```
shell
flow events get <eventname>
```

Example Usage

```
Get the event by name A.0b2a3299cc857e29.TopShot.Deposit from the last 20 blocks on mainnet.
```

```
shell
> flow events get A.0b2a3299cc857e29.TopShot.Deposit --last 20 --network mainnet
```

```
Events Block #12913388:
  Index      2
  Type  A.0b2a3299cc857e29.TopShot.Deposit
  Tx ID
0a1e6cdc4eeda0e23402193d7ad5ba01a175df4c08f48fa7ac8d53e811c5357c
  Values
    id (UInt64)      3102159
    to ({})?        24214cf0faa7844d

  Index      2
  Type  A.0b2a3299cc857e29.TopShot.Deposit
  Tx ID
1fa5e64dc8ed5dad87ba58207ee4c058feb38fa271fff659ab992dc2ec2645
  Values
    id (UInt64)      5178448
    to ({})?        26c96b6c2c31e419

  Index      9
  Type  A.0b2a3299cc857e29.TopShot.Deposit
  Tx ID
262ab3996bdf98f5f15804c12b4e5d4e89c0fa9b71d57be4d7c6e8288c507c4a
  Values
    id (UInt64)      1530408
```

```

        to ({ }?)    2da5c6d1a541971b

...

Get two events A.1654653399040a61.FlowToken.TokensDeposited
and A.1654653399040a61.FlowToken.TokensWithdrawn in the block height
range on mainnet.
shell
> flow events get \
A.1654653399040a61.FlowToken.TokensDeposited \
A.1654653399040a61.FlowToken.TokensWithdrawn \
--start 11559500 --end 11559600 --network mainnet

Events Block #17015045:
Index 0
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
6dcf60d54036acb52b2e01e69890ce34c3146849998d64364200e4b21e9ac7f1
Values
- amount (UFix64): 0.00100000
- from (Address?): 0x9e06eebf494e2d78

Index 1
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
6dcf60d54036acb52b2e01e69890ce34c3146849998d64364200e4b21e9ac7f1
Values
- amount (UFix64): 0.00100000
- from (Never?): nil

Events Block #17015047:
Index 0
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
24979a3c0203f514f7f5822cc8ae7046e24f25d4a775bef697a654898fb7673e
Values
- amount (UFix64): 0.00100000
- from (Address?): 0x18eb4ee6b3c026d2

Index 1
Type A.1654653399040a61.FlowToken.TokensWithdrawn
Tx ID
24979a3c0203f514f7f5822cc8ae7046e24f25d4a775bef697a654898fb7673e
Values
- amount (UFix64): 0.00100000
- from (Never?): nil

```

Arguments

Event Name

- Name: eventname

- Valid Input: String

Fully-qualified identifier for the events.

You can provide multiple event names separated by a space.

Flags

Start

- Flag: --start
- Valid inputs: valid block height

Specify the start block height used alongside the end flag.

This will define the lower boundary of the block range.

End

- Flag: --end
- Valid inputs: valid block height

Specify the end block height used alongside the start flag.

This will define the upper boundary of the block range.

Last

- Flag: --last
- Valid inputs: number
- Default: 10

Specify the number of blocks relative to the last block. Ignored if the start flag is set. Used as a default if no flags are provided.

Batch

- Flag: --batch
- Valid inputs: number
- Default: 25

Number of blocks each worker will fetch.

Workers

- Flag: --workers
- Valid inputs: number
- Default: 10

Number of workers to use when fetching events concurrently.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug

- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: a path in the current filesystem.
- Default: flow.json

Specify the path to the flow.json configuration file.
You can use the -f flag multiple times to merge several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# get-status.md:
```

```
---
```

```
title: Network Status
description: How to get access node status from the command line
sidebarposition: 4
---
```

The Flow CLI provides a command to get network status of specified Flow Access Node

```
flow status
```

Example Usage

```
shell
> flow status --network testnet

Status:          □ ONLINE
Network:        testnet
Access Node:    access.devnet.nodes.onflow.org:9000
```

Flags

Network

- Flag: --network
- Short Flag: -n

- Valid inputs: the name of a network defined in the configuration (flow.json) .

Specify which network you want the command to use for execution.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug

- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: --conf
- Short Flag: -f
- Valid inputs: a path in the current filesystem.
- Default: flow.json

Specify the path to the flow.json configuration file.
You can use the -f flag multiple times to merge several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# decode-keys.md:
```

```
---
```

```
title: Decode Public Keys
description: How to decode Flow public keys from the command line
sidebarposition: 2
---
```

The Flow CLI provides a command to decode encoded public account keys.

```
shell
flow keys decode <rlp|pem> <encoded public key>
```

Example Usage

```
Decode RLP Encoded Public Key
shell
> flow keys decode rlp
f847b84084d716c14b051ad6b001624f738f5d302636e6b07cc75e4530af7776a4368a2b5
86dbefc0564ee28384c2696f178cbcd52e62811bcc9ecb59568c996d342db2402038203e8

Public Key      84d716c1...bcc9ecb59568c996d342db24
Signature algorithm  ECDSAP256
Hash algorithm    SHA3256
Weight            1000
Revoked          false
```

Decode PEM Encoded Public Key From File

```
shell
> flow keys decode pem --from-file key.pem

Public Key          d479b3c...c4615360039a6660a366a95f
Signature algorithm   ECDSAP256
Hash algorithm       UNKNOWN
Revoked             false
```

Arguments

Encoding

- Valid inputs: rlp, pem

First argument specifies a valid encoding of the public key provided.

Optional: Public Key

- Name: encoded public key
- Valid inputs: valid encoded key content

Optional second argument provides content of the encoded public key.
If this argument is omitted the --from-file must be used instead.

Flags

From File

- Flag: --from-file
- Valid inputs: valid filepath

Provide file with the encoded public key.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s

- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# derive-keys.md:
```

```
---
```

```
title: Derive Public Key
description: How to derive Flow public key from a private key from the
command line
sidebarposition: 3
---
```

The Flow CLI provides a command to derive Public Key from a Private Key.

```
shell
flow keys derive <private key>
```

Example Usage

```
Derive Public Key from a Private Key
shell
> flow keys derive
c778170793026a9a7a3815dabed68ded445bde7f40a8c66889908197412be89f
```

Example response

```
shell
> flow keys generate
```

● Store Private Key safely and don't share with anyone!

```
Private Key
c778170793026a9a7a3815dabed68ded445bde7f40a8c66889908197412be89f
Public Key
584245c57e5316d6606c53b1ce46dae29f5c9bd26e9e8...aaa5091b2eebcb2ac71c75cf7
0842878878a2d650f7
```

Arguments

Private Key

- Name: private key
- Valid inputs: valid private key content

Flags

Signature Algorithm

- Flag: `--sig-algo`
- Valid inputs: "ECDSAP256", "ECDSAssecp256k1"

Specify the ECDSA signature algorithm for the key pair.

Flow supports the secp256k1 and P-256 curves.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

```
# generate-keys.md:  
---  
title: Generate Keys  
description: How to generate key pair from the command line  
sidebarposition: 1  
---
```

The Flow CLI provides a command to generate ECDSA key pairs that can be attached to new or existing Flow accounts.

```
shell  
flow keys generate
```

⚠ Store private key safely and don't share with anyone!

Example Usage

```
shell
flow keys generate
```

Example response

```
shell
> flow keys generate
```

● Store Private Key safely and don't share with anyone!

Private Key
c778170793026a9a7a3815dabed68ded445bde7f40a8c66889908197412be89f
Public Key
584245c57e5316d6606c53b1ce46dae29f5c9bd26e9e8...aaa5091b2eebcb2ac71c75cf7
0842878878a2d650f7

Flags

Seed

- Flag: --seed
- Valid inputs: any string with length >= 32

Specify a UTF-8 seed string that will be used to generate the key pair.
Key generation is deterministic, so the same seed will always result in the same key.

If no seed is specified, the key pair will be generated using a random 32 byte seed.

⚠ Using seed with production keys can be dangerous if seed was not generated by using safe random generators.

Signature Algorithm

- Flag: --sig-algo
- Valid inputs: "ECDSAP256", "ECDSAssecp256k1"

Specify the ECDSA signature algorithm for the key pair.

Flow supports the secp256k1 and P-256 curves.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: a path in the current filesystem.
- Default: flow.json

Specify the path to the flow.json configuration file.

You can use the -f flag multiple times to merge several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# execute-scripts.md:
```

```
---
```

```
title: Execute a Script
```

```
description: How to execute a Cadence script on Flow from the command
line
sidebarposition: 6
---
```

The Flow CLI provides a command to execute a Cadence script on the Flow execution state with any Flow Access API.

```
shell
flow scripts execute <filename> [<argument> <argument>...] [flags]
```

Example Usage

```
shell
Execute a script on Flow Testnet
> flow scripts execute script.cdc "Hello" "World"
"Hello World"
```

Script source code:

```
access(all) fun main(greeting: String, who: String): String {
    return greeting.concat(" ").concat(who)
}
```

Arguments

Filename

- Name: filename
- Valid inputs: a path in the current filesystem.

The first argument is a path to a Cadence file containing the script to be executed.

Arguments

- Name: argument
- Valid inputs: valid cadence values matching argument type in script code.

Input arguments values matching corresponding types in the source code and passed in the same order.

You can pass a nil value to optional arguments by executing the flow script like this: flow scripts execute script.cdc nil.

Flags

Arguments JSON

- Flag: --args-json

- Valid inputs: arguments in JSON-Cadence form.
- Example: flow scripts execute script.cdc '[{"type": "String", "value": "Hello World"}]'

Arguments passed to the Cadence script in the Cadence JSON format. Cadence JSON format contains type and value keys and is documented here.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# run-tests.md:  
---  
title: Run Cadence tests  
description: How to run Cadence tests from the command line  
sidebarposition: 11  
---
```

The Flow CLI provides a command to run Cadence tests.

```
shell  
flow test /path/to/testscript.cdc
```

⚠ The test command expects configuration to be initialized. See `flow init` command.

Example Usage

```
A simple Cadence script testscript.cdc, which has a test case for running
a cadence script on-chain:
cadence
import Test

access(all) let blockchain = Test.newEmulatorBlockchain()

access(all) fun testSumOfTwo() {
    let scriptResult = blockchain.executeScript(
        "access(all) fun main(a: Int, b: Int): Int { return a + b }",
        [2, 3]
    )

    Test.expect(scriptResult, Test.beSucceeded())

    let sum = scriptResult.returnValue! as! Int
    Test.assertEqual(5, sum)
}
```

The above test-script can be run with the CLI as follows, and the test results will be printed on the console.

```
shell
$ flow test testscript.cdc

Test results: "testscript.cdc"
- PASS: testSumOfTwo
```

To learn more about writing tests in Cadence, take a look at the Cadence testing framework.

Flags

Coverage

- Flag: --cover
- Default: false

Use the cover flag to calculate coverage report for the code being tested.

```
shell
$ flow test --cover testscript.cdc
```

```
Test results: "testscript.cdc"
- PASS: testSumOfTwo
Coverage: 96.5% of statements
```

Coverage Report File

- Flag: `--coverprofile`
- Valid inputs: valid filename and extension
- Default: "coverage.json"

Use the `coverprofile` to specify the filename where the calculated coverage report is to be written. Supported filename extensions are `.json` and `.lcov`.

```
shell  
$ flow test --cover tests/testscript.cdc
```

```
$ cat coverage.json
```

```
$ flow test --cover --coverprofile="coverage.lcov" tests/testscript.cdc  
$ cat coverage.lcov
```

Coverage Code Type

- Flag: `--covercode`
- Valid inputs: "all", "contracts"
- Default: "all"

Use the `covercode` flag to calculate coverage report only for certain types of code. A value of "contracts" will exclude scripts and transactions from the coverage report.

```
shell  
$ flow test --cover --covercode="contracts" tests/testscript.cdc
```

```
Test results: "tests/testscript.cdc"  
- PASS: testSumOfTwo  
There are no statements to cover
```

Since we did not use any contracts in our sample test script, there is no coverage percentage to be reported.

```
# build-transactions.md:
```

```
---
```

```
title: Build a Transaction  
description: How to build a Flow transaction from the command line  
sidebarposition: 3  
---
```

The Flow CLI provides a command to build a transactions with options to specify authorizer accounts, payer account and proposer account.

The build command doesn't produce any signatures and instead is designed to be used with the sign and send-signed commands.

Use this functionality in the following order:

1. Use this command (build) to build the transaction.
2. Use the sign command to sign with each account specified in the build process.
3. Use the send-signed command to submit the signed transaction to the Flow network.

```
shell
flow transactions build <code filename> [<argument> <argument>...]
[flags]
```

Example Usage

```
shell
> flow transactions build ./transaction.cdc "Meow" \
--authorizer alice \
--proposer bob \
--payer charlie \
--filter payload --save built.rlp

ID
e8c0a69952fbe50a66703985e220307c8d44b8fa36c76cbca03f8c43d0167847
Payer      e03daebcd8ca0615
Authorizers [f3fcfd2c1a78f5eee]

Proposal Key:
Address 179b6b1cb6755e31
Index    0
Sequence 1
```

No Payload Signatures

No Envelope Signatures

Arguments (1):
- Argument 0: {"type": "String", "value": "Meow"}

Code

```
transaction(greeting: String) {
  let guest: Address

  prepare(authorizer: &Account) {
    self.guest = authorizer.address
  }

  execute {
    log(greeting.concat(",")).concat(self.guest.toString()))
  }
}
```

Payload:
f9013df90138b8d17472616e...73616374696f6e286eec0c0

JSON arguments from a file example:

```
shell  
> flow transactions build tx1.cdc --args-json "$(cat args.json)"
```

Arguments

Code Filename

- Name: filename
- Valid inputs: Any filename and path valid on the system.

The first argument is a path to a Cadence file containing the transaction to be executed.

Arguments

- Name: argument
- Valid inputs: valid cadence values matching argument type in transaction code.

Input arguments values matching corresponding types in the source code and passed in the same order.

For passing complex argument values see send transaction document.

Flags

Payer

- Flag: --payer
- Valid Inputs: Flow address or account name from configuration.
- Default: service account

Specify account address that will be paying for the transaction.
Read more about payers here.

Proposer

- Flag: --proposer
- Valid inputs: Flow address or account name from configuration.
- Default: service account

Specify a name of the account that is proposing the transaction.
Account must be defined in flow configuration.

Proposer Key Index

- Flag: --proposer-key-index
- Valid inputs: number of existing key index

- Default: 0

Specify key index for the proposer account.

Authorizer

- Flag: --authorizer
- Valid Inputs: Flow address or account name from configuration.
- Default: service account

Additional authorizer addresses to add to the transaction.
Read more about authorizers [here](#).

Arguments JSON

- Flag: --args-json
- Valid inputs: arguments in JSON-Cadence form.
- Example: flow transactions build ./tx.cdc '[{"type": "String", "value": "Hello World"}]'

Arguments passed to the Cadence transaction in Cadence JSON format.
Cadence JSON format contains type and value keys and is
documented [here](#).

Gas Limit

- Flag: --gas-limit
- Valid inputs: an integer greater than zero.
- Default: 1000

Specify the gas limit for this transaction.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be
used to execute the commands.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be
used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n

- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Include Fields

- Flag: --include
- Valid inputs: code, payload, signatures

Specify fields to include in the result output. Applies only to the text output.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify in which format you want to display the result.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: valid filename

Specify the filename where you want the result to be saved.

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see while command execution.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using -f flag multiple times.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

complex-transactions.md:

```
---
```

title: Build a Complex Transaction
description: How to build and send a complex Flow transaction from the command line
sidebarposition: 4

Simple Transactions

Sending a transaction using the Flow CLI can simply be achieved by using the send command documented here.

Complex Transactions

If you would like to build more complex transactions the Flow CLI provides commands to build, sign and send transactions allowing you to specify different authorizers, signers and proposers.

The process of sending a complex transactions includes three steps:

1. build a transaction
2. sign the built transaction
3. send signed transaction

Read more about each command flags and arguments in the above links.

Examples

We will describe common examples for complex transactions. All examples are using an example configuration.

Single payer, proposer and authorizer

The simplest Flow transaction declares a single account as the proposer, payer and authorizer.

Build the transaction:

shell

```
> flow transactions build tx.cdc
--proposer alice
--payer alice
--authorizer alice
--filter payload --save tx1

Sign the transaction:
shell
> flow transactions sign tx1 --signer alice
--filter payload --save tx2

Submit the signed transaction:
shell
> flow transactions send-signed tx2

Transaction content (tx.cdc):
transaction {
    prepare(signer: &Account) {}
    execute { ... }
}

Single payer and proposer, multiple authorizers
A transaction that declares same payer and proposer but multiple
authorizers each required to sign the transaction. Please note that the
order of signing is important, and the payer must sign last.

Build the transaction:
shell
> flow transactions build tx.cdc
--proposer alice
--payer alice
--authorizer bob
--authorizer charlie
--filter payload --save tx1

Sign the transaction with authorizers:
shell
> flow transactions sign tx1 --signer bob
--filter payload --save tx2

shell
> flow transactions sign tx2 --signer charlie
--filter payload --save tx3

Sign the transaction with payer:
shell
> flow transactions sign tx3 --signer alice
--filter payload --save tx4

Submit the signed transaction:
shell
> flow transactions send-signed tx4
```

Transaction content (tx.cdc):

```
transaction {
    prepare(bob: &Account, charlie: &Account) {}
    execute { ... }
}
```

Different payer, proposer and authorizer

A transaction that declares different payer, proposer and authorizer each signing separately.

Please note that the order of signing is important, and the payer must sign last.

Build the transaction:

```
shell
> flow transactions build tx.cdc
--proposer alice
--payer bob
--authorizer charlie
--filter payload --save tx1
```

Sign the transaction with proposer:

```
shell
> flow transactions sign tx1 --signer alice
--filter payload --save tx2
```

Sign the transaction with authorizer:

```
shell
> flow transactions sign tx2 --signer charlie
--filter payload --save tx3
```

Sign the transaction with payer:

```
shell
> flow transactions sign tx3 --signer bob
--filter payload --save tx4
```

Submit the signed transaction:

```
shell
> flow transactions send-signed tx4
```

Transaction content (tx.cdc):

```
transaction {
    prepare(charlie: &Account) {}
    execute { ... }
}
```

Single payer, proposer and authorizer but multiple keys

A transaction that declares same payer, proposer and authorizer but the signer account has two keys with half weight, required to sign with both.

```
Build the transaction:  
shell  
> flow transactions build tx.cdc  
  --proposer dylan1  
  --payer dylan1  
  --authorizer dylan1  
  --filter payload --save tx1  
  
Sign the transaction with the first key:  
shell  
> flow transactions sign tx1 --signer dylan1  
  --filter payload --save tx2  
  
Sign the transaction with the second key:  
shell  
> flow transactions sign tx2 --signer dylan2  
  --filter payload --save tx3  
  
Submit the signed transaction:  
shell  
> flow transactions send-signed tx3  
  
Transaction content (tx.cdc):  
  
transaction {  
    prepare(signer: &Account) {}  
    execute { ... }  
}  
  
Configuration  
This is an example configuration using mock values:  
json  
{  
  ...  
  "accounts": {  
    "alice": {  
      "address": "0x1",  
      "key": "111...111"  
    },  
    "bob": {  
      "address": "0x2",  
      "key": "222...222"  
    },  
    "charlie": {  
      "address": "0x3",  
      "key": "333...333"  
    },  
    "dylan1": {  
      "address": "0x4",  
      "key": "444...444"  
    },  
    "dylan2": {  
    }  
  }  
}
```

```
        "address": "0x4",
        "key": "555...555"
    }
}
...
}

# decode-transactions.md:

---
```

```
title: Build a Complex Transaction
description: How to decode a Flow transaction from the command line
sidebarposition: 7
---
```

The Flow CLI provides a command to decode a transaction from RLP in a file. It uses same transaction format as get command

```
shell
flow transactions decode <file>
```

Example Usage

```
shell
> flow transactions decode ./rlp-file.rlp

ID
c1a52308fb906358d4a33c1f1d5fc458d3cfeca0d570a51a9dea915b90d678346
Payer      83de1a7075f190a1
Authorizers [83de1a7075f190a1]

Proposal Key:
  Address      83de1a7075f190a1
  Index        1
  Sequence     1
```

No Payload Signatures

```
Envelope Signature 0: 83de1a7075f190a1
Signatures (minimized, use --include signatures)
```

```
Code (hidden, use --include code)
```

```
Payload (hidden, use --include payload)
```

Arguments

Filename

```
- Name: <filename>
```

- Valid Input: file name.

The first argument is the filename containing the transaction RLP.

Flags

Include Fields

- Flag: --include
- Valid inputs: code, payload, signatures

Specify fields to include in the result output. Applies only to the text output.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify the format of the command results.

Save

- Flag: --save
- Short Flag: -s
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# get-transactions.md:
```

```
---
```

```
title: Get a Transaction
```

```
description: How to get a Flow transaction from the command line
```

```
sidebarposition: 2
```

```
---
```

The Flow CLI provides a command to fetch a transaction that was previously submitted to an Access API.

```
shell
flow transactions get <txid>
```

Example Usage

```

shell
> flow transactions get
40bc4b100c1930c61381c22e0f4c10a7f5827975ee25715527c1061b8d71e5aa --
network mainnet

Status           ✓ SEALED
ID              40bc4b100c1930c61381c22e0f4c10a7f5827975ee25715527c1061b8d71e5aa
Payer            18eb4ee6b3c026d2
Authorizers [18eb4ee6b3c026d2]

Proposal Key:
  Address 18eb4ee6b3c026d2
  Index   11
  Sequence 17930

Payload Signature 0: 18eb4ee6b3c026d2
Payload Signature 1: 18eb4ee6b3c026d2
Envelope Signature 0: 18eb4ee6b3c026d2
Signatures (minimized, use --include signatures)

Events:
  Index 0
  Type A.1654653399040a61.FlowToken.TokensWithdrawn
  Tx ID 40bc4b100c1930c61381c22e0f4c10a7f5827975ee25715527c1061b8d71e5aa
  Values
    - amount (UFix64): 0.00100000
    - from ({})?: 18eb4ee6b3c026d2

  Index 1
  Type A.1654653399040a61.FlowToken.TokensDeposited
  Tx ID 40bc4b100c1930c61381c22e0f4c10a7f5827975ee25715527c1061b8d71e5aa
  Values
    - amount (UFix64): 0.00100000
    - to ({})?: 5068e27f275c546c

  Index 2
  Type A.18eb4ee6b3c026d2.PrivateReceiverForwarder.PrivateDeposit
  Tx ID 40bc4b100c1930c61381c22e0f4c10a7f5827975ee25715527c1061b8d71e5aa
  Values
    - amount (UFix64): 0.00100000
    - to ({})?: 5068e27f275c546c

Code (hidden, use --include code)

Payload (hidden, use --include payload)

```

Arguments

Transaction ID

- Name: <txid>
- Valid Input: transaction ID.

The first argument is the ID (hash) of the transaction.

Flags

Include Fields

- Flag: --include
- Valid inputs: code, payload, signatures

Specify fields to include in the result output. Applies only to the text output.

Wait for Seal

- Flag: --sealed
- Default: false

Indicate whether to wait for the transaction to be sealed before displaying the result.

Exclude Fields

- Flag: --exclude
- Valid inputs: events

Specify fields to exclude from the result output. Applies only to the text output.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge

several configuration files.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

send-signed-transactions.md:

```
---
title: Send Signed Transaction
description: How to send a signed Flow transaction from the command line
sidebarposition: 6
---
```

The Flow CLI provides a command to send signed transactions to any Flow Access API.

Use this functionality in the following order:

1. Use the build command to build the transaction.
2. Use the sign command to sign with each account specified in the build process.
3. Use this command (send-signed) to submit the signed transaction to the Flow network.

```
shell
> flow transactions send-signed <signed transaction filename>
```

Example Usage

```
shell
> flow transactions send-signed ./signed.rlp

Status          ✓ SEALED
ID              528332aceb288cdfe4d11d6522aa27bed94fb3266b812cb350eb3526ed489d99
Payer           f8d6e0586b0a20c7
Authorizers     [f8d6e0586b0a20c7]

Proposal Key:
  Address f8d6e0586b0a20c7
  Index   0
  Sequence 0
```

No Payload Signatures

```
Envelope Signature 0: f8d6e0586b0a20c7
Signatures (minimized, use --include signatures)
```

Events: None

```
Code (hidden, use --include code)
Payload (hidden, use --include payload)
```

Arguments

Signed Code Filename

- Name: signed transaction filename
- Valid inputs: Any filename and path valid on the system.

The first argument is a path to a Cadence file containing the transaction to be executed.

Flags

Include Fields

- Flag: --include
- Valid inputs: code, payload

Specify fields to include in the result output. Applies only to the text output.

Exclude Fields

- Flag: --exclude
- Valid inputs: events

Specify fields to exclude from the result output. Applies only to the text output.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the --network flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.
You can use the `-f` flag multiple times to merge

```
several configuration files.
```

Version Check

- Flag: --skip-version-check
- Default: false

```
Skip version check during start up to speed up process for slow connections.
```

```
# send-transactions.md:
```

```
---
```

```
title: Send a Transaction
description: How to send a Flow transaction from the command line
sidebarposition: 1
---
```

```
The Flow CLI provides a command to sign and send transactions to any Flow Access API.
```

```
shell
flow transactions send <code filename> [<argument> <argument>...] [flags]
```

Example Usage

```
shell
> flow transactions send ./tx.cdc "Hello"
```

```
Status      ✓ SEALED
ID          b04b6bcc3164f5ee6b77fa502c3a682e0db57fc47e5b8a8ef3b56aae50ad49c8
Payer        f8d6e0586b0a20c7
Authorizers  [f8d6e0586b0a20c7]
```

Proposal Key:

```
Address f8d6e0586b0a20c7
Index   0
Sequence 0
```

No Payload Signatures

```
Envelope Signature 0: f8d6e0586b0a20c7
Signatures (minimized, use --include signatures)
```

```
Events:     None
```

```
Code (hidden, use --include code)
```

```
Payload (hidden, use --include payload)
```

```
Multiple arguments example:  
shell  
> flow transactions send tx1.cdc Foo 1 2 10.9 0x1 '[123,222]' '["a","b"]'
```

Transaction code:

```
transaction(a: String, b: Int, c: UInt16, d: UFix64, e: Address, f: [Int], g: [String]) {  
    prepare(authorizer: &Account) {}  
}
```

In the above example, the flow.json file would look something like this:

```
json  
{  
    "accounts": {  
        "my-testnet-account": {  
            "address": "a2c4941b5f3c7151",  
            "key": "12c5dfde...bb2e542f1af710bd1d40b2"  
        }  
    }  
}
```

JSON arguments from a file example:

```
shell  
> flow transactions send tx1.cdc --args-json "$(cat args.json)"
```

Arguments

Code Filename

- Name: code filename
- Valid inputs: Any filename and path valid on the system.

The first argument is a path to a Cadence file containing the transaction to be executed.

Arguments

- Name: argument
- Valid inputs: valid cadence values matching argument type in transaction code.

Input arguments values matching corresponding types in the source code and passed in the same order.

You can pass a nil value to optional arguments by sending the transaction like this: flow transactions send tx.cdc nil.

Flags

Include Fields

- Flag: --include

- Valid inputs: code, payload

Specify fields to include in the result output. Applies only to the text output.

Code

- Flag: --code

⚠ No longer supported: use filename argument.

Results

- Flag: --results

⚠ No longer supported: all transactions will provide result.

Exclude Fields

- Flag: --exclude
- Valid inputs: events

Specify fields to exclude from the result output. Applies only to the text output.

Signer

- Flag: --signer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used to sign the transaction.

Proposer

- Flag: --proposer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used as proposer in the transaction.

Payer

- Flag: --payer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used as payer in the transaction.

Authorizer

- Flag: `--authorizer`
- Valid inputs: the name of a single or multiple comma-separated accounts defined in the configuration (`flow.json`)

Specify the name of the account(s) that will be used as authorizer(s) in the transaction. If you want to provide multiple authorizers separate them using commas (e.g. `alice,bob`)

Arguments JSON

- Flag: `--args-json`
- Valid inputs: arguments in JSON-Cadence form.
- Example: `flow transactions send ./tx.cdc '[{"type": "String", "value": "Hello World"}]`

Arguments passed to the Cadence transaction in Cadence JSON format. Cadence JSON format contains type and value keys and is documented here.

Gas Limit

- Flag: `--gas-limit`
- Valid inputs: an integer greater than zero.
- Default: 1000

Specify the gas limit for this transaction.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the command. This flag overrides any host defined by the `--network` flag.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: a case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify the format of the command results.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: a path in the current filesystem.

Specify the filename where you want the result to be saved

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see during command execution.

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: a path in the current filesystem.
- Default: `flow.json`

Specify the path to the `flow.json` configuration file.

You can use the `-f` flag multiple times to merge several configuration files.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# sign-transaction.md:
```

```
---
```

```
title: Sign a Transaction
```

```
description: How to sign a Flow transaction from the command line
```

```
sidebarposition: 5
```

```
---
```

The Flow CLI provides a command to sign transactions with options to specify authorizer accounts, payer accounts and proposer accounts.

Use this functionality in the following order:

1. Use the build command to build the transaction.
2. Use this command (sign) to sign with each account specified in the build process.
3. Use the send-signed command to submit the signed transaction to the Flow network.

```
shell
flow transactions sign <built transaction filename>
```

Example Usage

```
shell
> flow transactions sign ./built.rlp --signer alice \
   --filter payload --save signed.rlp

Hash
  b03b18a8d9d30ff7c9f0fd8aa80fcab242c2f36eedb687dd9b368326311fe376
Payer      f8d6e0586b0a20c7
Authorizers [f8d6e0586b0a20c7]
```

Proposal Key:

```
  Address f8d6e0586b0a20c7
  Index   0
  Sequence 6
```

No Envelope Signatures

Payload Signature 0:

```
  Address f8d6e0586b0a20c7
  Signature
    b5b1dfed2a899037...164e1b224a7ac924018e7033b68b0df86769dd54
  Key Index 0
```

Arguments (1):

```
- Argument 0: {"type": "String", "value": "Meow"}
```

Code

```
transaction(greeting: String) {
    let guest: Address

    prepare(authorizer: &Account) {
        self.guest = authorizer.address
    }

    execute {
        log(greeting.concat(",")).concat(self.guest.toString()))
    }
}
```

Payload:

```
f90184f...a199bfd9b837a11a0885f9104b54014750f5e3e5bfe4a5795968b0df86769dd
54c0
```

Arguments

Built Transaction Filename or Remote Server URL

- Name: built transaction filename | --from-remote-url <url>
- Valid inputs: Any filename and path valid on the system or --from-remote-url flag and fully qualified remote server url.

Specify the filename containing valid transaction payload that will be used for signing.

To be used with the flow transaction build command.

When --from-remote-url flag is used the value needs to be a fully qualified url to transaction RLP

Example: flow transaction sign --from-remote-url
https://fully/qualified/url --signer alice

Flags

From Remote Url

- Flag: --from-remote-url
- Valid input: http(s)://fully/qualified/server/url

Specify this flag with a fully qualified url to transaction RLP. The RLP will be fetched from server then signed. The resulting signed RLP is then posted to the remote url. This feature is to support protocol level multiple signature transaction coordination between multiple signers.

Note: --yes flag is not supported and will fail sign command when this flag is used. This forces the user to verify the cadence code.

Include Fields

- Flag: --include
- Valid inputs: code, payload, signatures

Specify fields to include in the result output. Applies only to the text output.

Signer

- Flag: `--signer`
- Valid inputs: the name of an account defined in the configuration (`flow.json`)

Specify the name of the account that will be used to sign the transaction.

Host

- Flag: `--host`
- Valid inputs: an IP address or hostname.
- Default: `127.0.0.1:3569` (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the commands.

Network Key

- Flag: `--network-key`
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: `--network`
- Short Flag: `-n`
- Valid inputs: the name of a network defined in the configuration (`flow.json`)
- Default: `emulator`

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify in which format you want to display the result.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: valid filename

Specify the filename where you want the result to be saved.

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see while command execution.

Configuration

- Flag: `--conf`
- Short Flag: `-f`
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using `-f` flag multiple times.

Version Check

- Flag: `--skip-version-check`
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# signature-generate.md:
```

```
---
```

```
title: Generate a Signature
```

```
description: How to generate a new signature from the command line
```

```
---
```

Generate a signature using the private key of the signer account.

```
shell
```

```
flow signatures generate <message>
```

⚠ Make sure the account you want to use for signing is saved in the `flow.json` configuration.

The address of the account is not important, just the private key.

Example Usage

```
shell
> flow signatures generate 'The quick brown fox jumps over the lazy dog'
--signer alice

Signature          b33eabfb05d374b...f09929da96f5beec167fd1f123ec
Message           The quick brown fox jumps over the lazy dog
Public Key        0xc92a7c...042c4025d241fd430242368ce662d39636987
Hash Algorithm    SHA3256
Signature Algorithm ECDSAP256
```

Arguments

Message

- Name: message

Message used for signing.

Flags

Signer

- Flag: --signer
- Valid inputs: the name of an account defined in the configuration (flow.json)

Specify the name of the account that will be used to sign the transaction.

Filter

- Flag: --filter
- Short Flag: -x
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: --output
- Short Flag: -o
- Valid inputs: json, inline

Specify in which format you want to display the result.

Save

- Flag: --save
- Short Flag: -s

- Valid inputs: valid filename

Specify the filename where you want the result to be saved.

Log

- Flag: --log
- Short Flag: -l
- Valid inputs: none, error, debug
- Default: info

Specify the log level. Control how much output you want to see while command execution.

Configuration

- Flag: --config-path
- Short Flag: -f
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using -f flag multiple times.

Version Check

- Flag: --skip-version-check
- Default: false

Skip version check during start up to speed up process for slow connections.

```
# signature-verify.md:
```

```
---
title: Verify Signature
description: How to verify a signature from the command line
---
```

Verify validity of a signature based on provided message and public key of the signature creator.

```
shell
flow signatures verify <message> <signature> <public key>
```

Example Usage

```
shell
> flow signatures verify
```

```
'The quick brown fox jumps over the lazy dog'

b1c9eff5d829fdeaf2dad6308fc8033e3b8875bc185ef804ce5d0d980545ef5be0f98b47a
fc979d12272d257ce13c4b490e431bfcada485cb1d2e3f209be8d07

0xc92a7c72a78f8f046a79f8a5fe1ef72424258a55eb869f13e6133301d64ad025d3362d5
df9e7c82289637af1431042c4025d241fd430242368ce662d39636987

Valid          true
Message        The quick brown fox jumps over the lazy dog
Signature
b1c9eff5d829fdeaf2...7ce13c4b490ead485cb1d2e3f209be8d07
Public Key      c92a7c72a78...1431042c4025d241fd430242368ce662d39636987
Hash Algorithm  SHA3256
Signature Algorithm ECDSAP256
```

Arguments

Message
- Name: message

Message data used for creating the signature.

Signature
- Name: signature

Message signature that will be verified.

Public Key
- Name: public key

Public key of the private key used for creating the signature.

Flags

Public Key Signature Algorithm

- Flag: --sig-algo
- Valid inputs: "ECDSAP256", "ECDSAssecp256k1"

Specify the ECDSA signature algorithm of the key pair used for signing.

Flow supports the secp256k1 and P-256 curves.

Public Key Hash Algorithm

- Flag: --hash-algo
- Valid inputs: "SHA2256", "SHA3256"
- Default: "SHA3256"

Specify the hash algorithm of the key pair used for signing.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify in which format you want to display the result.

Save

- Flag: `--save`
- Short Flag: `-s`
- Valid inputs: valid filename

Specify the filename where you want the result to be saved.

Log

- Flag: `--log`
- Short Flag: `-l`
- Valid inputs: `none`, `error`, `debug`
- Default: `info`

Specify the log level. Control how much output you want to see while command execution.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# snapshot-save.md:  
---  
title: Snapshot Save  
description: How to save a protocol snapshot from the command line  
---
```

The FLOW CLI provides a command to save the latest finalized protocol state snapshot

```
shell
flow snapshot save <output path>
```

Example Usage

```
shell
flow snapshot save /tmp/snapshot.json --network testnet
```

Example response

```
shell
snapshot saved: /tmp/snapshot.json
```

Arguments

Output Path

- Name: output path
- Valid Input: any valid string path

Output path where the protocol snapshot JSON file will be saved.

Flags

Host

- Flag: --host
- Valid inputs: an IP address or hostname.
- Default: 127.0.0.1:3569 (Flow Emulator)

Specify the hostname of the Access API that will be used to execute the commands.

Network Key

- Flag: --network-key
- Valid inputs: A valid network public key of the host in hex string format

Specify the network public key of the Access API that will be used to create a secure GRPC client when executing the command.

Network

- Flag: --network
- Short Flag: -n
- Valid inputs: the name of a network defined in the configuration (flow.json)
- Default: emulator

Specify which network you want the command to use for execution.

Filter

- Flag: `--filter`
- Short Flag: `-x`
- Valid inputs: case-sensitive name of the result property.

Specify any property name from the result you want to return as the only value.

Output

- Flag: `--output`
- Short Flag: `-o`
- Valid inputs: `json`, `inline`

Specify in which format you want to display the result.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

```
# tools.md:
```

```
---
```

```
title: Development Tools
```

```
description: How to start development tools using the Flow CLI
```

```
---
```

The Flow CLI integrates different development tools, which can now be easily started and managed from a single place.

Currently the CLI supports starting:

- Flow Development Wallet

Flow Development Wallet

The Flow Dev Wallet is a mock Flow wallet that simulates the protocols used by FCL to interact with the Flow blockchain on behalf of simulated user accounts.

Be sure you have the emulator running before starting this command. You can start it using the `flow emulator` command.

```
shell
flow dev-wallet
```

⚠ This project implements an FCL compatible interface, but should not be used as a reference for building a production grade wallet.

After starting dev-wallet, you can set your fcl config to use it like below:

```
javascript
import  as fcl from "@onflow/fcl"

fcl.config()
  // Point App at Emulator
  .put("accessNode.api", "http://localhost:8080")
  // Point FCL at dev-wallet (default port)
  .put("discovery.wallet", "http://localhost:8701/fcl/authn")
```

You can read more about setting up dev-wallet at [Flow Dev Wallet Project](#)

Flags

Port

- Flag: `--port`
- Valid inputs: Number
- Default: 8701

Port on which the dev wallet server will listen on.

Emulator Host

- Flag: `--emulator-host`
- Valid inputs: a hostname
- Default: `http://localhost:8080`

Specifies the host configuration for dev wallet

Configuration

- Flag: `--config-path`
- Short Flag: `-f`
- Valid inputs: valid filename

Specify a filename for the configuration files, you can provide multiple configuration files by using `-f` flag multiple times.

Specify a filename for the configuration files, you can provide multiple configuration files by using `-f` flag multiple times.

Version Check

- Flag: `--skip-version-check`
- Default: `false`

Skip version check during start up to speed up process for slow connections.

index.md:

```
---
```

title: Flow Dev Wallet
sidebarlabel: Flow Dev Wallet

The Flow Dev Wallet is a mock Flow wallet that simulates the protocols used by FCL to interact with the Flow blockchain on behalf of simulated user accounts.

:::warning [IMPORTANT]

This project implements an FCL compatible interface, but should not be used as a reference for building a production grade wallet.

This project should only be used in aid of local development against a locally run instance of the Flow blockchain like the Flow emulator, and should never be used in conjunction with Flow Mainnet, Testnet, or any other instances of Flow.

:::

:::info

To see a full list of Flow compatible wallets visit Wallets page

:::

Getting Started

Before using the dev wallet, you'll need to start the Flow emulator.

Install the flow-cli

The Flow emulator is bundled with the Flow CLI. Instructions for installing the CLI can be found here: [flow-cli/install/](#)

Create a `flow.json` file

Run this command to create flow.json file (typically in your project's root directory):

```
sh
flow init --config-only
```

Start the Emulator

Start the Emulator and deploy the contracts by running the following command from the directory containing flow.json in your project:

```
sh
flow emulator start
flow project deploy --network emulator
```

Configuring Your JavaScript Application

The Flow Dev Wallet is designed to be used with @onflow/fcl version 1.0.0 or higher. The FCL package can be installed with: npm install @onflow/fcl or yarn add @onflow/fcl.

To use the dev wallet, configure FCL to point to the address of a locally running Flow emulator and the dev wallet endpoint.

```
javascript
import  as fcl from '@onflow/fcl';

fcl
  .config()
  // Point App at Emulator REST API
  .put('accessNode.api', 'http://localhost:8888')
  // Point FCL at dev-wallet (default port)
  .put('discovery.wallet', 'http://localhost:8701/fcl/authn');
```

:::info

For a full example refer to Authenticate using FCL snippet

:::

Test harness

It's easy to use this FCL harness app as a barebones app to interact with the dev-wallet during development:

Navigate to <http://localhost:8701/harness>

Wallet Discovery

Wallet Discovery offers a convenient modal and mechanism to authenticate users and connects to all wallets available in the Flow ecosystem.

The following code from Emerald Academy can be added to your React app to enable Wallet Discovery:

```
javascript
import { config, authenticate, unauthenticate, currentUser } from
'@onflow/fcl';
import { useEffect, useState } from 'react';

const fclConfigInfo = {
  emulator: {
    accessNode: 'http://127.0.0.1:8888',
    discoveryWallet: 'http://localhost:8701/fcl/authn',
    discoveryAuthInclude: [],
  },
  testnet: {
    accessNode: 'https://rest-testnet.onflow.org',
    discoveryWallet: 'https://fcl-discovery.onflow.org/testnet/authn',
    discoveryAuthnEndpoint:
      'https://fcl-discovery.onflow.org/api/testnet/authn',
    // Adds in Dapper + Ledger
    discoveryAuthInclude: ['0x82ec283f88a62e65', '0x9d2e44203cb13051'],
  },
  mainnet: {
    accessNode: 'https://rest-mainnet.onflow.org',
    discoveryWallet: 'https://fcl-discovery.onflow.org/authn',
    discoveryAuthnEndpoint: 'https://fcl-discovery.onflow.org/api/authn',
    // Adds in Dapper + Ledger
    discoveryAuthInclude: ['0xead892083b3e2c6c', '0xe5cd26afebe62781'],
  },
};

const network = 'emulator';

config({
  'walletconnect.projectId': 'YOURPROJECTID', // your WalletConnect
  project ID
  'app.detail.title': 'Emerald Academy', // the name of your DApp
  'app.detail.icon': 'https://academy.ecdao.org/favicon.png', // your
  DApps icon
  'app.detail.description': 'Emerald Academy is a DApp for learning
  Flow', // a description of your DApp
  'app.detail.url': 'https://academy.ecdao.org', // the URL of your DApp
  'flow.network': network,
  'accessNode.api': fclConfigInfo[network].accessNode,
  'discovery.wallet': fclConfigInfo[network].discoveryWallet,
  'discovery.authn.endpoint':
    fclConfigInfo[network].discoveryAuthnEndpoint,
  // adds in opt-in wallets like Dapper and Ledger
  'discovery.authn.include': fclConfigInfo[network].discoveryAuthInclude,
});

export default function App() {
  const [user, setUser] = useState({ loggedIn: false, addr: '' });
}
```

```

// So that the user stays logged in
// even if the page refreshes
useEffect(() => {
  currentUser.subscribe(setUser);
}, []);

return (
  <div className="App">
    <button onClick={authenticate}>Log In</button>
    <button onClick={unauthenticate}>Log Out</button>
    <p>{user.loggedIn ? Welcome, ${user.addr}! : 'Please log in.'}</p>
  </div>
);
}

```

Account/Address creation

You can create a new account by using the `&Account` constructor. When you do this, make sure to specify which account will pay for the creation fees by setting it as the payer.

The account you choose to pay these fees must have enough money to cover the cost. If it doesn't, the process will stop and the account won't be created.

```

cadence
transaction(publicKey: String) {
  prepare(signer: &Account) {
    let key = PublicKey(
      publicKey: publicKey.decodeHex(),
      signatureAlgorithm: SignatureAlgorithm.ECDSSAP256
    )
    let account = Account(payer: signer)
    account.keys.add(
      publicKey: key,
      hashAlgorithm: HashAlgorithm.SHA3256,
      weight: 1000.0
    )
  }
}

```

To create a new Flow account refer to these resources

- Create an Account with FCL snippet
- Create an Account in Cadence snippet

Get Flow Balance

Retrieving the token balance of a specific account involves writing a script to pull data from onchain. The user may have both locked tokens as

well as unlocked so to retrieve the total balance we would aggregate them together.

```
javascript
import  as fcl from '@onflow/fcl';
import  as t from '@onflow/types';
const CODE =
import "FungibleToken"
import "FlowToken"
import "LockedTokens"

access(all) fun main(address: Address): UFix64 {
    let account = getAccount(address)
    let unlockedVault = account
        .capabilities.get<&FlowToken.Vault>(/public/flowTokenBalance)
        .borrow()
    ?? panic("Could not borrow Balance reference to the Vault"
        .concat(" at path /public/flowTokenBalance!"))
    .concat(" Make sure that the account address is correct ")
    .concat("and that it has properly set up its account with a
FlowToken Vault.))

    let unlockedBalance = unlockedVault.balance
    let lockedAccountInfoCap = account
        .capabilities.get
        <&LockedTokens.TokenHolder>
        (LockedTokens.LockedAccountInfoPublicPath)
    if !(lockedAccountInfoCap!.check()) {
        return unlockedBalance
    }
    let lockedAccountInfoRef = lockedAccountInfoCap!.borrow()!
    let lockedBalance = lockedAccountInfoRef.getLockedAccountBalance()
    return lockedBalance + unlockedBalance
};

export const getTotalFlowBalance = async (address) => {
    return await fcl.decode(
        await fcl.send([fcl.script(CODE), fcl.args([fcl.arg(address,
t.Address)])]),
    );
};
```

Contributing

Releasing a new version of Dev Wallet is as simple as tagging and creating a release, a Github Action will then build a bundle of the Dev Wallet that can be used in other tools (such as CLI). If the update of the Dev Wallet is required in the CLI, a seperate update PR on the CLI should be created. For more information, please visit the [fcl-dev-wallet GitHub repository](#).

More

Additionally, consider exploring these resources:

- Guide to Creating a Fungible Token on Flow
- Tutorial on Fungible Tokens
- Faucets

```
# index.md:
```

```
---
```

```
title: Cadence VS Code Extension
```

```
---
```

This extension integrates Cadence, the resource-oriented smart contract programming language of Flow, into Visual Studio Code. It provides features like syntax highlighting, type checking, code completion, etc.

Note that most editing features (type checking, code completion, etc.) are implemented in the Cadence Language Server.

Features

- Syntax highlighting (including in Markdown code fences)
- Run the emulator, submit transactions, scripts from the editor

Installation

To install the extension, ensure you have VS Code installed and have configured the code command line interface.

Using the Flow CLI

The recommended way to install the latest released version is to use the Flow CLI.

```
shell script
brew install flow-cli
```

Check that it's been installed correctly.

```
shell script
flow version
```

Next, use the CLI to install the VS Code extension.

```
shell script
flow cadence install-vscode-extension
```

Restart VS Code and the extension should be installed!

Developing the Extension

Prerequisites

- Must have Typescript installed globally: `npm i -g typescript`

Getting Started

- Run the Typescript watcher: `tsc -watch -p ./`
- Launch the extension by pressing F5 in VSCode
- Manually reload the extension host when you make changes to TypeScript code

Configuration for Extension Host if Missing (`launch.json`):

```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "extensionHost",  
      "request": "launch",  
      "name": "Launch Extension",  
      "runtimeExecutable": "${execPath}",  
      "args": ["--extensionDevelopmentPath=${workspaceFolder}"],  
      "outFiles": ["${workspaceFolder}/out//.js"]  
    }  
  ]  
}
```

Building

If you are building the extension from source, you need to build both the extension itself and the Flow CLI (if you don't already have a version installed).

Unless you're developing the extension or need access to unreleased features,
you should use the Flow CLI install option (above). It's much easier!

If you haven't already, install dependencies.

```
shell script  
npm install
```

Next, build and package the extension.

```
shell script  
npm run package
```

This will result in a .vsix file containing the packaged extension.

Install the packaged extension.

```
shell script
code --install-extension cadence-.vsix
```

Restart VS Code and the extension should be installed!

```
# authorization-function.md:
```

Authorization Function

Overview

An Authorization Function is a function which enables the JS-SDK and FCL to know which Flow account fulfills which signatory role in a transaction and how to receive a signature on behalf of the supplied account.

How to Use an Authorization Function

An authorization function is a function that you may use in place of an authorization in the Flow JS-SDK and FCL. An authorization is a concept that is used when denoting a proposer, payer or authorizer for a transaction. An authorization can either be a data structure representing an authorization, or a function which when called returns an authorization called an Authorization Function. In this document we discuss the latter.

To use an Authorization Function, you specify that Authorization Function as the authorization for a proposer, payer or authorizer for a transaction.

```
> fcl.currentUser().authorization which is aliased to fcl.authz is itself
an authorization function. It tells the underlying js-sdk the current
users flow account will be used for the signatory role and supplies a
signing function that enables the application to request a signature from
the users wallet.
```

Example 1:

```
javascript
import as fcl from "@onflow/fcl"

const myAuthorizationFunction = ... // An Authorization Function

const response = fcl.send([
    fcl.transactiontransaction() { prepare(acct: &Account) {} execute {
        log("Hello, Flow!") },
    fcl.proposer(myAuthorizationFunction),
    fcl.payer(myAuthorizationFunction),
    fcl.authorizers([ myAuthorizationFunction ])
])
```

The builder functions, `fcl.proposer`, `fcl.payer` and `fcl.authorizations` each consume the Authorization Function and set it as the `resolve` field on the internal Account object it creates.

During the resolve phase of the Flow JS-SDK and FCL, when `resolveAccounts` is called, the `resolve` field on each internal Account object is called, which means each Authorization Function is called appropriately and the account is resolved into the data structure the `authorizationFunction` returns. These accounts are then deduped based on the a mix of the `addr`, `keyId` and `tempId` so that only a single signature request happens per address `keyId` pair. When `resolveSignatures` is called the signing function for each address `keyId` pair is called returning a composite signature for each signatory role.

How to Create An Authorization Function

Fortunately, creating an Authorization Function is relatively straight forward.

An Authorization Function needs to be able to do at minimum two things.

- Who will sign -- Know which account is going to sign and the `keyId` of the key it will use to sign
- How they sign -- Know how to get a signature for the supplied account and key from the first piece.

The Authorization Function has a concept of an account. An account represent a possible signatory for the transaction, it includes the who is signing as well as the how it will be signed. The Authorization Function is passed an empty Account and needs to return an Account, your job when making an Authorization Function is mostly to fill in this Account with the information so that the account you want to sign things can.

Lets say we knew up front the account, `keyId` and had a function that could sign things.

```
javascript
const ADDRESS = "0xba1132bc08f82fe2"
const KEYID = 1 // this account on testnet has three keys, we want the
one with an index of 1 (has a weight of 1000)
const sign = msg => { / ... returns signature (for the key above) for
supplied message ... / }
```

Our Authorization Function becomes about filling things in:

```
Example 2:
javascript
const authorizationFunction = async (account) => {
  // authorization function need to return an account
  return {
    ...account, // bunch of defaults in here, we want to overload some of
them though
```

```
tempId: ${ADDRESS}-${KEYID}, // tempIds are more of an advanced
topic, for 99% of the times where you know the address and keyId you will
want it to be a unique string per that address and keyId
    addr: ADDRESS, // the address of the signatory
    keyId: Number(KEYID), // this is the keyId for the accounts
registered key that will be used to sign, make extra sure this is a
number and not a string
    signingFunction: async signable => {
        // Singing functions are passed a signable and need to return a
composite signature
        // signable.message is a hex string of what needs to be signed.
        return {
            addr: ADDRESS, // needs to be the same as the account.addr
            keyId: Number(KEYID), // needs to be the same as account.keyId,
once again make sure its a number and not a string
            signature: sign(signable.message), // this needs to be a hex
string of the signature, where signable.message is the hex value that
needs to be signed
        }
    }
}
```

Async stuff

Both the Authorization Function, and the accounts Signing Function can be asynchronous. This means both of these functions can go and get the information needed elsewhere. Say each of your users had a userId. From this userId say you had an api call that could return the corresponding address and key that is needed for the Authorization Functions account. You could also have another endpoint that when posted the signable (includes what needs to be signed) and the userId it can return with the composite signature if your api decides its okay to sign (the signable has all sorts of info to help you decide). An authorization function that can do that could look something like this.

Example 3:

```
javascript
const getAccount = (userId) => fetch(`/api/user/${userId}/account`).then(d => d.json())
const getSignature = (userId, signable) = fetch(`/api/user/${userId}/sign`, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(signable),
})
function authz (userId) {
  return async function authorizationFunction (account) {
    const {addr, keyId} = await getAccount(userId)

    return {
      ...account,
```

```

        tempId: ${addr}-${keyId},
        addr: addr,
        keyId: Number(keyId),
        signingFunction: signable => {
            return getSignature(userId, signable)
        }
    }
}

```

The above Example 3 is the same as Example 2, but the information is gathered during the execution of the authorization function based on the supplied user id.

How to create a Signing Function

Creating a signing function is also relatively simple!

To create a signing function you specify a function which consumes a payload and returns a signature data structure.

Example 3:

```

javascript
const signingFunction = ({
    message, // The encoded string which needs to be used to produce the
    signature.
    addr, // The address of the Flow Account this signature is to be
    produced for.
    keyId, // The keyId of the key which is to be used to produce the
    signature.
    roles: {
        proposer, // A Boolean representing if this signature to be produced
        for a proposer.
        authorizer, // A Boolean representing if this signature to be
        produced for a authorizer.
        payer, // A Boolean representing if this signature to be produced for
        a payer.
    },
    voucher, // The raw transactions information, can be used to create the
    message for additional safety and lack of trust in the supplied message.
}) => {
    return {
        addr, // The address of the Flow Account this signature was produced
        for.
        keyId, // The keyId for which key was used to produce the signature.
        signature: produceSignature(message) // The hex encoded string
        representing the signature of the message.
    }
}

# custodial.md:

```

Introduction

A Wallet Provider handles Authentications and Authorizations. They play a very important role of being the place the users control their information and approve transactions.

One of FCLs core ideals is for the user to be in control of their data, a wallet provider is where many users will do just that.

FCL has been built in a way that it doesn't need to know any intimate details about a wallet provider up front, they can be discovered when the users wishes to let the dapp know about them. This gives us a concept we have been calling Bring Your Own Identity.

Identity

Conceptually, FCL thinks of identity in two ways: Public and Private.

Public identity will be stored on chain as a resource, it will be publicly available to anyone that knows the Flow Address for the account.

In FCL getting a users public identity will be as easy as:

```
javascript
import {user} from "@onflow/fcl"

const identity = await user(flowAddress).snapshot()
//           ^
//           ----- The public identity for flowAddress

const unsub = user(flowAddress).subscribe(identity =>
  console.log(identity)
  //                                     ^
  //                                     ----- The public identity
for flowAddress
```

Private identity will be stored by the Wallet Provider, it will only be available to the currentUser.

In FCL getting the currentUser's identity will fetch both the public and the private identities, merging the private into the public.

Private info needs to be requested via scopes before the challenge step, more on that later.

We highly recommend Wallet Providers let the user see what scopes are being requested, and decide what scopes to share with the dapp.

Consumers of identities in FCL should always assume all data is optional, and should store as little as possible, FCL will make sure the users always see the latest.

```
javascript
import {config, currentUser, authenticate} from "@onflow/fcl"
```

```

config.put("challenge.scope", "email") // request the email scope

const unsub = currentUser().subscribe(identity => console.log(identity))
// ^----- The private identity for
// the currentUser

authenticate() // trigger the challenge step (authenticate the user via a
wallet provider)

```

Identity Data

- All information in Identities are optional and may not be there.
- All values can be stored on chain, but most probably shouldn't be.

We would love to see Wallet Providers enable the user to control the following info publicly, sort of a public profile starter kit if you will.

FCL will always publicly try to fetch these fields when asked for a users information and it will be up to the Wallet provider to make sure they are there and keep them up to date if the user wants to change them.

- name -- A human readable name/alias/nym for a dapp users display name
- avatar -- A fully qualified url to a smaller image used to visually represent the dapp user
- cover -- A fully qualified url to a bigger image, could be used by the dapp for personalization
- color -- A 6 character hex color, could be used by the dapp for personalization
- bio -- A small amount of text that a user can use to express themselves

If we can give dapp developers a solid foundation of usable information that is in the direct control of the users from the very start, which we believe the above fields would do, our hopes are they can rely more on the chain and will need to store less in their own database.

Private data on the other hand has more use cases than general data. It is pretty easy to imagine ordering something and needing information like contact details and where to ship something.

Eventually we would love to see that sort of thing handled completely on-chain, securely, privately and safely, but in the interim it probably means storing a copy of data in a database when its needed, and allowed by a user.

The process of a dapp receiving private data is as follows:

1. The dapp requests the scopes they want up front
fcl.config().put("challenge.scope", "email+shippingAddress").
2. The User authenticates fcl.authenticate() and inside the Wallet Providers authentication process decides its okay for the dapp to know

both the email and the shippingAddress. The User should be able to decide which information to share, if any at all.

3. When the dapp needs the information they can request it from FCLs current cache of data, if it isn't there the dapp needs to be okay with that and adjust accordingly.

Below are the scopes we are thinking of supporting privately:
FCL will only publicly and privately try to fetch these when specified up front by a dapp.

- email
- fullName
- phone
- textMessage
- address
- shippingAddress
- location
- publicKey

All of the above are still subject to change as it is still early days, we would like to work closely with Wallet Providers to produce a robust, detailed and consistent spec regarding scopes. Feedback and thoughts are always welcome.

Authentication Challenge

Authentication can happen one of two ways:

- Iframe Flow
- Redirection Flow

As a Wallet Provider you will be expected to register a URL endpoint (and some other information) with a handshake service (FCL will be launching with one in which registration happens on chain and is completely open source (Apache-2.0 license)).

This registered URL will be what is shown inside the iFrame or where the dapp users will be redirected.

For the remainder of this documentation we will refer to it as the Authentication Endpoint and pair it with the GET
<https://provider.com/flow/authentication> route.

The Authentication Endpoint will receive the following data as query params:

- 16n (required) -- location (origin) of dapp
- nonce (required) -- a random string supplied by the FCL
- scope (optional) -- the scopes requested by the dapp
- redirect (optional) -- where to redirect once the authentication challenge is complete

```
GET https://provider.com/flow/authenticate
?16n=https%3A%2F%2Fdapp.com
&nonce=asdfasfasdf
```

```
&scope=email+shippingAddress  
&redirect=https%3A%2Fdapp.com%2Fflow%2Fcallback
```

The values will use javascripts encodeURIComponent function and scopes will be + delimited.

We can tell that this challenge is using the Redirect Flow because of the inclusion of the redirect query param.

The Iframe Flow will still need to be supported as it will be the default flow for dapps.

At this point its on the Wallet Provider to do their magic and be confident enough that the user is who they say they are.

The user should then be shown in some form what the dapp is requesting via the scopes and allow them to opt in or out of anything they want.

Once the Wallet Provider is ready to hand back control to the dapp and FCL it needs to complete the challenge by redirecting or emitting a javascript postMessage event.

Redirecting will look like this:

```
GET https://dapp.com/flow/callback  
param above  
?16n=https%3A%2Fdapp.com  
&nonce=asdfasdfsadfa  
above  
&addr=0xab4U9KMf  
account (if available) -- will be used to fetch public identity  
information and hooks  
&paddr=0xhMgqTff86  
Providers account -- will be used to fetch provider information  
&code=afseasdfsadf  
the Wallet Provider, FCL will use this token when requesting private  
information and hooks, can be any url safe value  
&exp=1650400809517  
when the code expires, a value  
of 0 will be considered as never expires  
&hks==https%3A%2Fprovider.com%2Fhooks a URL where FCL can request  
the private information and hooks
```

supplied by the redirect query
the 16n supplied by FCL above
the nonce supplied by FCL
address for the users flow
a token supplied to FCL from
when the code expires, a value
a URL where FCL can request

Iframe will look like this:

```
javascript  
parent.postMessage()  
{  
  type: "FCL::CHALLENGE::RESPONSE", // used by FCL to know what kind of  
  message this is  
  addr: "0xab4U9KMf",  
  paddr: "0xhMgqTff86",  
  code: "afseasdfsadf",  
  exp: 1650400809517,  
  hks: "https://provider.com/hooks",
```

```

        nonce: "asdfasdfasdfs",
        l6n: decodeURIComponent(l6n),
    },
    decodeURIComponent(l6n)
)

```

FCL should now have everything it needs to collect the Public, Private and Wallet Provider Info.

The Wallet Provider info will be on chain so its not something that needs to be worried about here by the Wallet Provider.

What does need to be worried about handling the hooks request which was supplied to FCL via the hks value in the challenge response <https://provider.hooks>.

The hooks request will be to the hks value supplied in the challenge response. The request will also include the code as a query param

```
GET https://povider.com/hooks
?code=afseasdfsadf
```

This request needs to happen for a number of reasons.

- If it fails FCL knows something is wrong and will attempt to re-authenticate.
- If it succeeds FCL knows that the code it has is valid.
- It creates a direct way for FCL to "verify" the user against the Wallet Provider.
- It gives FCL a direct way to get Private Identity Information and Hooks.
- The code can be passed to the backend to create a back-channel between the backend and the Wallet Provider.

When users return to a dapp, if the code FCL stored hasn't expired, FCL will make this request again in order to stay up to date with the latest information. FCL may also intermittently request this information before some critical actions.

The hooks request should respond with the following JSON

```
javascript
const privateHooks = {
    addr: "0xab4U9KMf",           // the flow address this user is using for
    the dapp
    keyId: 3,                     // the keyId the user wants to use when
    authorizing transaction
    identity: {                   // the identity information fcl always wants
        if its there, will be deep merged into public info
        name: "Bob the Builder",
        avatar: "https://avatars.onflow.org/avatar/0xab4U9KMf.svg"
        cover: "https://placekittens.com/g/900/300",
        color: "cccc00",
    }
}
```

```

        bio: "",
    },
    scoped: { // the private info request in the original
challenge
        email: "bob@bob.bob", // the user said it was okay for the dapp to
know the email
        shippingAddress: null, // the user said it was NOT okay for the dapp
to know the shippingAddress
    },
    provider: {
        addr: "0xhMgqTff86", // the flow address for the wallet provider
(used in the identity composite id)
        pid: 2345432, // the wallet providers internal id for the user
(used in the identity composite id)
        name: "Super Wallet",
        icon: "https://provider.com/assets/icon.svg",
        authn: "https://provider.com/flow/authenticate",
    }
}

```

When FCL requested the Public info from the chain it is expecting something like this.

It will be on the Wallet Provider to keep this information up to date.

```

javascript
const publicHooks = {
    addr: "0xab4U9KMf",
    keyId: 2,
    identity: {
        name: "Bob the Builder",
        avatar: "https://avatars.onflow.org/avatar/0xab4U9KMf.svg"
        cover: "https://placekittens.com/g/900/300",
        color: "cccc00",
        bio: "",
    },
    authorizations: [
        {
            id: 345324539,
            addr: "0xhMgqTff86",
            method: "HTTP/POST",
            endpoint: "https://provider.com/flow/authorize",
            data: {
                id: 2345432
            }
        }
    ]
}

```

At this point FCL can be fairly confident who the currentUser is and is ready to initiate transactions the user can authorize.

Authorization

FCL will broadcast authorization requests to the Public and Private authorization hooks it knows for a User, in a process we call Asynchronous Remote Signing.

The core concepts to this idea are:

- Hooks tell FCL where to send authorization requests (Wallet Provider)
- Wallet Provider responds immediately with:
 - a back-channel where FCL can request the results of the authorization
 - some optional local hooks ways the currentUser can authorize
- FCL will trigger the local hooks if they are for the currentUser
- FCL will poll the back-channel requesting updates until an approval or denial is given

Below is the public authorization hook we received during the challenge above.

```
javascript
{
  id: 345324539,
  addr: "0xhMgqTff86",
  method: "HTTP/POST",
  endpoint: "https://provider.com/flow/authorize",
  data: {
    id: 2345432
  }
}
```

FCL will take that hook and do the following post request:

```
POST https://provider.com/flow/authorize
?id=2345432
---
{
  message: "...",      // what needs to be signed (needs to be converted
from hex to binary before signing)
  addr: "0xab4U9KMf", // the flow address that needs to sign
  keyId: 3,            // the flow account keyId for the private key that
needs to sign
  roles: {
    proposer: true,    // this account's sequence number will be used in
the transaction
    authorizer: true, // this transaction can "move" and "modify" the
accounts resources directly
    payer: true,      // this transaction will be paid for by this
account (also signifies that they are signing an envelopeMessage instead
of a payloadMessage)
  },
  interaction: {...} // needed to recreate the message if the Wallet
Provider wants to verify the message.
}
```

FCL is expecting something like this in response:

```
javascript
{
  status: "PENDING",
  reason: null,
  compositeSignature: null,
  authorizationUpdates: {
    method: "HTTP/POST",
    endpoint: "https://provider.com/flow/authorizations/4323",
  },
  local: [
    {
      method: "BROWSER/IFRAME",
      endpoint: "https://provider.com/authorizations/4324",
      width: "300",
      height: "600",
      background: "#ff0066"
    }
  ]
}
```

That local hook will be consumed by FCL, rendering an iframe with the endpoint as the src. If the user is already authenticated this screen could show them the Wallet Providers transaction approval process directly.

Because FCL isn't relying on any communication to or from the Iframe it can lock it down as much as possible, and remove it once the authorization is complete.

While displaying the local hook, it will request the status of the authorization from the authorizationUpdates hook.

```
POST https://provider.com/flow/authorizations/4323
```

Expecting a response that has the same structure as the origin but without the local hooks:

```
javascript
{
  status: "PENDING",
  reason: "",
  compositeSignature: null,
  authorizationUpdates: {
    method: "HTTP/POST",
    endpoint: "https://provider.com/flow/authorizations/4323",
  },
}
```

FCL will then follow the new authorizationUpdates hooks until the status changes to "APPROVED" or "DECLINED".

If the authorization is declined it should include a reason if possible.

```
javascript
{
  status: "DECLINED",
  reason: "They said no",
}
```

If the authorization is approved it should include a composite signature:

```
javascript
{
  status: "APPROVED",
  compositeSignature: {
    addr: "0xab4U9KMf", // the flow address that needs to sign
    keyId: 3,           // the flow account keyId for the private key
    that needs to sign
    signature: "..."   // binary signature of message encoded as hex
  }
}
```

FCL can now submit the transaction to the Flow blockchain.

TL;DR Wallet Provider

Register Provider with FCL Handshake and implement 5 Endpoints.

- GET flow/authenticate -> parent.postMessage(..., 16n)
- GET flow/hooks?code= -> { ...identityAndHooks }
- POST flow/authorize -> { status, reason, compositeSignature, authorizationUpdates, local }
- POST authorizations/:authorizationid
- GET authorizations/:authorizationid

!diagram showing current fcl authn and authz flow

index.md:

```
---
title: Wallet Provider Spec
sidebartitle: Draft v4
---
Status

- Last Updated: June 20th 2022
- Stable: Yes
- Risk of Breaking Change: Medium
- Compatibility: >= @onflow/fcl@1.0.0-alpha.0
```

Definitions

This document is written with the perspective that you who are reading this right now are an FCL Wallet Developer. All references to you in this doc are done with this perspective in mind.

Overview

Flow Client Library (FCL) approaches the idea of blockchain wallets on Flow in a different way than how wallets may be supported on other blockchains. For example, with FCL, a wallet is not necessarily limited to being a browser extension or even a native application on a users device. FCL offers wallet developers the flexibility and freedom to build many different types of applications. Since wallet applications can take on many forms, we needed to create a way for these varying applications to be able to communicate and work together.

FCL acts in many ways as a protocol to facilitate communication and configuration between the different parties involved in a blockchain application. An Application can use FCL to authenticate users, and request authorizations for transactions, as well as mutate and query the Blockchain. An application using FCL offers its Users a way to connect and select any number of Wallet Providers and their Wallet Services. A selected Wallet provides an Application's instance of FCL with configuration information about itself and its Wallet Services, allowing the User and Application to interact with them.

In the following paragraphs we'll explore ways in which you can integrate with FCL by providing implementations of various FCL services.

The following services will be covered:

- Authentication (Authn) Service
- Authorization (Authz) Service
- User Signature Service
- Pre-Authz Service

Service Methods

FCL Services are your way as a Wallet Provider of configuring FCL with information about what your wallet can do. FCL uses what it calls Service Methods to perform your supported FCL services. Service Methods are the ways FCL can talk to your wallet. Your wallet gets to decide which of these service methods each of your supported services use to communicate with you.

Sometimes services just configure FCL and that's it. An example of this can be seen with the Authentication Service and the OpenID Service. With those two services you are simply telling FCL "here is a bunch of info about the current user". (You will see that those two services both have a method: "DATA" field in them.)

Currently these are the only two cases that can be a data service.)

Other services can be a little more complex. For example, they might require a back and forth communication between FCL and the Service in question.

Ultimately we want to do this back and forth via a secure back-channel (https requests to servers), but in some situations that isn't a viable option, so there is also a front-channel option.

Where possible, you should aim to provide a back-channel support for services, and only fall back to a front-channel if absolutely necessary.

Back-channel communications use method: "HTTP/POST", while front-channel communications use method: "IFRAME/RPC", method: "POP/RPC", method: "TAB/RPC and method: "EXT/RPC".

Service Method	Front	Back
HTTP/POST	✗	✓
IFRAME/RPC	✓	✗
POP/RPC	✓	✗
TAB/RPC	✓	✗
EXT/RPC	✓	✗

It's important to note that regardless of the method of communication, the data that is sent back and forth between the parties involved is the same.

Protocol schema definitions

In this section we define the schema of objects used in the protocol. While they are JavaScript objects, only features supported by JSON should be used. (Meaning that conversion of an object to and from JSON should not result in any loss.)

For the schema definition language we choose TypeScript, so that the schema closely resembles the actual type definitions one would use when making an FCL implementation.

Note that currently there are no official type definitions available for FCL. If you are using TypeScript, you will have to create your own type definitions (possibly based on the schema definitions presented in this document).

Common definitions

In this section we introduce some common definitions that the individual object definitions will be deriving from.

First, let us define the kinds of FCL objects available:

```
typescript
type ObjectType =
  | 'PollingResponse'
  | 'Service'
  | 'Identity'
  | 'ServiceProvider'
  | 'AuthnResponse'
  | 'Signable'
```

```
| 'CompositeSignature'  
| 'OpenID'
```

The fields common to all FCL objects then can be defined as follows:

```
typescript  
interface ObjectBase<Version = '1.0.0'> {  
    fvsn: Version  
    ftype: ObjectType  
}
```

The fvsn field is usually 1.0.0 for this specification, but some exceptions will be defined by passing a different Version type parameter to ObjectBase.

All FCL objects carry an ftype field so that their types can be identified at runtime.

FCL objects

In this section we will define the FCL objects with each ObjectType.

We also define the union of them to mean any FCL object:

```
typescript  
type FclObject =  
    | PollingResponse  
    | Service  
    | Identity  
    | ServiceProvider  
    | AuthnResponse  
    | Signable  
    | CompositeSignature  
    | OpenID
```

PollingResponse

```
typescript  
interface PollingResponse extends ObjectBase {  
    ftype: 'PollingResponse'  
    status: 'APPROVED' | 'DECLINED' | 'PENDING' | 'REDIRECT'  
    reason: string | null  
    data?: FclObject  
    updates?: FclObject  
    local?: FclObject  
}
```

Each response back to FCL must be "wrapped" in a PollingResponse. The status field determines the meaning of the response:

- An APPROVED status means that the request has been approved. The data field should be present.
- A DECLINED status means that the request has been declined. The reason field should contain a human readable reason for the refusal.

- A PENDING status means that the request is being processed. More PENDING responses may follow, but eventually a non-pending status should be returned. The updates and local fields may be present.
- The REDIRECT status is reserved, and should not be used by wallet services.

In summary, zero or more PENDING responses should be followed by a non-pending response. It is entirely acceptable for your service to immediately return an APPROVED Polling Response, skipping a PENDING state.

See also [PollingResponse](#).

Here are some examples of valid PollingResponse objects:

```
javascript
// APPROVED
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  data: , // what the service needs to send to FCL
}

// Declined
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "DECLINED",
  reason: "Declined by user."
}

// Pending - Simple
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "PENDING",
  updates: {
    ftype: "Service",
    fvsn: "1.0.0",
    type: "back-channel-rpc",
    endpoint: "https://", // where post request will be sent
    method: "HTTP/POST",
    data: {}, // will be included in the request's body
    params: {}, // will be included in the request's url
  }
}

// Pending - First Time with Local
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "PENDING",
  updates: {
    ftype: "Service",
```

```

    fvsn: "1.0.0",
    type: "back-channel-rpc",
    endpoint: "https://", // where post request will be sent
    method: "HTTP/POST",
    data: {}, // included in body of request
    params: {}, // included as query params on endpoint
  },
  local: {
    ftype: "Service",
    fvsn: "1.0.0",
    endpoint: "https://", // the iframe that will be rendered,
    method: "VIEW/IFRAME",
    data: {}, // sent to frame when ready
    params: {}, // included as query params on endpoint
  }
}

```

A PollingResponse can alternatively be constructed using WalletUtils when sending "APPROVED" or "DECLINED" responses.

```

javascript
import {WalletUtils} from "@onflow/fcl"

// Approving a PollingResponse
// Example using an AuthnResponse as the PollingResponse data
WalletUtils.approve({
  ftype: "AuthnResponse",
  fvsn: "1.0.0"
  ...
})

// Rejecting a PollingResponse
// Supplies a reason for declining
const reason = "User declined to authenticate."
WalletUtils.decline(reason)

```

Service

```

typescript
type ServiceType =
  | 'authn'
  | 'authz'
  | 'user-signature'
  | 'pre-authz'
  | 'open-id'
  | 'back-channel-rpc'
  | 'authn-refresh'

type ServiceMethod =
  | 'HTTP/POST'
  | 'IFRAME/RPC'
  | 'POP/RPC'

```

```

| 'TAB/RPC'
| 'EXT/RPC'
| 'DATA'

interface Service extends ObjectBase {
  ftype: 'Service'
  type: ServiceType
  method: ServiceMethod
  uid: string
  endpoint: string
  id: string
  identity: Identity
  provider?: ServiceProvider
  data?: FclObject
}

```

The meaning of the fields is as follows.

- **type**: The type of this service.
- **method**: The service method this service uses. DATA means that the purpose of this service is just to provide the information in this Service object, and no active communication services are provided.
- **uid**: A unique identifier for the service. A common scheme for deriving this is to use '`wallet-name#${type}`', where `#{type}` refers to the type of this service.
- **endpoint**: Defines where to communicate with the service.
 - When method is EXT/RPC, this can be an arbitrary unique string, and the extension will need to use it to identify its own services. A common scheme for deriving the endpoint is to use '`ext:${address}`', where `${address}` refers to the wallet's address. (See ServiceProvider for more information.)
- **id**: The wallet's internal identifier for the user. If no other identifier is used, simply the user's flow account address can be used here.
- **identity**: Information about the identity of the user.
- **provider**: Information about the wallet.
- **data**: Additional information used with a service of type open-id.

See also:

- authn
- authz
- user-signature
- pre-authz
- open-id
- back-channel-rpc

Identity

This object is used to define the identity of the user.

```

typescript
interface Identity extends ObjectBase {
  ftype: 'Identity'
  address: string
  keyId?: number

```

```
}
```

The meaning of the fields is as follows.

- address: The flow account address of the user.
- keyId: The id of the key associated with this account that will be used for signing.

ServiceProvider

This object is used to communicate information about a wallet.

typescript

```
interface ServiceProvider extends ObjectBase {  
    ftype: 'ServiceProvider'  
    address: string  
    name?: string  
    description?: string  
    icon?: string  
    website?: string  
    supportUrl?: string  
    supportEmail?: string  
}
```

The meaning of the fields is as follows.

- address: A flow account address owned by the wallet. It is unspecified what this will be used for.
- name: The name of the wallet.
- description: A short description for the wallet.
- icon: An image URL for the wallet's icon.
- website: The wallet's website.
- supportUrl: A URL the user can use to get support with the wallet.
- supportEmail: An e-mail address the user can use to get support with the wallet.

AuthnResponse

This object is used to inform FCL about the services a wallet provides.

typescript

```
interface AuthnResponse extends ObjectBase {  
    ftype: 'AuthnResponse'  
    addr: string  
    services: Service[]  
}
```

The meaning of the fields is as follows.

- addr: The flow account address of the user.
- services: The list of services provided by the wallet.

Signable

typescript

```
interface Signable extends ObjectBase<'1.0.1'> {
```

```
        ftype: 'Signable'
        addr: string
        keyId: number
        voucher: {
            cadence: string
            refBlock: string
            computeLimit: number
            arguments: {
                type: string
                value: unknown
            } []
        proposalKey: {
            address: string
            keyId: number
            sequenceNum: number
        }
        payer: string
        authorizers: string[]
    }
}
```

The `WalletUtils.encodeMessageFromSignable` function can be used to calculate the message that needs to be signed.

CompositeSignature

```
typescript
interface CompositeSignature extends ObjectBase {
    ftype: 'CompositeSignature'
    addr: string
    keyId: number
    signature: string
}
```

See also `CompositeSignature`.

OpenID

TODO

Miscellaneous objects

```
Message
typescript
type MessageType =
    | 'FCL:VIEW:READY'
    | 'FCL:VIEW:READY:RESPONSE'
    | 'FCL:VIEW:RESPONSE'
    | 'FCL:VIEW:CLOSE'

type Message = {
    type: MessageType
}
```

A message that indicates the status of the protocol invocation.

This type is sometimes used as part of an intersection type. For example, the type Message & PollingResponse means a PollingResponse extended with the type field from Message.

```
ExtensionServiceInitiationMessage  
typescript  
type ExtensionServiceInitiationMessage = {  
    service: Service  
}
```

This object is used to invoke a service when the EXT/RPC service method is used.

See also

- local-view
- frame

Service Methods

IFRAME/RPC (Front Channel)

IFRAME/RPC is the easiest to explain, so we will start with it:

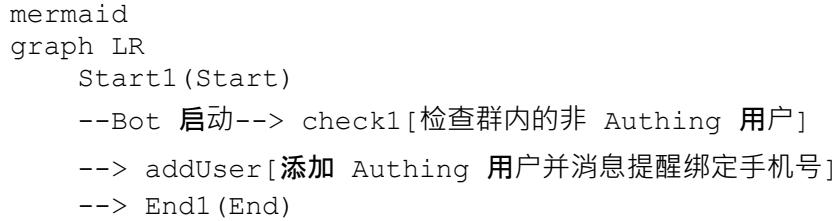
- An iframe is rendered (comes from the endpoint in the service).
- The rendered iframe adds a listener and sends the "FCL:VIEW:READY" message. This can be simplified WalletUtils.ready(callback)
- FCL will send the data to be dealt with:
 - Where body is the stuff you care about, params and data are additional information you can provide in the service object.
- The wallet sends back an "APPROVED" or "DECLINED" post message. (It will be a ftype: "PollingResponse", which we will get to in a bit). This can be simplified using WalletUtils.approve and WalletUtils.decline
 - If it's approved, the polling response's data field will need to be what FCL is expecting.
 - If it's declined, the polling response's reason field should say why it was declined.

```
javascript  
export const WalletUtils.approve = data => {  
    sendMsgToFCL("FCL:VIEW:RESPONSE", {  
        ftype: "PollingResponse",  
        fvsn: "1.0.0",  
        status: "APPROVED",  
        reason: null,  
        data: data,  
    })  
}  
  
export const WalletUtils.decline = reason => {
```

```

    sendMsgToFCL("FCL:VIEW:RESPONSE", {
      ftype: "PollingResponse",
      fvsn: "1.0.0",
      status: "DECLINED",
      reason: reason,
      data: null,
    })
}

```



!IFRAME/RPC Diagram

POP/RPC | TAB/RPC (Front Channel)

POP/RPC and TAB/RPC work in an almost entirely similar way to IFRAME/RPC, except instead of rendering the method in an iframe, we render it in a popup or new tab. The same communication protocol between the rendered view and FCL applies.

!POP/RPC Diagram

!TAB/RPC Diagram

HTTP/POST (Back Channel)

HTTP/POST initially sends a post request to the endpoint specified in the service, which should immediately return a ftype: "PollingResponse".

Like IFRAME/RPC, POP/RPC or TAB/RPC, our goal is to eventually get an APPROVED or DECLINED polling response, and technically this endpoint could return one of those immediately.

But more than likely that isn't the case and it will be in a PENDING state (PENDING is not available to IFRAME/RPC, POP/RPC or TAB/RPC). When the polling response is PENDING it requires an updates field that includes a service, BackChannelRpc, that FCL can use to request an updated PollingResponse from.

FCL will use that BackChannelRpc to request a new PollingResponse which itself can be APPROVED, DECLINED or PENDING.

If it is APPROVED FCL will return, otherwise if it is DECLINED FCL will error. However, if it is PENDING, it will use the BackChannelRpc supplied in the new PollingResponse updates field. It will repeat this cycle until it is either APPROVED or DECLINED.

There is an additional optional feature that HTTP/POST enables in the first PollingResponse that is returned.
This optional feature is the ability for FCL to render an iframe, popup or new tab, and it can be triggered by supplying a service type: "VIEW/IFRAME", type: "VIEW/POP" or type: "VIEW/TAB" and the endpoint that the wallet wishes to render in the local field of the PollingResponse.
This is a great way for a wallet provider to switch to a webpage if displaying a UI is necessary for the service it is performing.

!HTTP/POST Diagram

EXT/RPC (Front Channel)

EXT/RPC is used to enable and communicate between FCL and an installed web browser extension. (Though this specification is geared towards Chromium based browsers, it should be implementable in any browser with similar extension APIs available. From now on we will be using the word Chrome to refer to Chromium based browsers.)

An implementation of EXT/RPC needs to somehow enable communication between the application and the extension context. Implementing this is a bit more complex and usually relies on 3 key scripts to allow message passing between an installed extension and FCL. The separation of contexts enforced by Chrome and the availability of different Chrome APIs within those contexts require these scripts to be set up in a particular sequence so that the communication channels needed by FCL's EXT/RPC service method will work.

The following is an overview of these scripts and the functionality they need to support FCL:

- background.js: Used to launch the extension popup with chrome.windows.create if selected by the user from Discovery or set directly via fcl.config.discovery.wallet
- content.js: Used to proxy messages between the application to the extension via chrome.runtime.sendMessage.
- script.js: Injected by content.js into the application's HTML page. It appends the extension authn service to the window.fclexensions array on page load. This allows FCL to confirm installation and send extension details to Discovery or launch your wallet as the default wallet.

An example and guide showing how to build an FCL compatible wallet extension on Flow can be found [here](#).

Once the extension is enabled (for example when the user selects it through the discovery service), the following communication protocol applies. (The term send should specifically refer to using window.postMessage in the application context, as this is the only interface between the application and the extension. Note that since window.postMessage broadcasts messages to all message event handlers, care should be taken by each party to filter only the messages targeted at them.)

- An ExtensionServiceInitiationMessage object is sent by FCL. It is the extension's responsibility to inspect the endpoint field of the service, and only activate itself (e.g. by opening a popup) if it is the provider of this service.
- The extension should respond by sending a Message with type FCL:VIEW:READY. (Usually this message will originate from the extension popup, and be relayed to the application context.)
- FCL will send a Message with type FCL:VIEW:READY:RESPONSE. Additional fields specific to the service (such as body, params or data) are usually present. See the section on the specific service for a description of these fields.
- The wallet sends back a Message & PollingResponse with type FCL:VIEW:RESPONSE with either an APPROVED or DECLINED status.
 - If it's approved, the polling response's data field will need to be what FCL is expecting.
 - If it's declined, the polling response's reason field should say why it was declined.

The extension can send a Message with type FCL:VIEW:CLOSE at any point during this protocol to indicate an interruption. This will halt FCL's current routine. On the other hand, once a PollingResponse with either an APPROVED or DECLINED status was sent, the protocol is considered finished, and the extension should not send any further messages as part of this exchange.

Conversely, when FCL sends a new ExtensionServiceInitiationMessage, the previous routine is interrupted. (This is the case even when the new service invocation is targeted at a different extension.)

Note that as a consequence of the above restrictions, only single service invocation can be in progress at a time.

Here is a code example for how an extension popup might send its response:

```
javascript
chrome.tabs.sendMessage(tabs[0].id, {
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  reason: null,
  data: {
    ftype: "AuthnResponse",
    fvsn: "1.0.0",
    addr: address,
    services: services,
  },
});
```

!EXT/RPC Diagram

data and params

- data and params are information that the wallet can provide in the service config that FCL will pass back to the service.
- params will be added onto the endpoint as query params.
 - data will be included in the body of the HTTP/POST request or in the FCL:VIEW:READY:RESPONSE for a IFRAME/RPC, POP/RPC, TAB/RPC or EXT/RPC.

Authentication Service

In the following examples, we'll walk you through the process of building an authentication service.

In FCL, wallets are configured by passing in a wallet provider's authentication URL or extension endpoint as the discovery.wallet config variable.

You will need to make and expose a webpage or API hosted at an authentication endpoint that FCL will use.

```
javascript
// IN APPLICATION
// configuring fcl to point at a wallet looks like this
import {config} from "@onflow/fcl"

config({
  "discovery.wallet": "url-or-endpoint-fcl-will-use-for-authentication",
  // FCL Discovery endpoint, wallet provider's authentication URL or
  extension endpoint
  "discovery.wallet.method": "IFRAME/RPC" // Optional. Available methods
  are "IFRAME/RPC", "POP/RPC", "TAB/RPC", "EXT/RPC" or "HTTP/POST",
  defaults to "IFRAME/RPC".
})
```

If the method specified is IFRAME/RPC, POP/RPC or TAB/RPC, then the URL specified as discovery.wallet will be rendered as a webpage. If the configured method is EXT/RPC, discovery.wallet should be set to the extension's authn endpoint. Otherwise, if the method specified is HTTP/POST, then the authentication process will happen over HTTP requests. (While authentication can be accomplished using any of those service methods, this example will use the IFRAME/RPC service method.)

Once the Authentication webpage is rendered, the extension popup is enabled, or the API is ready, you then need to tell FCL that it is ready. You will do this by sending a message to FCL, and FCL will send back a message with some additional information that you can use about the application requesting authentication on behalf of the user.

The following example is using the IFRAME/RPC method. Your authentication webpage will likely resemble the following code:

```
javascript
// IN WALLET AUTHENTICATION FRAME
import {WalletUtils} from "@onflow/fcl"
```

```

function callback(data) {
  if (typeof data != "object") return
  if (data.type !== "FCL:VIEW:READY:RESPONSE") return

  ... // Do authentication things ...

  // Send back AuthnResponse
  WalletUtils.sendMsgToFCL("FCL:VIEW:RESPONSE", {
    ftype: "PollingResponse",
    fvsn: "1.0.0",
    status: "APPROVED",
    data: {
      ftype: "AuthnResponse",
      fvsn: "1.0.0"
      ...
    }
  })

  // Alternatively be sent using WalletUtils.approve (or
  WalletUtils.decline)
  // which will wrap AuthnResponse in a PollingResponse
  WalletUtils.approve({
    ftype: "AuthnResponse",
    fvsn: "1.0.0"
    ...
  })
  // add event listener first
  WalletUtils.onMsgFromFCL("FCL:VIEW:READY:RESPONSE", callback)

  // tell fcl the wallet is ready
  WalletUtils.sendMsgToFCL("FCL:VIEW:READY")

  // alternatively adds "FCL:VIEW:READY:RESPONSE" listener and sends
  "FCL:VIEW:READY"
  WalletUtils.ready(callback)
}

```

During authentication, the application has a chance to request to you what they would like you to send back to them. These requests are included in the FCL:VIEW:READY:RESPONSE message sent to the wallet from FCL.

An example of such a request is the OpenID service. The application can request for example that you to send them the email address of the current user. The application requesting this information does not mean you need to send it. It's entirely optional for you to do so. However, some applications may depend on you sending the requested information back, and should you decline to do so it may cause the application to not work.

In the config they can also tell you a variety of things about them, such as the name of their application or a url for an icon of their

application. You can use these pieces of information to customize your wallet's user experience should you desire to do so.

Your wallet having a visual distinction from the application, but still a seamless and connected experience is our goal here.

Whether your authentication process happens using a webpage with the IFRAME/RPC, POP/RPC or TAB/RPC methods, via an enabled extension using the EXT/RPC method, or using a backchannel to an API with the HTTP/POST method, the handshake is the same. The same messages are sent in all methods, however the transport mechanism changes. For IFRAME/RPC, POP/RPC, TAB/RPC or EXT/RPC methods, the transport is `window.postMessage()`, with the HTTP/POST method, the transport is HTTP post messages.

As always, you must never trust anything you receive from an application. Always do your due-diligence and be alert as you are the user's first line of defense against potentially malicious applications.

Authenticate your User

It's important that you are confident that the user is who the user claims to be.

Have them provide enough proof to you that you are okay with passing their details back to FCL.

Using Blocto as an example, an authentication code is sent to the email a user enters at login.

This code can be used as validation and is everything Blocto needs to be confident in the user's identity.

Once you know who your User is

Once you're confident in the user's identity, we can complete the authentication process.

The authentication process is complete once FCL receives back a response that configures FCL with FCL Services for the current user. This response is extremely important to FCL. At its core it tells FCL who the user is, and then via the included services it tells FCL how the user authenticated, how to request transaction signatures, how to get a personal message signed and the user's email and other details if requested. In the future it may also include many more things!

You can kind of think of FCL as a plugin system. But since those plugins exist elsewhere outside of FCL, FCL needs to be configured with information on how to communicate with them.

What you are sending back to FCL is everything that it needs to communicate with the plugins that you are supplying.

Your wallet is like a plugin to FCL, and these details tell FCL how to use you as a plugin.

Here is an example of an authentication response:

```

javascript
// IN WALLET AUTHENTICATION FRAME
import {WalletUtils} from "@onflow/fcl"

WalletUtils.approve({
  ftype: "AuthnResponse",
  fvsn: "1.0.0",
  addr: "0xUSER", // The user's flow address

  services: [ // All the stuff that configures
FCL

    // Authentication Service - REQUIRED
    {
      ftype: "Service", // It's a service!
      fvsn: "1.0.0", // Follows the v1.0.0 spec for the service
      type: "authn", // the type of service it is
      method: "DATA", // It's data!
      uid: "amazing-wallet#authn", // A unique identifier for the service
      endpoint: "your-url-that-fcl-will-use-for-authentication", // should be the same as was passed into the config
      id: "0xUSER", // the wallet's internal id for the user, use flow address if you don't have one
      // The User's Info
      identity: {
        ftype: "Identity", // It's an Identity!
        fvsn: "1.0.0", // Follows the v1.0.0 spec for an identity
        address: "0xUSER", // The user's address
        keyId: 0, // OPTIONAL - The User's KeyId they will use
      },
      // The Wallet's Info
      provider: {
        ftype: "ServiceProvider", // It's a Service Provider
        fvsn: "1.0.0", // Follows the v1.0.0 spec for service providers
        address: "0xWallet", // A flow address owned by the wallet
        name: "Amazing Wallet", // OPTIONAL - The name of your wallet. ie: "Dapper Wallet" or "Blocto Wallet"
        description: "The best wallet", // OPTIONAL - A short description for your wallet
        icon: "https://", // OPTIONAL - Image url for your wallet's icon
        website: "https://", // OPTIONAL - Your wallet's website
      }
    }
  ]
})

```

```

                supportUrl: "https://",           // OPTIONAL - An url the user
can use to get support from you
                supportEmail: "help@aw.com",    // OPTIONAL - An email the
user can use to get support from you
            },
        },
        // Authorization Service
    {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authz",
        uid: "amazing-wallet#authz",
        ...
        // We will cover this at length in the authorization section of
this guide
    },
    // User Signature Service
    {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "user-signature",
        uid: "amazing-wallet#user-signature",
        ...
        // We will cover this at length in the user signature section
of this guide
    },
    // OpenID Service
    {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "open-id",
        uid: "amazing-wallet#open-id",
        method: "DATA",
        data: { // only include data that was request, ideally only if
the user approves the sharing of data, everything is optional
            ftype: "OpenID",
            fvsn: "1.0.0",
            profile: {
                name: "Jeff",
                familyname: "D", // icky underscored names because of
OpenID Connect spec
                givenname: "Jeffrey",
                middlename: "FakeMiddleName",
                nickname: "JeffJeff",
                preferredusername: "Jeff",
                profile: "https://www.jeff.jeff/",
                picture: "https://avatars.onflow.org/avatar/jeff",
                website: "https://www.jeff.jeff/",
                gender: "male",
                birthday: "1900-01-01", // can use 0000 for year if
year is not known
            }
        }
    }
}
```

```

        zoneinfo: "America/Vancouver",
        locale: "en",
        updatedat: "1625588304427"
    },
    email: {
        email: "jeff@jeff.jeff",
        emailverified: false,
    }
},
]
})

```

Stopping an Authentication Process

From any frame, you can send a FCL:VIEW:CLOSE post message to FCL, which will halt FCL's current routine and close the frame.

```

javascript
import {WalletUtils} from "@onflow/fcl"

WalletUtils.sendMsgToFCL("FCL:VIEW:CLOSE")

```

Authorization Service

Authorization services are depicted with a type: "authz", and a method of either HTTP/POST, IFRAME/RPC, POP/RPC, TAB/RPC or EXT/RPC. They are expected to eventually return a ftype: "CompositeSignature".

An authorization service is expected to know the Account and the Key that will be used to sign the transaction at the time the service is sent to FCL (during authentication).

```

javascript
{
    ftype: "Service",
    fvsn: "1.0.0",
    type: "authz",           // say it's an authorization service
    uid: "amazing-wallet#authz", // standard service uid
    method: "HTTP/POST",       // can also be IFRAME/RPC or POP/RPC
    endpoint: "https://",      // where to talk to the service
    identity: {
        ftype: "Identity",
        fvsn: "1.0.0",
        address: "0xUser",           // the address that the signature will be
        for: {
            keyId: 0,                // the key for the address that the
            signature will be for
        },
        data: {},
        params: {}
    }
}

```

FCL will use the method provided to request an array of composite signature from authorization service (Wrapped in a PollingResponse). The authorization service will be sent a Signable. The service is expected to construct an encoded message to sign from Signable.voucher. It then needs to hash the encoded message, and prepend a required transaction domain tag. Finally it signs the payload with the user/s keys, producing a signature. This signature, as a HEX string, is sent back to FCL as part of the CompositeSignature which includes the user address and keyID in the data property of a PollingResponse.

```
elixir
signature =
  signable.voucher
    |> encode
    |> hash
    |> tag
    |> sign
    |> converttohex
```

The eventual response back from the authorization service should resolve to something like this:

```
javascript
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  data: {
    ftype: "CompositeSignature",
    fvsn: "1.0.0",
    addr: "0xUSER",
    keyId: 0,
    signature: "signature as hex value"
  }
}
```

A CompositeSignature can alternatively be constructed using WalletUtils

```
javascript
import {WalletUtils} from "@onflow/fcl"

WalletUtils.CompositeSignature(addr: String, keyId: Number, signature: Hex)
```

User Signature Service

User Signature services are depicted with a type: "user-signature" and a method of either HTTP/POST, IFRAME/RPC, POP/RPC, TAB/RPC or EXT/RPC. They are expected to eventually return an array of ftype: "CompositeSignature".

The User Signature service is a stock/standard service.

```
javascript
{
  ftype: "Service",
  fvsn: "1.0.0",
  type: "user-signature", // say it's an user-signature
  service
  uid: "amazing-wallet#user-signature", // standard service uid
  method: "HTTP/POST", // can also be IFRAME/RPC
  endpoint: "https://", // where to talk to the service
  data: {},
  params: {},
}
```

FCL will use the method provided to request an array of composite signatures from the user signature service (Wrapped in a PollingResponse).

The user signature service will be sent a Signable.

The service is expected to tag the Signable.message and then sign it with enough keys to produce a full weight.

The signatures need to be sent back to FCL as HEX strings in an array of CompositeSignatures.

```
javascript
// Pseudocode:
// For every required signature
import {WalletUtils} from "@onflow/fcl"

const encoded = WalletUtils.encodeMessageFromSignable(signable,
signerAddress)
const taggedMessage = tagMessage(encoded) // Tag the message to sign
const signature = signMessage(taggedMessage) // Sign the message
const hexSignature = signatureToHex(signature) // Convert the signature
to hex, if required.

return hexSignature
```

The eventual response back from the user signature service should resolve to something like this:

```
javascript
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  data: [
    {
      "signature": "0x...",
      "weight": 100
    }
  ]
}
```

```

        ftype: "CompositeSignature",
        fvsn: "1.0.0",
        addr: "0xUSER",
        keyId: 0,
        signature: "signature as hex value"
    },
{
    ftype: "CompositeSignature",
    fvsn: "1.0.0",
    addr: "0xUSER",
    keyId: 1,
    signature: "signature as hex value"
}
]
}

```

Pre Authz Service

This is a strange one, but extremely powerful. This service should be used when a wallet is responsible for an account that's signing as multiple roles of a transaction, and wants the ability to change the accounts on a per role basis.

Pre Authz Services are depicted with a type: "pre-authz" and a method of either HTTP/POST, IFRAME/RPC, POP/RPC, TAB/RPC or EXT/RPC. They are expected to eventually return a ftype: "PreAuthzResponse".

The Pre Authz Service is a stock/standard service.

```

javascript
{
    ftype: "Service",
    fvsn: "1.0.0",
    type: "pre-authz",           // say it's a pre-authz service
    uid: "amazing-wallet#pre-authz", // standard service uid
    method: "HTTP/POST",          // can also be IFRAME/RPC, POP/RPC,
TAB/RPC
    endpoint: "https://",         // where to talk to the service
    data: {},
    params: {}
}

```

FCL will use the method provided to request a PreAuthzReponse (Wrapped in a PollingResponse).

The Authorizations service will be sent a PreSignable.

The pre-authz service is expected to look at the PreSignable and determine the breakdown of accounts to be used.

The pre-authz service is expected to return Authz services for each role it is responsible for.

A pre-authz service can only supply roles it is responsible for.

If a pre-authz service is responsible for multiple roles, but it wants the same account to be responsible for all the roles, it will need to supply an Authz service per role.

The eventual response back from the pre-authz service should resolve to something like this:

```
javascript
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  data: {
    ftype: "PreAuthzResponse",
    fvsn: "1.0.0",
    proposer: { // A single Authz Service
      ftype: "Service",
      fvsn: "1.0.0",
      type: "authz",
      ...
    },
    payer: [ // An array of Authz Services
      {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authz",
        ...
      }
    ],
    authorization: [ // An array of Authz Services (it's singular
because it only represents a singular authorization)
      {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authz",
        ...
      }
    ],
  }
}
```

Authentication Refresh Service

Since synchronization of a user's session is important to provide a seamless user experience when using an app and transacting with the Flow Blockchain, a way to confirm, extend, and refresh a user session can be provided by the wallet.

Authentication Refresh Services should include a type: "authn-refresh", endpoint, and supported method (HTTP/POST, IFRAME/RPC, POP/RPC, or EXT/RPC).

FCL will use the endpoint and service method provided to request updated authentication data.

The authn-refresh service should refresh the user's session if necessary and return updated authentication configuration and user session data.

The service is expected to return a PollingResponse with a new AuthnResponse as data. If user input is required, a PENDING PollingResponse can be returned with a local view for approval/resubmission of user details.

The Authentication Refresh Service is a stock/standard service.

```
javascript
{
  "ftype": "Service",
  "fvsn": "1.0.0",
  "type": "authn-refresh",
  "uid": "uniqueDedupeKey",
  "endpoint": "https://rawr",
  "method": "HTTP/POST", // "HTTP/POST", // HTTP/POST | IFRAME/RPC | HTTP/RPC
  "id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx", // wallet's internal id for the user
  "data": {}, // included in body of request
  "params": {}, // included as query params on endpoint url
}
```

The provided data and params should include all the wallet needs to identify and re-authenticate the user if necessary.

The eventual response back from the authn-refresh service should resolve to an AuthnResponse and look something like this:

```
javascript
{
  ftype: "PollingResponse",
  fvsn: "1.0.0",
  status: "APPROVED",
  data: {
    ftype: "AuthnResponse",
    fvsn: "1.0.0",
    addr: "0xUSER",
    services: [
      // Authentication Service - REQUIRED
      {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authn",
        ...
      },
      // Authorization Service
      {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authz",
        ...
      }
    ]
  }
}
```

```

        ...
    },
    // Authentication Refresh Service
    {
        ftype: "Service",
        fvsn: "1.0.0",
        type: "authn-refresh",
        ...
    }
    // Additional Services
],
}
}

```

provable-authn.md:

Provable Authn

In order to improve UX/DX and encourage seamless integration with App backends and services, `fcl.authenticate` has been upgraded.

Additional data is sent in the body of FCL:VIEW:READY:RESPONSE. This data includes what the wallet needs to build a message for signing with the user's private key/s.

The signature can be returned as part of an optional account-proof service with the FCL:VIEW:RESPONSE.

When provided by the wallet, this signature and additional account-proof data is available to the App via `fcl.currentUser` services. The service data can be used to recreate the message, and verify the signature on the Flow Blockchain.

For example, it can be sent to the App's backend and after validating the signature and the other account-proof data, it can safely associate the included account address to a user and log them in.

TL;DR Wallet Provider

1. Wallet receives Authn FCL:VIEW:READY:RESPONSE request and parses out the `appIdentifier`, and `nonce`.
2. The wallet authenticates the user however they choose to do, and determines the user's account address
3. Wallet prepares and signs the message:
 - Encodes the `appIdentifier`, `nonce`, and address along with the "FCL-ACCOUNT-PROOF-V0.0" domain separation tag, using the encoding scheme described below.
 - Signs the message with the `signatureAlgorithm` and `hashAlgorithm` specified on user's key. It is highly recommended that the wallet display the message data and receive user approval before signing.

4. Wallet sends back this new service and data along with the other service configuration when completing Authn.

Account Proof Message Encoding

The account proof message is encoded as follows:

```
text
MESSAGE =
USERDOMAINTAG ||
RLPENCODE([
  APPIDENTIFIER,
  ADDRESS,
  NONCE
])
```

with the following values:

- ACCOUNTPROOFDOMAINTAG is the constant "FCL-ACCOUNT-PROOF-V0.0", encoded as UTF-8 byte array and right-padded with zero bytes to a length of 32 bytes.
- APPIDENTIFIER is an arbitrary length string.
- ADDRESS is a byte array containing the address bytes, left-padded with zero bytes to a length of 8 bytes.
- NONCE is an byte array with a minimum length of 32 bytes.

RLPENCODE is a function that performs RLP encoding and returns the encoded value as bytes.

JavaScript Signing Example

```
javascript
// Using WalletUtils
import {WalletUtils} from "@onflow/fcl"

const message = WalletUtils.encodeAccountProof(
  appIdentifier, // A human readable string to identify your application
  during signing
  address,       // Flow address of the user authenticating
  nonce,         // minimum 32-btye nonce
)

sign(privateKey, message)

// Without using FCL WalletUtils
const ACCOUNTPROOFDOMAINTAG = rightPaddedHexBuffer(
  Buffer.from("FCL-ACCOUNT-PROOF-V0.0").toString("hex"),
  32
)
const message = rlp([appIdentifier, address, nonce])
const prependUserDomainTag = (message) => ACCOUNTPROOFDOMAINTAG + message

sign(privateKey, prependUserDomainTag(message))
```

```

json
// Authentication Proof Service
{
  ftype: "Service",                                // Its a service!
  fvsn: "1.0.0",                                   // Follows the v1.0.0 spec for
the service
  type: "account-proof",                           // the type of service it is
  method: "DATA",                                  // Its data!
  uid: "awesome-wallet#account-proof",             // A unique identifier for the
service
  data: {
    ftype: "account-proof",
    fvsn: "1.0.0"
    // The user's address (8 bytes, i.e 16 hex characters)
    address: "0xf8d6e0586b0a20c7",
    // Nonce signed by the current account-proof (minimum 32 bytes in
total, i.e 64 hex characters)
    nonce:
"75f8587e5bd5f9dcc9909d0dae1f0ac5814458b2ae129620502cb936fde7120a",
    signatures: [CompositeSignature],
  }
}

```

user-signature.md:

User Signature

Status

- Last Updated: June 1st 2021
- Stable: Yes
- Risk of Breaking Change: Low
- Compatibility: >= @onflow/fcl@0.0.71

Overview and Introduction

Personally sign data via FCL Compatible Wallets

FCL now includes `signUserMessage()` which allows for the sending of unencrypted message data to a connected wallet provider or service to be signed with a user's private key.

An application or service can verify a signature against a user's public key on the Flow Blockchain, providing proof a user controls the account's private key.

Use Cases

- Authentication: Cryptographically verify the ownership of a Flow account by signing a piece of data using a private key

- Improved Application Login
 - Increased security: Arguably more secure than proof of ownership by email/password
 - Simplified UX: No application password required
 - Increased privacy: No email or third party authentication service needed
- Message Validation: Assuring that a message sent or received has not been tampered with
- Multisig contracts
- Decentralised exchanges
- Meta transactions

Config and Authentication

As a prerequisite, FCL is configured to point to the Wallet Provider's Authentication Endpoint. No additional configuration is required.

- > During development (and on mainnet) FCL can be configured to use the wallet directly by
- > setting the Wallet Discovery Url to the wallet provider's Authentication Endpoint
- > by configuring fcl like this config().put("discovery.wallet", "https://my-awesome-wallet-provider.com/fcl/authenticate") .

Common Configuration Keys and additional info can be found here [How to Configure FCL](#)

1. A user initiates authentication with the wallet provider via application UI
2. The wallet confirms a user's identity and sends back information used to configure FCL for future user actions in the application
3. Included in the authentication response should be the provider's Key Services including a user-signature service for use with signUserMessage()

User Signature Service

A user-signature service is a standard service, with methods for IFRAME/RPC or HTTP/POST.

The user-signature service receives a signable message from FCL and returns a standard PollingResponse with an array of CompositeSignatures or null as the data.

A status of Approved needs to have an array of composite signatures as data.

A status of Declined needs to include a reason why.

A Pending status needs to include an updates service and can include a local.

A service using the IFRAME/RPC method can only respond with approved or declined, as pending is not valid for iframes.

When `signUserMessage()` is called by the application, FCL uses the service method to decide how to send the signable to the wallet.

The Wallet is responsible for prepending the signable with the correct `UserDomainTag`, hashing, and signing the message.

Signing Sequence

1. Application sends message to signing service. FCL expects a hexadecimal string
3. Wallet/Service tags the message with required `UserDomainTag` (see below), hashes, and signs using the `signatureAlgorithm` specified on account key
2. Wallet makes available a Composite Signature consisting of `addr`, `keyId`, and `signature` as a hex string

UserDomainTag

The `UserDomainTag` is the prefix of all signed user space payloads.

Before hashing and signing the message, the wallet must add a specified DOMAIN TAG.

```
> currently "FLOW-V0.0-user"
```

A domain tag is encoded as UTF-8 bytes, right padded to a total length of 32 bytes, prepended to the message.

The signature can now be verified on the Flow blockchain. The following illustrates an example using `fcl.verifyUserSignatures`

```
javascript
/
  Verify a valid signature/s for an account on Flow.

  @param {string} msg - A message string in hexadecimal format
  @param {Array} compSigs - An array of Composite Signatures
  @param {string} compSigs[].addr - The account address
  @param {number} compSigs[].keyId - The account keyId
  @param {string} compSigs[].signature - The signature to verify
  @return {bool}

@example

  const isValid = await fcl.verifyUserSignatures(
    Buffer.from('FOO').toString("hex"),
    [{ftype: "CompositeSignature", fvsn: "1.0.0", addr: "0x123", keyId: 0, signature: "abc123"}]
  )
/
```

- Register with FCL and provide signing service endpoint. No further configuration is needed.
- On receipt of message, prompt user to approve or decline
- Prepend UserDomainTag, hash and sign the message with the signatureAlgorithm specified on user's key
- Return a standard PollingResponse with an array of CompositeSignatures as data or null and reason if declined