Example Codebases

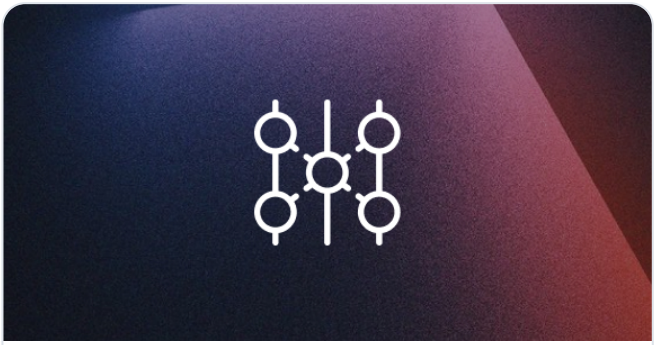
The Euclid SDK is designed to provide developers with the tools they need to build robust and scalable decentralized applications on the Constellation Network. To help you get started, we have curated a list of exemplary codebases that you can explore and learn from. These codebases are open-source projects that demonstrate various aspects of using the Euclid SDK in real-world scenarios.

Codebases

Metagraph Examples

The Metagraph Examples repository contains several minimalist metagraph codebases designed to demonstrate specific metagraph features in a simplified context. All projects in this repo can be installed with `hydra install-template`

Displays: many concepts.



Dor Metagraph

This repository is the codebase of the Dor Metagraph, the first metagraph to launch to MainNet. The Dor Metagraph ingests foot traffic data from a network of IoT sensors.

Displays: strategies for processing binary data types using decoders, reward distribution, and separation of public/private data using calculated state.

EL PACA Metagraph

This repository is the codebase of the EL PACA

Edit this page

Toolkit

Euclid Development Environment

An upgradable base for launching minimal development environments and developing metagraph projects locally.

Hydra CLI

A powerful command line utility to manage network clusters within the Euclid Development Environment.

Developer Dashboard

A frontend codebase for visual interaction with local development clusters.

Telemetry Dashboard

Custom telemetry tooling for monitoring local or deployed metagraph networks.

Metagraph Framework

A framework for Metagraph development.

Edit this page

Development Environment

The Euclid Development Environment is an upgradeable project framework to simplify the process of configuring a local development environment for metagraph developers on the Constellation Network.

Getting started with metagraph development can be challenging due to the infrastructure requirements for a minimal development environment. This is a significantly bigger challenge for Constellation Network developers compared to developers on blockchain-based networks due to the inherent complexity of Constellation's microservice-based architecture. The Euclid Development Environment was created to simplify that process as much as possible for metagraph developers, so that developers can focus on building their applications and business logic rather than deploying infrastructure. It includes open source tools that can be used as a starting point for developing more customized tooling specific to your project team's needs.

Minimal Development Environment

A minimal development environment for the Constellation Network consists of the following components:

- 1 Global L0 node
- 3 Metagraph L0 node
- 3 Metagraph L1 - Currency nodes
- 3 DAG L1 nodes (optional)
- 3 Metagraph L1 - Data nodes (optional)

Note that a cluster of at least three L1 nodes is necessary for the L1 layer to reach consensus (Metagraph L1 - Currency + DAG L1 + Metagraph L1 - Data).

See Network Architecture for an overview of the role each cluster plays in the Hypergraph.

System Requirements

For local development, it is sufficient to run each necessary node in a Docker container on a single developer machine. The minimal setup requires at least 5 docker containers which can be taxing on system resources. For that reason, we recommend your development machine have a minimum of **16 GB of RAM** with at least **8GB of that RAM allocated to Docker**. We recommend allocating 10GB RAM or more for a smoother development experience if your development machine can support it.

The system requirements for running a Euclid Development Environment project are:

- Linux or macOS operating system
- 16 GB RAM or more
- Docker installed with 8GB RAM allocated to it

Included Tools

The Euclid Development Environment includes the following components:

- The Hydra CLI tool for building and managing clusters of docker containers for each of the network configurations
- Docker files for building and connecting each of the required local clusters
- A Telemetry Dashboard consisting of two additional docker containers running Prometheus and Grafana.
- Optionally, run the Developer Dashboard, a NextJS frontend javascript app for use during development.

Install

Clone the repo from Github

```
git clone https://github.com/Constellation-Labs/euclid-development-environment.git
cd euclid-development-environment
```

See Hydra CLI for additional installation and configuration instructions.

Project Directory Structure

The project has the following structure:

```
- infra
- scripts
- source
- euclid.json
```

Infra

This directory contains infrastructure related to the running of Euclid.

- Docker: This directory contains docker configuration, including Dockerfiles, and port, IP, and name configurations for running Euclid locally.
- Ansible: This directory contains Ansible configurations to start your nodes locally and remotely
  - **local**: Used for start and stop the nodes locally.
  - **remote**: Used for configuring and deploying to remote hosts

Scripts

Thats the "home" of hydra script, here you'll find the `hydra` and `hydra-update (deprecated)` scripts.

euclid.json

Here is the hydra configuration file, there you can set the `p12` file names and your GITHUB_TOKEN. It's required to fill the GitHub token here to run the `hydra` script

Source

This directory contains your local source code for each project under the `project` directory. These directories will be empty by default, until the project is installed using `hydra install` or `hydra install-template` which will generate these project directories from a template. The files in these directories are automatically included in the metagraph-l0, data-l1, and currency-l1 docker containers when you run `hydra build`.

In this directory inside the `global-l0` subdirectory, you will find the genesis file for your Global L0 network. Updating this file will allow you to attribute DAG token amounts to addresses at the genesis of your network. The source for the Global L0 network is stored in a jar file under `source/docker` since this codebase is not meant to be modified.

Similarly, within the `metagraph-l0` subdirectory, you will find the genesis file for your Currency L0 network. Updating this file will allow you to attribute **metagraph token** amounts to addresses at the genesis of your network.

In the `p12-files` directory, you will find p12 files used in the default node configuration. You may update these files to use your own keys for your nodes. Environment variables in the `euclid.json` file should be updated with the new file aliases and passwords if you do choose to update them.

Hydra CLI

Hydra CLI is a powerful command line utility designed to manage local development Docker clusters in the Euclid Development Environment. With Hydra CLI, developers can easily create, configure, and manage Constellation Network development clusters for metagraph development.

Hydra CLI is a free and open-source tool that can be easily installed on any operating system that supports bash. Hydra is currently distributed as a part of the Euclid Development Environment project and can be found in the `scripts` directory.

Install Dependencies

Argc

```
cargo install argc
```

Or you can install the argc binaries directly.

Docker

- macOS
- linux

Ansible

- Ansible is a configuration tool for configuring and deploying to remote hosts
- Here you can check how to install Ansible

JQ

- jq is a lightweight and flexible command-line JSON processor. It allows you to manipulate JSON data easily, making it ideal for tasks like querying, filtering, and transforming JSON documents.
- Here you can check how to install jq

YQ

- yq is a powerful command-line YAML processor and parser, similar to jq but for YAML data. It allows you to query, filter, and manipulate YAML documents easily from the command line, making it a handy tool for tasks such as extracting specific data, updating YAML files, and formatting output.
- Here you can check how to install yq

Install Project

Run the `install` command which accomplishes two things:

- Creates templated currency starter projects for L0 and L1 Currency apps in the `source/` directory.
- Removes the project's git configuration so that you're free to check your changes into your own repo. Further infrastructure upgrades can be handled through Hydra.

```
scripts/hydra install
```

You can import a metagraph template from custom examples by using the following command:

```
scripts/hydra install-template
```

By default, we use the Metagraph Examples repository. You should provide the template name when running this command. To list the templates available to install, type:

```
scripts/hydra install-template --list
```

Configure

Create a Github personal access token. This token is necessary for building Tessellation from source.

See Creating a personal access token for details on how to create your token. The token only needs the `read:packages` permission.

Edit the `euclid.json` file with your token. You can leave the other variables as default for now.

```
{
  "github_token": "<your-token>",
  ...
}
```

Build

Build using the Hydra CLI. This will build a minimal development environment for your project using Docker.

```
scripts/hydra build
```

Usage

The primary purpose of Hydra is to manage local deployment and configuration of development clusters for developing metagraph projects. Running all the necessary network clusters for development can be quite complex to do from scratch, so Hydra aims to simplify that process.

See Network Architecture for an overview of the role each cluster plays in the Hypergraph.

Hydra uses Docker to launch minimal development clusters for the following supported networks:

- Global L0
- Currency L0
- Currency L1
- DAG L1

It also includes a pair of monitoring containers supporting:

- Prometheus
- Grafana

Building

Build the default clusters (Global L0, Currency L0, Currency L1, Monitoring)

```
scripts/hydra build
```

To include the DAG L1 network, you can add `dag-l1` to the `layers` field on `euclid.json`. This option is disabled by default because it is not strictly necessary for metagraph development.

Destroying

Built containers can be destroyed with the `destroy` command

```
scripts/hydra destroy
```

Starting

Run your built clusters with the `start-genesis` and `start-rollback` commands.

```
scripts/hydra start-genesis
```

```
scripts/hydra start-rollback
```

Stopping

Stop running containers with the `stop` command.

```
scripts/hydra stop
```

Check Status

Check the status of all running containers.

```
scripts/hydra status
```

Deployment

Configuring, deploying, and starting remote node instances is supported through Ansible playbooks. The default settings deploy to three node instances via SSH which host all layers of

your metagraph project (gL0, mL0, cL1, dL1). Two hydra methods are available to help with the deployment process: `hydra remote-deploy` and `hydra remote-start`. Prior to running these methods, remote host information must be configured in `infra/ansible/remote/hosts.ansible.yml`.

By default, we use the default directory for the SSH file, which is `~/.ssh/id_rsa`. However, you can change it to your preferred SSH file directory. You can find instructions on how to generate your SSH file here.

Ansible functions more effectively with `.pem` key files. If you possess a `.ppk` key file, you can utilize these instructions to convert it to `.pem`.

If your file contains a password, you will be prompted to enter it to proceed with remote operations.

Host Configuration

To run your metagraph remotely, you'll need remote server instances - 3 instances for the default configuration. These hosts should be running either `ubuntu-20.04` or `ubuntu-22.04`. It's recommended that each host meets the following minimum requirements:

- 16GB of RAM
- 8vCPU
- 160GB of storage

You can choose your preferred platform for hosting your instances, such as AWS or DigitalOcean. After creating your hosts, you'll need to provide the following information in the `hosts.ansible.yml` file:

- Host IP
- Host user
- Host SSH key (optional if your default SSH token already has access to the remote host)

P12 Files

P12 files contain the public/private key pair identifying each node (peerID) and should be located in the `source/p12-files` directory by default. The `file-name`, `key-alias`, and `password` should be specified in the `euclid.json` file under the `p12_files` section. By default, Euclid comes with three example files: `token-key.p12`, `token-key-1.p12`, and `token-key-2.p12`. **NOTE:** Before deploying, be sure to replace these example files with your own, as these files are public and their credentials are shared.

NOTE: If deploying to MainNet, ensure that your peerIDs are registered and present on the metagraph seedlist. Otherwise, the metagraph startup will fail because the network will reject the snapshots.

Network Selection

Currently, there are two networks available for running your metagraph: `IntegrationNet`, and `MainNet`. You need to specify the network on which your metagraph will run in the `euclid.json` file under `deploy -> network -> name`.

GL0 Node Configuration

The deploy script does not deploy the `gl0` node. It's recommended to use `nodectl` to build your `gl0` node. Information on installing `nodectl` can be found here. `Nodectl` helps manage `gl0` nodes by providing tools such as `auto-upgrade` and `auto-restart` which keep the node online in the case of a disconnection or network upgrade. Using these features is highly recommended for the stability of your metagraph.

NOTE: Your GL0 node must be up and running before deploying your metagraph. You can use the same host to run all four layers: `gl0`, `ml0`, `cl1`, and `dl1`.

`hydra remote-deploy`

This method configures remote instances with all the necessary dependencies to run a metagraph, including Java, Scala, and required build tools. The Ansible playbook used for this process can be found and edited in `infra/ansible/playbooks/deploy.ansible.yml`. It also

creates all required directories on the remote hosts, and creates or updates metagraph files to match your local Euclid environment. Specifically, it creates the following directories:

- `code/global-l0`
- `code/metagraph-l0`
- `code/currency-l1`
- `code/data-l1`

Each directory will be created with `cl-keytool.jar`, `cl-wallet.jar`, and a P12 file for the instance. Additionally, they contain the following:

In `code/metagraph-l0`:

- metagraph-l0.jar // The executable for the mL0 layer
- genesis.csv // The initial token balance allocations
- genesis.snapshot // The genesis snapshot created locally
- genesis.address // The metagraph address created in the genesis snapshot

In `code/currency-l1`:

- currency-l1.jar // The executable for the cL1 layer

In `code/data-l1`:

- data-l1.jar // The executable for the dL1 layer

`hydra remote-start`

This method initiates the remote startup of your metagraph in one of the available networks: integrationnet or mainnet. The network should be set in `euclid.json` under `deploy -> network`

To begin the remote startup of the metagraph, we utilize the parameters configured in euclid.json (`network`, `gl0_node -> ip`, `gl0_node -> id`, `gl0_node -> public_port`,

`ansible -> hosts`, and `ansible -> playbooks -> start`). The startup process unfolds as follows:

1. Termination of any processes currently running on the metagraph ports, which by default are 7000 for ml0, 8000 for cl1, and 9000 for dl1 (you can change on `hosts.ansible.yml`).
2. Relocation of any existing logs to a folder named `archived-logs`, residing within each layer directory: `metagraph-l0`, `currency-l1`, and `data-l1`.
3. Initiation of the `metagraph-l0` layer, with `node-1` designated as the genesis node.
4. Initial startup as `genesis`, transitioning to `rollback` for subsequent executions. To force a genesis startup, utilize the `--force_genesis` flag with the `hydra remote-start` command. This will move the current `data` directory to a folder named `archived-data` and restart the metagraph from the first snapshot.
5. Detection of missing files required for layer execution, such as `:your_file.p12` and `metagraph-l0.jar`, triggering an error and halting execution.
6. Following the initiation of `metagraph-l0`, the l1 layers, namely `currency-l1` and `data-l1`, are started. These layers only started if present in your project.

After the script completes execution, you can verify if your metagraph is generating snapshots by checking the block explorer of the selected network:

- Integrationnet: https://be-integrationnet.constellationnetwork.io/currency/:your_metagraph_id/snapshots/latest
- Mainnet: https://be-mainnet.constellationnetwork.io/currency/:your_metagraph_id/snapshots/latest

You can verify if the cluster was successfully built by accessing the following URL:

`http://{your_host_ip}:{your_layer_port}/cluster/info`

Replace:

- `{your_host_ip}`: Provide your host's IP address.
- `{your_layer_port}`: Enter the public port you assigned to each layer.

Each layer directory on every node contains a folder named `logs`. You can monitor and track your metagraph logs by running:

```
tail -f logs/app.log
```

NOTE: Don't forget to add your hosts' information, such as host, user, and SSH key file, to your `infra/ansible/remote/hosts.ansible.yml` file.

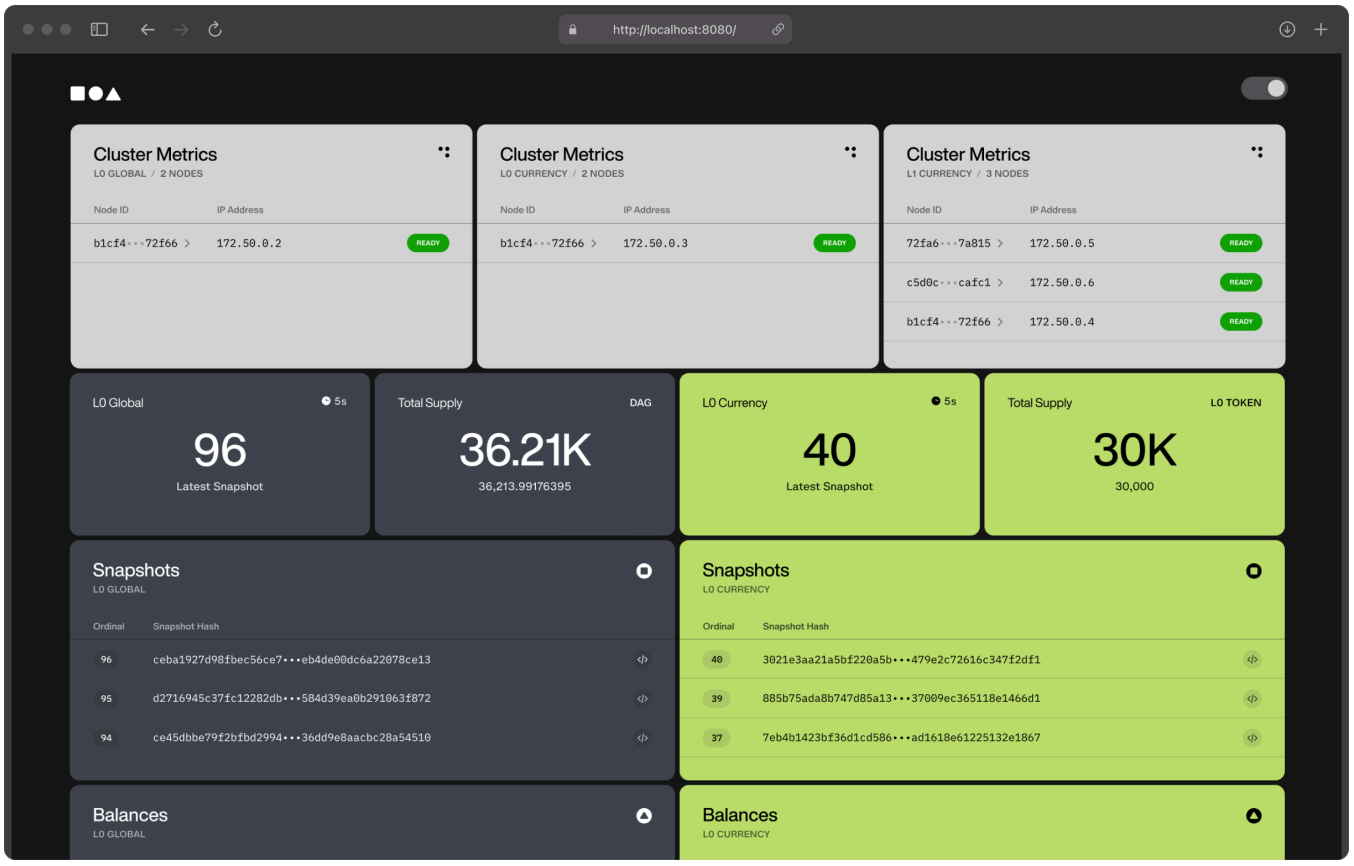
`hydra remote-status`

This method will return the status of your remote hosts. You should see the following:

```
################################### Node 1 ###################################
Metagraph L0
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId

Currency L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId

Data L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId


################################### Node 2 ###################################
Metagraph L0
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId
```

```
Currency L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId

Data L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId


################################## Node 3 ###################################
Metagraph L0
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId

Currency L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId

Data L1
URL: http://:your_node_ip:your_port/node/info
State: :state
Host: :host
Public port: :your_port
P2P port: :your_port
Peer id: :peerId
```

Developer Dashboard



The Euclid Developer Dashboard is a tool crafted specifically for developers working with the Constellation ecosystem. The dashboard presents a unified view of the status of local and deployed clusters, allowing developers to easily monitor their projects' progress and the flow of data between network layers.

Engineered to seamlessly integrate with the Euclid Development Environment, the dashboard serves as an excellent starting point for constructing bespoke developer tools tailored to the unique requirements of your project. Developed using NextJS and Tailwind CSS, the codebase promotes rapid development, ensuring a smooth and efficient workflow for developers. Regardless of whether you're working on a small-scale project or building a sophisticated application, this dashboard helps you stay informed about your project's status while also laying the groundwork for the creation of additional tools designed to optimize your development process.

Setup

Clone the project from Github

```
git clone https://github.com/Constellation-Labs/sdk-developer-dashboard.git
cd sdk-developer-dashboard
```

Install dependencies

```
npm install
```

Run the project

```
npm run dev
```

Open a browser and navigate to

```
http://localhost:8080
```

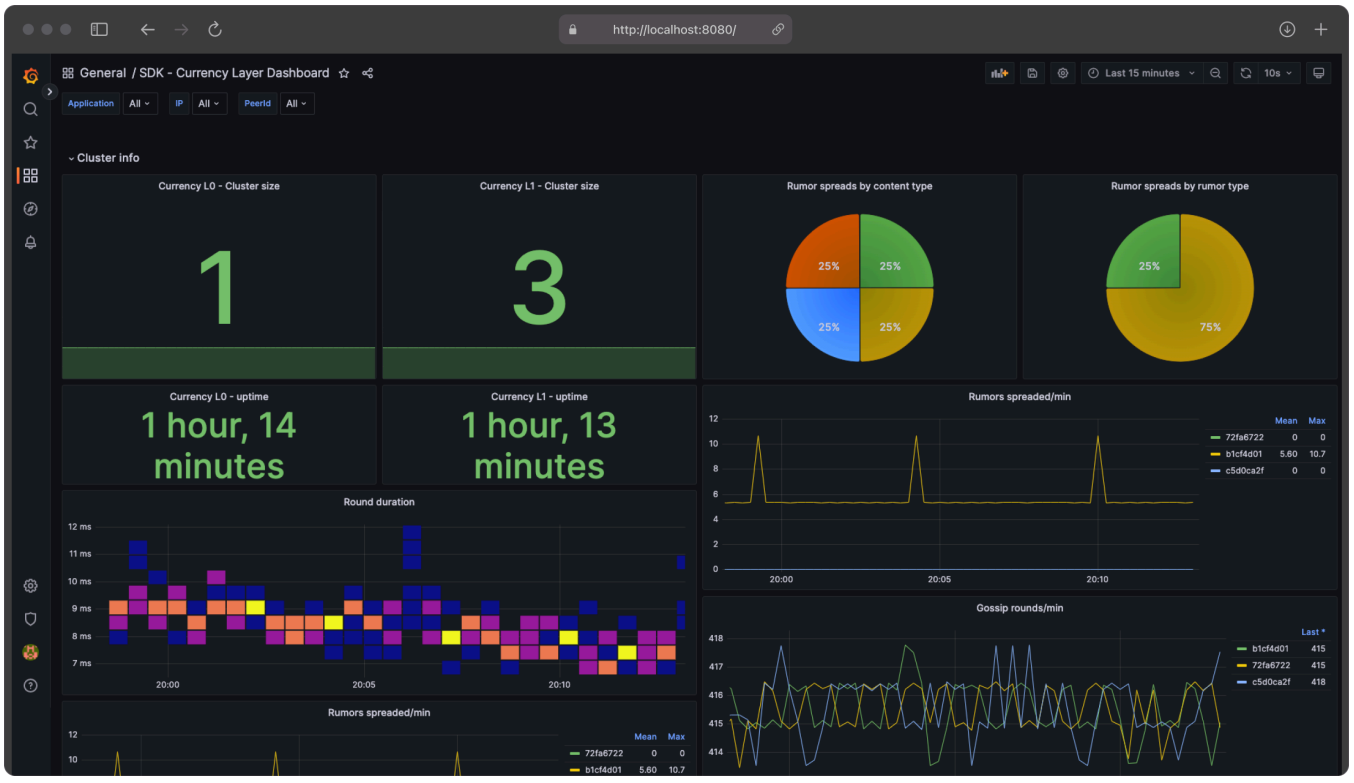
Configuration

The `.env` file in the root of the project contains defaults that work with the Euclid Development Environment out of the box but you can edit the defaults to match your desired configuration.

```
L0_GLOBAL_URL=http://localhost:9000
L0_CURRENCY_URL=http://localhost:9200
L1_CURRENCY_URL=http://localhost:9300
```

Edit this page

Telemetry Dashboard



The Telemetry Dashboard is an essential tool provided as part of the SDK to allow project teams to monitor network health for metagraph development. It consists of dashboard templates to track global network health (Global L0 and DAG L1) and local metagraph health (Currency L0 and Currency L1). The Telemetry Dashboard is built on a Grafana instance using data collected from the network via Prometheus.

The dashboard provides a visual representation of network health and helps developers to quickly identify any issues or bottlenecks in the system. By monitoring key metrics such as consensus duration, gossip round frequency, and transaction throughput, developers can make informed decisions about how to optimize their metagraphs networks and improve overall performance.

Installation

The Telemetry dashboard is included as part of the Euclid Development Environment. Once the framework is installed, it can be started with Hydra. To start the dashboard, ensure that the `start_grafana_container` option is enabled in `euclid.json`:

```
...
  "docker": {
    "start_grafana_container": true
  }
...
```

Then, you can initiate the system from genesis with the following command:

```
scripts/hydra start-genesis
```

Alternatively, to start from a rollback, execute:

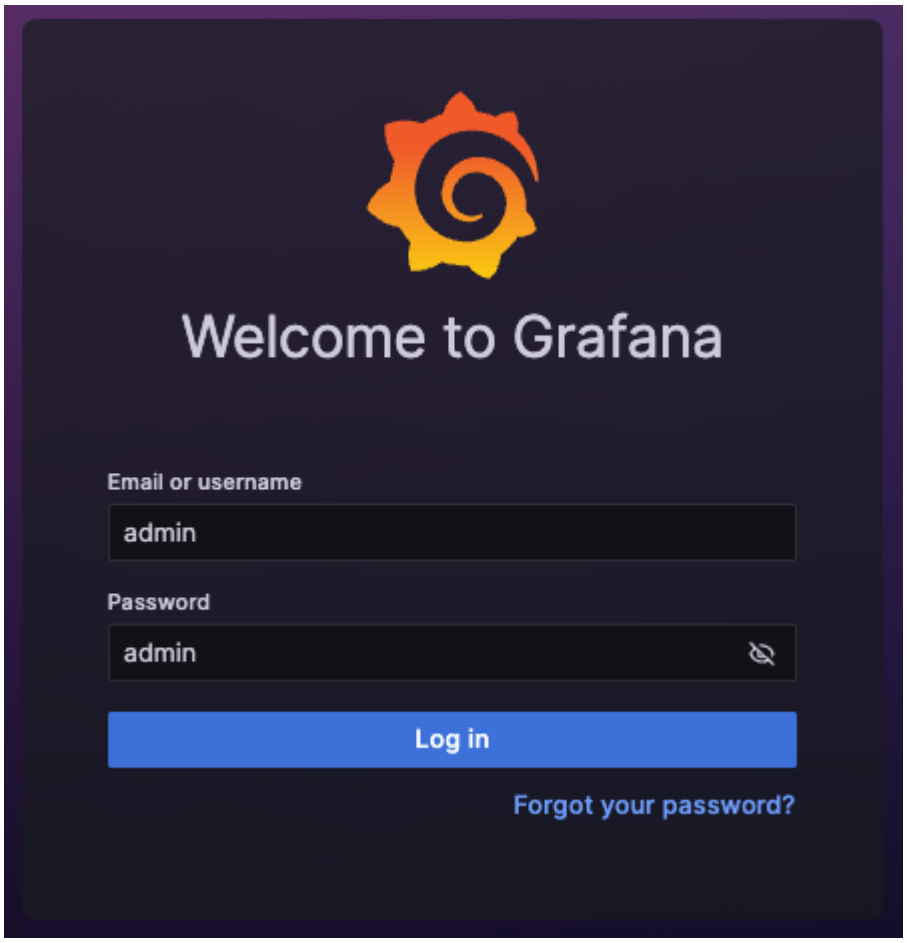
```
scripts/hydra start-rollback
```

By default, Grafana runs on port 3000 and can be accessed at the following url

```
http://localhost:3000
```

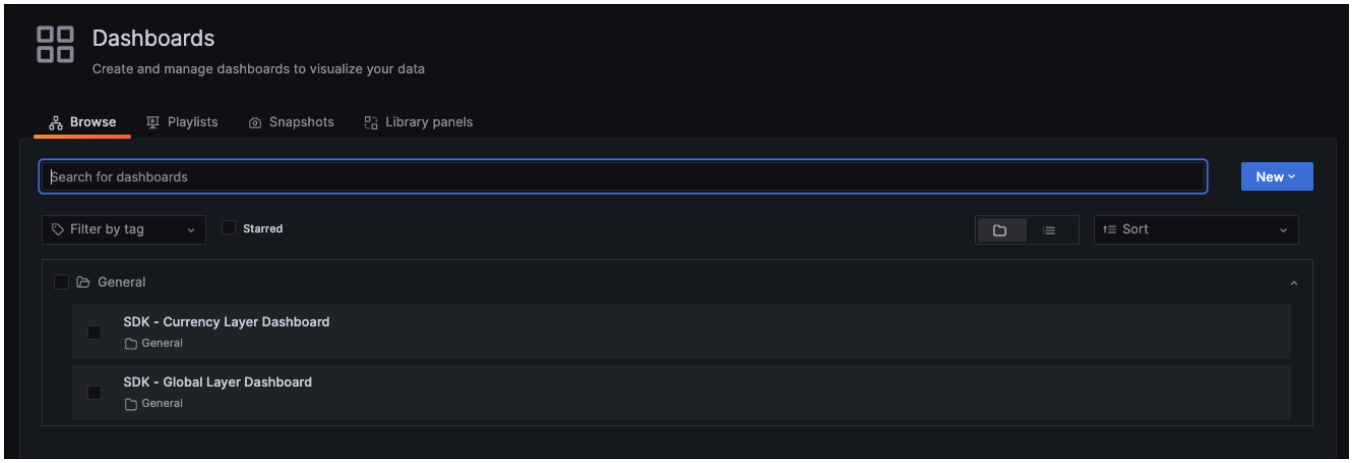
Setup

The default username and password both “admin”.



Dashboards

You can find the default dashboards in the “Dashboards” section on the left menu.



By default, the following templates are included:

- Global Layer dashboard - Information about the Global L0 and DAG L1 networks.
- Currency Layer dashboard - Information about the Currency L0 and Currency L1 networks.

Edit this page

Metagraph Monitoring Service

We have introduced a monitoring tool in version `v0.10.0` of the Euclid Development Environment project that can assess the health of your metagraph and initiate restarts if necessary.

A healthy metagraph requires a minimum of three nodes per layer: `metagraph-l0`, `currency-l1`, and `data-l1`. The `currency-l1` and `data-l1` layers should be implemented if your metagraph requires these layers, but at least three nodes per layer are necessary for correct operation.

Sometimes, your node can be unhealthy for several reasons, as examples:

- The remote host becoming stuck in a process.
- The remote host shutting down unexpectedly.
- The node experiencing a fork.
- The metagraph sending snapshots to a node that subsequently becomes unhealthy.

Each of these conditions may require attention and, in some cases, intervention, such as a restart.

For example, consider a scenario where your metagraph is operating normally but sends a `MetagraphSnapshot` to a `global-l0` node that has forked on the network. If this node is not part of the main and valid fork, your `MetagraphSnapshot` will never reach the `global-l0` layer. The monitoring service will detect this issue and automatically trigger a metagraph restart.

In another scenario, if the `currency-l1` process stops on one of your nodes, the monitoring service will detect this anomaly and initiate a restart for the affected node on that layer.

Introduction

The service is developed using `NodeJS`, and all necessary dependencies are installed on your remote instance during deployment.

Running in the background with PM2, the service initiates checks at intervals specified in the configuration under the field: `check_healthy_interval_in_minutes`. It evaluates the health of the metagraph based on predefined and customizable `restart-conditions`, detailed in the metagraph-monitoring-service repository. For example, if an unhealthy node is detected, the service triggers a restart.

The service runs health checks on a regular interval and evaluates the metagraph cluster on a set of predefined or developer-created health criteria (restart-conditions). Restart conditions can target the whole cluster, a specific layer, or individual nodes to ensure that the metagraph is operating properly in each case. If an issue is detected, the service uses SSH to access the impacted node(s) and restart their process and rejoin them to the network.

Installation

This tool does not ship by default with Euclid, so it must be installed before use. To install within a Euclid project, run the following:

```
hydra install-monitoring-service
```

this command to creates a monitoring project in your source directory, which will be named `metagraph-monitoring-service`

To use this feature, information about the remote hosts to be monitored must be configured in the `infra/ansible/remote/hosts.ansible.yml` file under the monitoring section. The monitoring service must have SSH access to the other nodes with a user with sudo privileges without requiring a password. Refer to this document to learn how to enable password-less sudo for a user.

Monitoring Configuration

Before deploying to remote instances, configure your monitoring by editing the `config/config.json` file located in the root of the `metagraph-monitoring-service` directory. After installing the service, when you run the install command, some fields will be automatically populated based on the `euclid.json` file. These fields include:

- `metagraph.id`: The unique identifier for your metagraph.
- `metagraph.name`: The name of your metagraph.
- `metagraph.version`: The version of your metagraph.
- `metagraph.default_restart_conditions`: Specifies conditions under which your metagraph should restart. These conditions are located in `src/jobs/restart/conditions`, including:
- `SnapshotStopped`: Triggers if your metagraph stops producing snapshots.
- `UnhealthyNodes`: Triggers if your metagraph nodes become unhealthy.
- `metagraph.layers`:
- `ignore_layer`: Set to `true` to disable a specific layer.
- `ports`: Specifies public, P2P, and CLI ports.
- `additional_env_variables`: Lists additional environment variables needed upon restart, formatted as `["TEST=MY_VARIABLE, TEST_2=MY_VARIABLE_2"]`.
- `seedlist`: Provides information about the layer seedlist, e.g., `{ base_url: ":your_url", file_name: ":your_file_name"}`.
- `metagraph.nodes`:
- `ip`: IP address of the node.
- `username`: Username for SSH access.
- `privateKeyPath`: Path to the private SSH key, relative to the service's root directory. Example: `config/your_key_file.pem`.
- `key_file`: Details of the `.p12` key file used for node startup, including `name`, `alias`, and `password`.
- `network.name`: The network your metagraph is part of, such as `integrationnet` or `mainnet`.

- `network.nodes` : Information about the GL0s nodes.
- `check_healthy_interval_in_minutes` : The interval, in minutes, for running the health check.

NOTE: You must provide your SSH key file that has access to each node. It is recommended to place this under the `config` directory. Ensure that this file has access to the node and that the user you've provided also has sudo privileges without a password.

Customize Monitoring

Learn how to customize your monitoring by checking the repositories:

- metagraph-monitoring-service-package: This repository houses the `npm` package that provides core restart functionalities.
- metagraph-monitoring-service: This repository utilizes the aforementioned package to implement a basic restart functionality.

Deploying Monitoring

Once you've configured your metagraph monitoring, deploy it to the remote host with:

```
hydra remote-deploy-monitoring-service
```

This command sends your current monitoring service from euclid to your remote instance and downloads all necessary dependencies.

Starting Monitoring

After deployment, start your monitoring with:

```
hydra remote-start-monitoring-service
```

To force a complete restart of your metagraph, use:

```
hydra remote-start-monitoring-service --force-restart
```

Edit this page

Overview

Euclid's Metagraph Framework is a rapid development framework specifically designed for creating metagraph applications within the Constellation Network ecosystem. Written in Scala, this framework offers a robust and flexible environment for building advanced blockchain solutions. The framework supports two key modules: the Currency Module and the Data Module, enabling developers to create metagraphs with comprehensive L0 token support and custom data processing capabilities.

Framework Features

The Metagraph Framework (also known as the Currency Framework) provides a complete blockchain-in-a-box solution for the creation of a layer 1 DLT network. It offers a stable starting point for application developers, while allowing full customization of the codebase to meet specific business or personal application goals.

Key Benefits

Customization and Interoperability

One of the core strengths of the Metagraph Framework is its complete customization capability. Developers have full control over the metagraph codebase, enabling the addition of custom validation logic, support for various external data types and ingestion formats, and the integration of external Scala or Java packages to accelerate development by leveraging existing libraries and tools.

The framework also facilitates seamless interoperability with other metagraphs, allowing for the effortless adoption of network standards and best practices. This ensures metagraph projects are adaptable and upgradable to make use of future network functionality.

Scalability and Security

The Metagraph Framework promotes best practices that support high levels of scalability and security. The hybrid DAG/linear-chain architecture of HGTP ensures efficient handling of large transaction volumes and complex data processes, making it suitable for high-frequency and data-intensive applications.

Integrated Development Toolkit

The Metagraph Framework works seamlessly with the rest of the Euclid SDK, providing developers with a comprehensive toolkit for local development and testing. This includes tools for remote deployment and cluster monitoring in production environments, ensuring that developers can easily transition from development to production.

Modules

Currency Module

The Currency Module adds L0 token support to a metagraph project. Developers get default token functionality out of the box, with the ability to customize key network logic around token minting, distribution, and implementation of fees.

Data Module

The Data Module allows developers to define and manage custom data types, validation logic, and data structures for their metagraphs. This capability enables the creation of sophisticated data pipelines and processing mechanisms, while giving developers complete control over data storage and privacy.

Getting Started

Explore the following sections to gain an understanding of how Metagraph Framework applications are structured, the core concepts of their development, and how to best leverage their powerful development constructs to create secure, scalable, and decentralized blockchain applications.

By utilizing the Metagraph Framework, developers can rapidly build and deploy metagraph applications that meet the evolving demands of the Web3 ecosystem, ensuring a durable and future-proof foundation for their projects.

Framework Archictecture

This section contains information about the Metagraph Framework and its relation to the deployed architecture of a metagraph.

In order to understand how the framework functions, it is important to understand the multi-layered architecture of a metagraph. Make sure you're familiar with Metagraph Architecture before continuing.

Code Structure

A new metagraph project generated from the currency template will have the following module directory structure:

```
- modules/
- - l0/
- - - Main.scala
- - l1/
- - - Main.scala
- - data_l1/
- - - Main.scala
- - shared_data
- - - Main.scala
```

Let's break down each directory and its function.

L0

This directory contains a `Main.scala` file with a Main object instance that extends `CurrencyL0App`. `CurrencyL0App` contains overridable functions that allow for customization of the operation of the metagraph L0 layer including validation, snapshot formation, and

management of off chain state. It also contains the `rewards` overridable function that allows for minting of new tokens on the network.

> **Note**
>
> While the class `CurrencyL0App` has the "Currency" in its name, it defines the L0 layer through which both Currency L1 and Data L1 data flows.

L1

This directory contains a `Main.scala` file with a Main object instance that extends `CurrencyL1App`. `CurrencyL1App` contains overridable functions relevant to customizing token validation behavior such as `transactionValidator`.

L1 Data

This directory also contains a `Main.scala` file, with a Main object instance that extends `CurrencyL1App`. In order to have this L1 behave as a DataApplication, the `dataApplication` method should be overridden with your custom configuration. The metagraph examples repo has implemented examples to reference, for example the NFT example.

Shared Data

This directory contains an empty Main.scala file but is provided as a suggestion for application directory structure. Several of the lifecycle functions are run on both Data L1 and on L0. For example, serializers/deserializers, validators, and data types will likely shared between layers. Organizing them in a separate directory makes their use in multiple layers clear.

See the Data Application Lifecycle for more information.

Installation

The Metagraph Framework can be installed in several ways:

- **Euclid** (recommended): Install an empty project in Euclid SDK using the `hydra install` command.
- **Metagraph Examples** (recommended): Explore ready-to-use examples of metagraph codebases in the metagraph-examples repo. These examples can also be installed automatically via the `hydra install-template` command.
- **giter8**: The Metagraph Framework is distributed as a g8 template project that can be customized for your organization. This template can be manually built using giter8. For more details, visit the project repository.

> **Quick Start**
>
> See the Euclid Quick Start guide for a walkthrough of framework installation within the Euclid Development Environment. This is the recommended development environment and installation method for most users.

Installation Using Giter8

> **Note**
>
> Manual installation using giter8 necessitates pre-generated Tessellation dependencies. To generate these dependencies, execute the following commands in the tessellation repository on the desired tag:

```
sbt shared/publishM2 kernel/publishM2 keytool/publishM2 nodeShared/publishM2
```

```
dagL1/publishM2 currencyL0/publishM2 currencyL1/publishM2
```

Ensure Scala and giter8 are installed. Install giter8 with:

```
./cs install giter8
```

Then, install the template using the specified tag:

```
# replace v2.8.0 with the version to install
g8 Constellation-Labs/currency --tag "v2.8.0"
```

Compiling the Project

After installing your project and the Tessellation dependencies, compile the project to generate your local JAR files. You can compile as follows:

For metagraph-l0:

```
sbt currencyL0/assembly
```

For currency-l1:

```
sbt currencyL1/assembly
```

For data-l1:

```
sbt dataL1/assembly
```

Installation Using Metagraph Examples

Using Euclid, you can execute the following command to list the available examples:

```
hydra install-template --list
```

To install the desired template, execute this command:

```
# replace 'nft' with the name of the desired template
hydra install-template nft
```

Compiling the Project

To compile the project using Euclid, you just need to run:

```
hydra build
```

Edit this page

Currency Module

The Currency Module provides a fast and customizable way to launch a metagraph with native support for currency (L0 token) transactions. Unlike closed-loop systems such as smart-contract platforms, this framework offers enhanced flexibility by allowing direct modifications of application-level code. It encapsulates functionalities necessary for peer-to-peer token transactions, chain data construction, and more, in an easy-to-use package.

Note that launching a token is not required when developing with the Metagraph Framework, and at present, the Metagraph Framework is also the most efficient way to launch a data-focused application with or without a token associated with it. Extending the framework with the Data module allows developers to build metagraphs with custom data ingestion, validation, and storage logic. See the Data Application section for more detail.

By basing their metagraphs on this framework, developers gain the advantage of working with a proven set of underlying features and functionality in order to be able to focus on their project's own business logic rather than boilerplate functionality. Example projects are provided in order to speed up development. See metagraph examples or use the `install-template` method of `hydra`.

Working with Tokens

This section will cover the use of L0 tokens within a metagraph - how to mint them, how to determine fees for user actions, and different strategies for state management in relation to token distribution.

Minting Tokens

Tokens can be minted in two ways through a metagraph: at genesis through the use of the *genesis file*, or as part of a incremental snapshot using the *rewards* function.

Genesis File

The genesis file is a file used during the creation of a genesis snapshot, i.e. the first snapshot of the chain, which contains initial wallet balances. Configuring the genesis file sets the initial circulating supply for the network and assigns that supply to specific wallets.

The genesis file is a simple CSV file that can be found in `source/metagraph-l0/genesis/genesis.csv` with the format of

```
<ADDRESS>,<BALANCE>
```

Euclid comes with a default set of addresses and balances in this file. You can (and should) edit for your own needs however you like.

NOTE: The balances in the genesis file are denominated in datum rather than DAG. 1 DAG is 100,000,000 datum. So for example to set a balance of 25 tokens to a wallet, you would add the following line: `DAG123..., 2500000000`

The genesis file is used only once, during the formation of the genesis snapshot, and as such you only have one chance to set your initial balances. Changing the contents of this file once the snapshot chain has progressed beyond the first snapshot will not have any effect on token balances on your network.

Once the metagraph state has progressed beyond the genesis snapshot (ordinal 1+), any changes to the token balance map must come through either token transactions or new minting of tokens through the rewards function.

Rewards Function

The rewards function is a function of the `CurrencyL0App` called during the Metagraph Snapshot Lifecycle which has the ability to create Reward transactions. Reward transactions are special minting transactions on the network which increase the circulating supply of the L0 token and distribute it to an address.

If we examine the function in `modules→l0→Main.scala`, we can see that the function is provided with the following context:

- `Signed[CurrencyIncrementalSnapshot]`
- `SortedMap[Address, Balance]`
- `SortedSet[Signed[Transaction]]`
- `ConsensusTrigger`
- `Set[CurrencySnapshotEvent]`
- `Option[DataCalculatedState]`

The expected output of the rewards function is a SortedSet with reward transactions. Using the context data provided, a number of strategies for token minting are possible.

Supported token minting strategies:

- Continuous minting to reward network participation (ex: rewarding validator nodes that participate in consensus)

- State-triggered minting (ex: minting rewards to wallets based on data fetched on a schedule)
- One-off minting (ex: an airdrop or one-time creation of a wallet).

See the Reward API metagraph example repo for an example of how to use the rewards function.

Setting Metagraph Fees

Fees charged by the metagraph fall into two categories: Token transaction fees, and fee transactions charged for custom data updates. These perform similar actions on different kinds of transactions but have unique ways that they need to be configured and managed.

> **Note**
>
> All fees collected on a metagraph are denominated in the metagraph's token as the currency. Fees are not collected or managed in DAG.

Token Transaction Fees

By default, L0 tokens share the fee characteristics of DAG. DAG has zero required fee for transfers, but adding a minimal fee (1+ datum) will prioritize the processing of a transaction above non-fee transactions on the network. Priority processing also allows many more transactions to be processed from the same address in a single snapshot: 100 per snapshot for priority vs 1 per snapshot for non-priority transactions. This default configuration allows senders to pay for increased throughput if needed, for example in airdrop or bulk sending use cases, while otherwise supporting a feeless network.

L0 tokens share these default attributes of DAG but can customize the minimum required fee for a specific transaction (default zero). This behavior can be managed with the `transactionValidator` function on the `CurrencyL1App` object. The `transactionValidator` function can either accept or reject a transaction based on any attributes of transaction, but is most commonly used to reject if the fee below a particular threshold.

[Tessellation v2.9.0+] An additional function, `transactionEstimator` returns the expected fee for a given transaction. This function is exposed over the `/estimate-fee` Currency L1 endpoint and is used by wallets and other 3rd party integrations to understand the fee required to send a successful transaction.

See the Custom Transaction Validation repo for an example implementation.

> **Note**
>
> Fees collected by the network are currently removed from circulation (in other words burned). While custom transaction fee behavior and especially destination wallets will likely be added in the future, it is possible to deposit fees into a specific wallet with current feature through a mint/burn mechanism. The `rewards` function can be used to mint the equivalent of the burned fees into a particular wallet by monitoring transactions in each snapshot.

Data Update Fees

> **V2.9.0+**
>
> FeeTransactions and associated functionality are currently set to be included in Tessellation v2.9.0. The description of functionality below applies only to that future version or later.

Metagraphs have the ability to require custom fees for data payloads submitted through the `/data` endpoint on a DataApplication. These fees allow the application to charge fees for certain actions such as creating or updating resources (minting) or based on the resources required to handle the request. By default, fees are set to zero.

Much like currency transactions, data transaction fees must be explicitly approved and signed by the private key representing the address of the requester. Since data updates are fully custom data types defined within the metagraph codebase, there is no inherently included fee

structure or predefined destination wallet for the fee to be transferred to within the data update that can be referenced.

Both the amount and destination wallet of the DataUpdate must be set through a `FeeTransaction` object nested in the DataUpdate request body. This object is signed independently of the DataUpdate and then included in the DataUpdate body, and then the entire DataUpdate body including the FeeTransaction is signed. This format allows the metagraph to validate the signature of the DataUpdate as whole, as well as the FeeTransaction independently. All FeeTransactions must be included in the metagraph on-chain data and shared with the gL0. Requiring a separate signature on the FeeTransaction itself preserves the metagraph developer's ability to exclude parts or all of the DataUpdate contents from the on-chain data.

FeeTransaction has the following format:

```
// FeeTransaction
{
  value: {
    destination: 'DAG...', // metagraph-defined fee wallet
    amount: 1 // amount of fee in datum
  },
  proofs: [] // signature of only FeeTransaction value
}
```

FeeTransaction included in an example data update:

```
// CustomDataUpdate
{
  value: {
    customField1: 42,
    customField2: "custom-value",  // any fields of the DataUpdate
    FeeTransaction: {
      value: {
        destination: 'DAG...', // metagraph-defined fee wallet
        amount: 1 // amount of fee in datum
      },
      proofs: [] // signature of only FeeTransaction value
    }
  },
```

```
  proofs: [] // signature of CustomDataUpdate value
}
```

In order to support wallets and other 3rd party services that need to know the cost of processing a DataUpdate before sending it, the `/data/estimate-fee` endpoint is provided. This endpoint accepts an unsigned DataUpdate and returns the expected fee and metagraph destination wallet for the fee.

For example:

```
**Request:**
POST /data/estimate-fee
{
  customField1: 42,
  customField2: "custom-value",  // any fields of the DataUpdate
}

**Response:**
{
  fee: 100,  // minimum required fee
  address: "DAG..." // metagraph-defined fee wallet
}
```

Edit this page

Data Application Module

The Data Application (or Data API) is a module available to Metagraph Framework developers to ingest and validate custom data types through their metagraphs. It's a set of tools for managing data interactions within the metagraph that is flexible enough to support a wide range of application use cases such as IoT, NFTs, or custom application-specific blockchains.

> **Example Code**
>
> Want to jump directly to a code example? A number of examples can be found on Github under the Metagraph Examples repo.

Basics

The basic function of a Data Application is to accept data through a special endpoint on the Data L1 layer, found at `POST /data`. Receiving a request triggers a series of lifecycle functions for the data update, running it through validation and consensus on both Data L1 and L0, and then eventual inclusion into the custom data portion of the metagraph snapshot.

This process is defined in detail in Lifecycle Functions but an abreviated version is provided below as an overview. In order to interact with the framework, developers can tap into these lifecycle events to override the default behavior and introduce their own custom logic.

Data Flow

1. Data accepted by the `/data` endpoint
2. Data is parsed (and reformatted if necessary) with `signedDataEntityEncoder`
3. Custom validations are run on Data L1 with `validateUpdate`

4. Data is packaged into blocks, run through L1 consensus and sent to L0
5. Additional custom validations are run w/L0 context available with `validateData`
6. Data is packaged into on-chain (snapshot) and off-chain (calculated state) representations with the `combine` function
7. The snapshot undergoes consensus and is accepted into the chain

See Lifecycle Functions for more detail.

State Management and Storage

State is defined within the Metagraph Framework in two ways: on-chain (snapshot) and off-chain (calculated state). The developer has control over how both kinds of state are created via the `combine` lifecycle function which is called prior to each round of L0 consensus.

On-chain state is stored in the metagraph's snapshot chain and each of these snapshots is submitted to the Global L0 for inclusion in a global snapshot. As such, on chain state incurs fees and has a maximum size of 500kb (see Limitations and Fees). This also means that any data stored in on-chain state is made public through the public nature of global snapshots on the Hypergraph. However, the developer has the option to encrypt that state through the `serializeState` lifecycle function, or alternatively, to limit the data that's stored in on-chain state.

Off-chain or "calculated" state is data that is stored off-chain but can be recreated by the accumulation of all snapshots in order from genesis to current. In this way, it's calculated from the chain but not part of the chain data itself. Calculated state is stored in memory by default, and recreated from a file-based cache on bootup, but the relevant lifecycle functions can be used to hook into other data stores such as a local database or an external storage service. Since calculated state is never sent to the Hypergraph, it does not incur any fees and has no limitations on size or structure beyond hardware limits.

See State Management for more details.

Querying Metagraph Data

Metagraphs support the creation of custom HTTP endpoints on any of the metagraph layers. These endpoints are useful for allowing external access to calculated state or creating views of the chain data for users.

See Custom Queries for more details.

Scheduled Tasks

Scheduled tasks on a metagraph are possible through the concept of daemons, worker processes that run on a timer. These processes allow the metagraph codebase to react to time-based triggers rather than waiting for an incoming transaction or data update to react to. Daemons are especially useful for syncing behavior, such as fetching data from an external source on a regular schedule or pushing internal data externally on a regular basis.

State Management

The DataAPI encompasses two distinct types of states: `OnChainState` and `CalculatedState`, each serving unique purposes.

- `OnChainState`: As the name implies, this state contains all the information intended to be permanently stored on the blockchain. This state
- `CalculatedState`: This state exists off-chain and is constructed based on incoming data. It is not stored on the blockchain

Creating State Classes

Each state described above represents functionality from the Data Application. To create these states, you need to implement custom `traits` provided by the Data Application:

- The `OnChainState` must extend the `DataOnChainState` trait.
- The `CalculatedState` must extend the `DataCalculatedState` trait.

Both traits, `DataOnChainState` and `DataCalculatedState`, can be found in the tessellation repository.

You can create your state classes within the `shared_data` module of your metagraph. To see examples of how these states are implemented, refer to the metagraph examples.

States are typically placed in the directory path: `modules -> shared_data -> types -> Types.scala`, but you are encouraged to organize your directories and files as suits your project's needs.

OnChainState x CalculatedState

As previously described, both `OnChainState` and `CalculatedState` serve distinct purposes. The `OnChainState` is designed to persist information that users want recorded on the blockchain, whereas the `CalculatedState` is intended to hold data that should remain off the blockchain.

Consider the example from the water and energy usage. project.

In this scenario, every update to water and energy usage is recorded. For instance, if we receive a request to update the water usage by 10 units, this information should be persisted on the blockchain using `OnChainState` to track the update accurately.

Shifting focus slightly, let's say there is a need to provide an encrypted private key for making requests to an external API, which only your metagraph can decrypt and execute. Suppose we need to check the energy expenditure from two months prior using this key. In this case, your metagraph decrypts the key, performs the external call, and retrieves the data. However, since this data is sensitive, it should not be stored on the blockchain. This is a perfect use case for `CalculatedState`, which conceals sensitive information from the blockchain while still allowing access to it.

The above example illustrates a basic application of these states. Further details on how to update these states will be provided in the subsequent sections.

Updating State

The DataAPI includes several lifecycle functions crucial for the proper functioning of the metagraph.

You can review all these functions in the Lifecycle Functions section.

In this discussion, we'll focus on the following functions: `combine` and `setCalculatedState`

combine

Is the central function to updating the states. This function processes incoming requests/updates by either increasing or overwriting the existing states. Here is the function's signature:

```
override def combine(
  currentState: DataState[OnChainState, CalculatedState],
  updates: List[Signed[Update]]
): IO[DataState[OnChainState, CalculatedState]]
```

The combine function is invoked after the requests have been validated at both layers (l0 and l1) using the `validateUpdate` and `validateData` functions.

The `combine` function receives the `currentState` and the `updates`

- `currentState` : As indicated by the name, this is the current state of your metagraph since the last update was received.
- `updates` : This is the list of incoming updates. It may be empty if no updates have been provided to the current snapshot.

The output of this function is also a state, reflecting the new state of the metagraph post-update. Therefore, it's crucial to ensure that the function returns the correct updated state.

Returning to the `water and energy usage` example, you can review the implementation of the combine function here. In this implementation, the function retrieves the current value of water or energy and then increments it based on the amount specified in the incoming request for the `CalculatedState` , while also using the current updates as the `OnChainState` .

setCalculatedState

Following the combine function and after the snapshot has been accepted and consensus reached, we obtain the `majority snapshot` . This becomes the official snapshot for the metagraph. At this point, we invoke the `setCalculatedState` function to update the `CalculatedState` .

This state is typically stored `in memory`, although user preferences may dictate alternative storage methods. You can explore the implementation of storing the `CalculatedState` in memory by checking the CalculatedState.scala and CalculatedStateService.scala classes, where we have detailed examples.

In the sections below, we will discuss `serializers` used to serialize the states.

Serializers/Deserializers

We also utilize other lifecycle functions for `serialize/deserialize` processes, each designed specifically for different types of states.

For the `OnChainState`, we use the following functions:

```
def serializeState(
  state: OnChainState
): F[Array[Byte]]

def deserializeState(
  bytes: Array[Byte]
): F[Either[Throwable, OnChainState]]
```

For the `CalculatedState` we have:

```
def serializeCalculatedState(
  state: CalculatedState
): F[Array[Byte]]

def deserializeCalculatedState(
  bytes: Array[Byte]
): F[Either[Throwable, CalculatedState]]
```

The `OnChainState` serializer is employed during the snapshot production phase, prior to consensus, when nodes propose snapshots to become the official one. Once the official

snapshot is selected, based on the majority, the `CalculatedState` serializer is used to serialize this state and store the `CalculatedState` on disk.

The deserialization functions are invoked when constructing states from the `snapshots/calculatedStates` stored on disk. For instance, when restarting a metagraph, it's necessary to retrieve the state prior to the restart from the stored information on disk.

In the following section, we will provide a detailed explanation about disk storage.

Disk Storage

When operating a Metagraph on layer 0 (ml0), a directory named `data` is created. This directory is organized into the following subfolders:

- `incremental_snapshot`: Contains the Metagraph snapshots.
- `snapshot_info`: Stores information about the snapshots, including internal states like balances.
- `calculated_state`: Holds the Metagraph calculated state.

Focusing on the `calculated_state`, within this folder, files are named after the snapshot ordinal. These files contain the CalculatedState corresponding to that ordinal. We employ a logarithmic cutoff strategy to manage the storage of these states.

This folder is crucial when restarting the Metagraph. It functions as a `checkpoint`: instead of rerunning the entire chain to rebuild the `CalculatedState`, we utilize the files in the `calculated_state` directory. This method allows us to rebuild the state more efficiently, saving significant time.

Data Privacy

As previously mentioned, the `CalculatedState` serves a crucial role by allowing the storage of any type of information discreetly, without exposing it to the public. This functionality is particularly useful for safeguarding sensitive data. When you use the `CalculatedState`, you

can access your information whenever necessary, but it remains shielded from being recorded on the blockchain.. This method offers an added layer of security, ensuring that sensitive data is not accessible or visible on the decentralized ledger.

By leveraging `CalculatedState`, organizations can manage proprietary or confidential information such as personal user data, trade secrets, or financial details securely within the metagraph architecture. The integrity and privacy of this data are maintained, as it is stored in a secure compartment separated from the public blockchain.

Scalability

Metagraphs face a constraint concerning the size of snapshots: `they must not exceed 500kb`. If snapshots surpass this threshold, they will be rejected, which can impose significant limitations on the amount of information that can be recorded on the blockchain.

This is where the CalculatedState becomesparticularly valuable. It allows for the storage of any amount of data, bypassing the size constraints of blockchain snapshots. Moreover, CalculatedState offers flexibility in terms of storage preferences,enabling users to choose how and where their data is stored.

This functionality not only alleviates the burden of blockchain size limitations but also enhances data management strategies. By utilizing CalculatedState, organizations can efficiently manage larger datasets, secure sensitive information off-chain, and optimize their blockchain resources for critical transactional data.

Lifecycle functions

Lifecycle functions are essential to the design and operation of a metagraph within the Euclid SDK. These functions enable developers to hook into various stages of the framework's lifecycle, allowing for the customization and extension of the core functionality of their Data Application. By understanding and implementing these functions, developers can influence how data is processed, validated, and persisted, ultimately defining the behavior of their metagraph.

How Lifecycle Functions Fit into the Framework

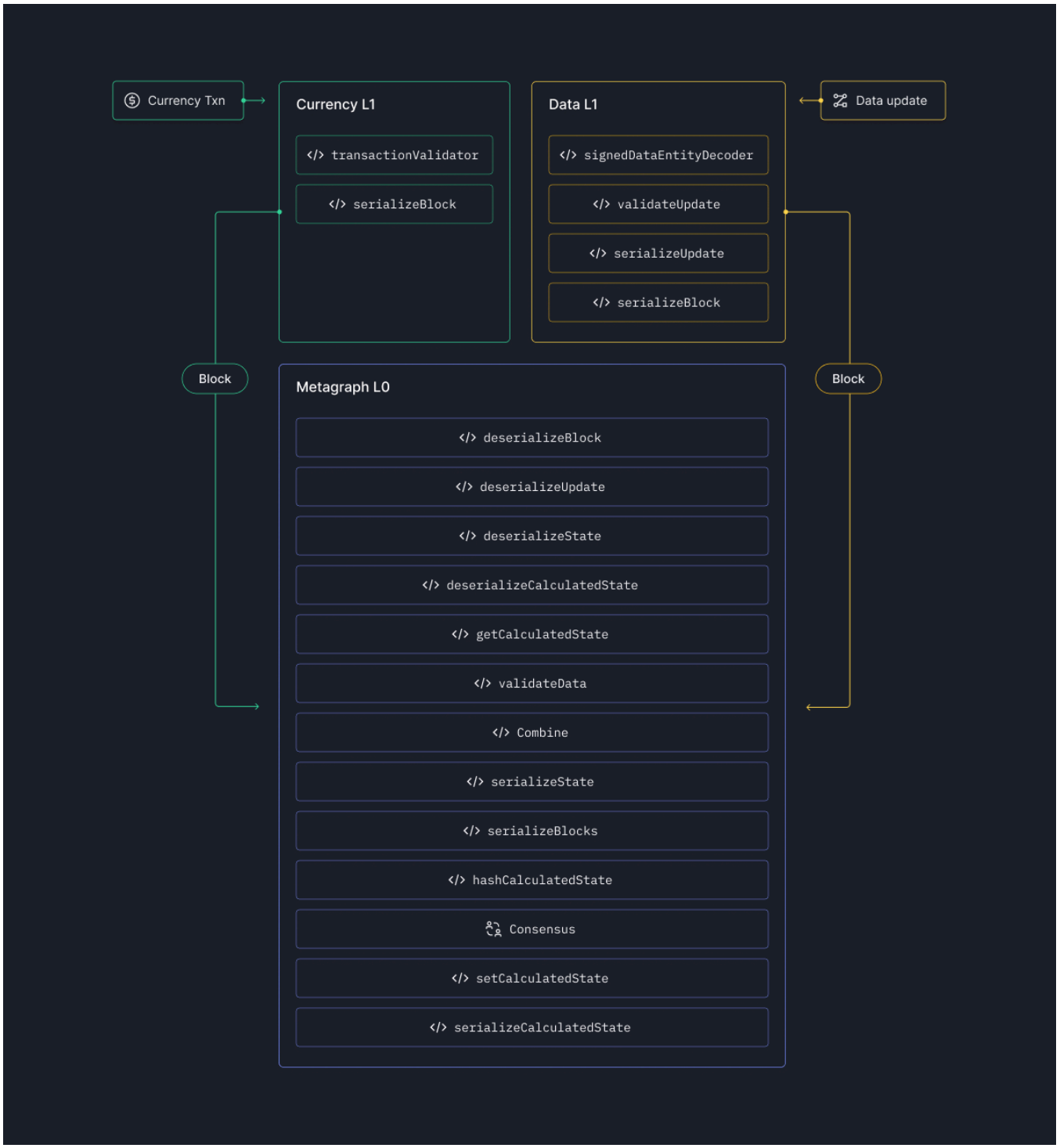
In the Euclid SDK, lifecycle functions are organized within the L0 (DataApplicationL0Service), Currency L1 (CurrencyL1App), and Data L1 (DataApplicationL1Service) modules. These modules represent different layers of the metagraph architecture:

- **L0 Layer:** This is the base layer responsible for core operations like state management, validation, and consensus. Functions in this layer are critical for maintaining the integrity and consistency of the metagraph as they handle operations both before (`validateData` , `combine`) and after consensus (`setCalculatedState`).
- **Data L1 Layer:** This layer manages initial validations and data transformations through the /data endpoint. It is responsible for filtering and preparing data before it is sent to the L0 layer for further processing.
- **Currency L1 Layer:** This layer handles initial validations and transaction processing through the /transactions endpoint before passing data to the L0 layer. It plays a crucial role in ensuring that only valid and well-formed transactions are forwarded for final processing. Note that currency transactions are handled automatically by the framework and so only a small number of lifecycle events are available to customize currency transaction handling (`transactionValidator` , etc.).

By implementing lifecycle functions in these layers, developers can manage everything from the initialization of state in the `genesis` function to the final serialization of data blocks. Each function serves a specific purpose in the metagraph's lifecycle, whether it’s validating incoming data, updating states, or handling custom logic through routes and decoders.

Lifecycle Overview

The diagram below illustrates the flow of data within a metagraph, highlighting how transactions and data updates move from the Currency L1 and Data L1 layers into the L0 layer. The graphic also shows the sequence of lifecycle functions that are invoked at each stage of this process. By following this flow, developers can understand how their custom logic integrates with the framework and how data is processed, validated, and persisted as it progresses through the metagraph.



Note

Note that some lifecycle functions are called multiple times and across L1 and L0 layers. It is usually recommended to create a common, shared implementation for these functions.

Functions

genesis

Data Applications allow developers to define custom state schemas for their metagraph. Initial states are established in the `genesis` function within the `l0` module's `DataApplicationL0Service`. Use the `OnChainState` and `CalculatedState` methods to define the initial schema and content of the application `state` for the `genesis snapshot`.

For example, you can set up your initial states using map types, as illustrated in the Scala code below:

```
class OnChainState(updates: List[Update]) extends DataOnChainState
class CalculatedState(info: Map[String, String]) extends DataCalculatedState

override def genesis: DataState[OnChainState, CalculatedState] =
DataState(OnChainState(List.empty), CalculatedState(Map.empty))
```

In the code above, we set the initial state to be:

- `OnChainState`: Empty list
- `CalculatedState`: Empty Map

signedDataEntityDecoder

This method parses custom requests at the `/data` endpoint into the `Signed[Update]` type. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers. By default, you can use the `circeEntityDecoder` to parse the JSON:

```
{
  "value": {
    // This type is defined by your application code
  },
  "proofs": [{
    "id": "<public key>",
    "signature": "<signature of data in value key above>"
  }]
}
```

The default implementation is straightforward:

```
def signedDataEntityDecoder[F[_] : Async: Env]: EntityDecoder[F, Signed[Update]] =
circeEntityDecoder
```

For custom parsing of the request, refer to the example below:

```
  def signedDataEntityDecoder[F[_] : Async: Env]: EntityDecoder[F, Signed[Update]] =
{
    EntityDecoder.decodeBy(MediaType.text.plain) { msg =>
    // Assuming msg.body is a comma-separated string of key-value pairs.
      val dataMap = msg.body.split(",").map { pair =>
        val Array(key, value) = pair.split(":")
        key.trim -> value.trim
      }.toMap

      val update = Update(dataMap.value)
      val hexId = Hex(dataMap.pubKey)
      val hexSignature = Hex(dataMap.signature)
      val signatureProof = SignatureProof(Id(hexId), Signature(hexSignature))
      val proofsSet = SortedSet(signatureProof)

      val proofs = NonEmptySet.fromSetUnsafe(proofsSet)
      Signed(update, proofs)
    }
  }
```

In this custom example, we parse a simple string formatted as a map, extracting the `value`, `pubKey`, and `signature` necessary to construct the `Signed[Update]`. This method allows for efficient handling of incoming data, converting it into a structured form ready for further processing.

validateUpdate

This method validates the update on the L1 layer and can return synchronous errors through the `/data` API endpoint. Context information (oldState, etc.) is not available to this method so validations need to be based on the contents of the update only. Validations requiring context

should be run in `validateData` instead. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, validate a field is within a positive range:

```
def validateUpdate(update: Update): IO[DataApplicationValidationErrorOr[Unit]] = IO {
  if (update.usage <= 0) {
    DataApplicationValidationError.invalidNec
  } else {
    ().validNec
  }
}
```

The code above rejects any update that has the update value less than or equal to 0.

validateData

This method runs on the L0 layer and validates an update (data) that has passed L1 validation and consensus. `validateData` has access to the old or current application state, and a list of updates. Validations that require access to state information should be run here. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, validate that a user has a balance before allowing an action:

```
def validateData(oldState: DataState[OnChainState, CalculatedState], updates:
NonEmptyList[Signed[Update]]): IO[DataApplicationValidationErrorOr[Unit]] = IO {
  updates
    .map(_.value)
    .map {
      val currentBalance = acc.balances.getOrElse(update.address, 0)

      if (currentBalance > 0) {
        ().validNec
      } else {
        DataApplicationValidationError.invalidNec
      }
    }
    .reduce
}
```

The code above rejects any update that the current balance is lower than 0.

combine

The `combine` method accepts the current state and a list of validated updates and should return the new state. This is where state is ultimately updated to generate the new snapshot state. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, subtract one from a balance map:

```
def combine(oldState: DataState[OnChainState, CalculatedState], updates:
NonEmptyList[Signed[Update]]): IO[State] = IO {
  updates.foldLeft(oldState) { (acc, update) =>
    val currentBalance = acc.balances.getOrElse(update.address, 0)

    acc.focus(_.balances).modify(_.updated(update.address, currentBalance - 1))
  }
}
```

The code above will subtract one for the given address and update the state

serializeState and deserializeState

These methods are required to convert the onChain state to and from byte arrays, used in the snapshot, and the OnChainState class defined in the genesis method. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, serialize to/from a State object:

```
  def serializeState(state: OnChainState): IO[Array[Byte]] = IO {
    state.asJson.deepDropNullValues.noSpaces.getBytes(StandardCharsets.UTF_8)
  }

  def deserializeState(bytes: Array[Byte]): IO[Either[Throwable, OnChainState]] = IO
{
    parser.parse(new String(bytes, StandardCharsets.UTF_8)).flatMap { json =>
      json.as[OnChainState]
```

```
    }
  }
```

The codes above serialize and deserialize using `Json`

serializeCalculatedState and deserializeCalculatedState

These methods are essential for converting the CalculatedState to and from byte arrays. Although the `CalculatedState` does not go into the snapshot, it is stored in the `calculated_state` directory under the `data` directory. For more details, refer to the State Management section. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, serialize to/from a State object:

```
  def serializeCalculatedState(state: CalculatedState): IO[Array[Byte]] = IO {
    state.asJson.deepDropNullValues.noSpaces.getBytes(StandardCharsets.UTF_8)
  }

  def deserializeCalculatedState(bytes: Array[Byte]): IO[Either[Throwable,
CalculatedState]] = IO {
    parser.parse(new String(bytes, StandardCharsets.UTF_8)).flatMap { json =>
      json.as[CalculatedState]
    }
  }
```

The codes above serialize and deserialize using `Json`

serializeUpdate and deserializeUpdate

These methods are required to convert updates sent to the `/data` endpoint to and from byte arrays. Signatures are checked against the byte value of the `value` key of the update so these methods give the option to introduce custom logic for how data is signed by the client. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, serialize to/from a JSON update:

```
  def serializeUpdate(update: Update): IO[Array[Byte]] = IO {
    update.asJson.deepDropNullValues.noSpaces.getBytes(StandardCharsets.UTF_8)
  }

  def deserializeUpdate(bytes: Array[Byte]): IO[Either[Throwable, Update]] = IO {
    parser.parse(new String(bytes, StandardCharsets.UTF_8)).flatMap { json =>
      json.as[Update]
    }
  }
```

The codes above serialize and deserialize using `Json`

serializeBlock and deserializeBlock

These methods are required to convert the data application blocks to and from byte arrays, used in the snapshot.
It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

For example, serialize to/from a State object:

```
  def serializeBlock(block: Signed[DataApplicationBlock]): IO[Array[Byte]] = IO {
    state.asJson.deepDropNullValues.noSpaces.getBytes(StandardCharsets.UTF_8)
  }

  def deserializeBlock(bytes: Array[Byte]): IO[Either[Throwable,
Signed[DataApplicationBlock]]] = IO {
    parser.parse(new String(bytes, StandardCharsets.UTF_8)).flatMap { json =>
      json.as[Signed[DataApplicationBlock]]
    }
  }
```

The codes above serialize and deserialize using `Json`

setCalculatedState

This function updates the `calculatedState`. For details on when and why this function is called, refer to the State Management section. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

```
  override def setCalculatedState(
    ordinal: SnapshotOrdinal,
    state  : CalculatedState
  )(implicit context: L0NodeContext[IO]): IO[Boolean] = {
      val currentCalculatedState = currentState.state
      val updated = state.devices.foldLeft(currentCalculatedState.devices) {
        case (acc, (address, value)) =>
          acc.updated(address, value)
      }

      CalculatedState(snapshotOrdinal, CalculatedState(updated))
    }.as(true)
```

The code above simply replaces the current address with the new value, thereby overwriting it.

getCalculatedState

This function retrieves the `calculatedState`. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

```
  override def getCalculatedState(implicit context: L0NodeContext[IO]):
IO[(SnapshotOrdinal, CheckInDataCalculatedState)] =
  currentState.state.map(calculatedState => (calculatedState.ordinal,
calculatedState.state))
```

The code above is an example of how to implement the retrieval of `calculatedState`.

hashCalculatedState

This function hashes the `calculatedState`, which is used for `proofs`. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

```
  override def hashCalculatedState(
    state: CalculatedState
  )(implicit context: L0NodeContext[IO]): IO[Hash] = {
    val jsonState = state.asJson.deepDropNullValues.noSpaces
```

```
    Hash.fromBytes(jsonState.getBytes(StandardCharsets.UTF_8))
  }
```

The code above is an example of how to implement the hashing of `calculatedState`.

dataEncoder and dataDecoder

Custom encoders/decoders for the updates. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

```
def dataEncoder: Encoder[Update] = deriveEncoder
def dataDecoder: Decoder[Update] = deriveDecoder
```

The code above uses the `circe` semiauto deriveEncoder and deriveDecoder

calculatedStateEncoder and calculatedStateDecoder

Custom encoders/decoders for the calculatedStates. It should be implemented in both `Main.scala` files for the `l0` and `data-l1` layers

```
def calculatedStateEncoder: Encoder[CalculatedState] = deriveEncoder
def calculatedStateDecoder: Decoder[CalculatedState] = deriveDecoder
```

The code above uses the `circe` semiauto deriveEncoder and deriveDecoder

Framework Endpoints

A metagraph functions similarly to a traditional back-end server, interacting with the external world through HTTP endpoints with specific read (GET) and write (POST) functionalities. While a metagraph is decentralized by default and backed by an on-chain data store, it operates much like any other web server. This section outlines the default endpoints available to developers to interact with their metagraph.

See also Custom Queries for information on how to create your own metagraph endpoints.

Framework Endpoints

Below is a list of available endpoints made available by default through the Metagraph Framework. Each endpoint is hosted by a node running either the Metagraph L0, Currency L1, or Data L1.

This is not an exhaustive list of available endpoints, please see Metagraph APIs for more information and links to the OpenAPI specifications of each API.

Universally Available Endpoints

These endpoints are available on all (mL0, cL1, and dL1) APIs and are useful for debugging and monitoring purposes.

Method	Endpoint	Description
GET	/node/info	Returns info about the health and connectivity state of a particular node. This is useful for understanding if a node is connected to its layer of the network and its ready state.

Method	Endpoint	Description
GET	/cluster/info	Returns info about the cluster of nodes connected to the node's layer of the network. This is useful for understanding how many nodes are connected at each layer and diagnosing issues related to cluster health.

Metagraph L0

Endpoints available on metagraph L0 nodes.

Method	Endpoint	Description
GET	/snapshots/latest	Returns the latest incremental snapshot created by the metagraph. Incremental snapshots contain only changes since the previous snapshot. This endpoint also supports returning snapshots at specific ordinals with the format `GET /snapshots/:ordinal`.
GET	/snapshots/latest/combined	Returns the latest full snapshot of the metagraph which includes some calculated values. This shows the complete state of the metagraph at that moment in time.
GET	/currency/:address/balance	Returns the balance of a particular address on the metagraph at the current snapshot.
GET	/currency/total-supply	Returns the total number of tokens in circulation at the current snapshot. Note that "total supply" in this case is total supply created currently. It doesn't represent max supply of the token.

Currency L1

Endpoints available on currency L1 nodes.

Method	Endpoint	Description
POST	/transactions	Accepts signed L0 token transactions.
GET	/transactions/:hash	Returns a single transaction by hash if the transaction is in the node's mempool waiting to be processed. Does not have access to non-pending transactions.
GET	/transactions/last-reference/:address	Returns the lastRef value for the provided address. LastRef is necessary for constructing a new transaction.
POST	/estimate-fee	Returns the minimum fee required give the (unsigned) currency transaction.

Data L1

Method	Endpoint	Description
POST	/data	Accepts custom-defined data updates.
GET	/data	Returns all data updates in mempool waiting to be processed.
POST	/data/estimate-fee	(v2.9.0+) Returns the minimum fee and destination address to process the given (unsigned) custom data update.

Custom Endpoints

Metagraph developers have the ability to define their own endpoints to add additional functionality to their applications. Custom endpoints are supported on each of the layers, with different contextual data and scalability considerations for each. These endpoints can be used to provide custom views into snapshot state, or for any custom handling that the developer wishes to include as part of the application.

Defining a Route

A route can be defined by overriding the `routes` function available on `DataApplicationL0Service` or `DataApplicationL1Service`, creating endpoints on the metagraph L0 node or data L1 node, respectively. Custom routes are defined as instances of http4s `HttpRoutes`.

Here is a minimal example that shows how to return a map of currency addresses with a balance on the metagraph. The example accesses the `addresses` property of L0 chain context and returns it to the requester.

```
  // modules/l0/.../l0/Main.scala

  override def routes(implicit context: L0NodeContext[IO]): HttpRoutes[IO] =
HttpRoutes.of {
    case GET -> Root / "addresses" =>
      OptionT(context.getLastCurrencySnapshot)
        .flatMap(_.dataApplication.toOptionT)
        .flatMapF(da => deserializeState(da.onChainState).map(_.toOption))
        .value
        .flatMap {
          case Some(value) => Ok(value.addresses)
          case None => NotFound()
        }
  }
```

For a slightly more complex example, the code below shows how to return the Data Application's calculated state from an endpoint. It also shows a more common pattern for route definition which moves route definitions to their own file, defined as a case class extending `Http4sDsl[F]`. Note that `calculatedStateService` is not available as part of `L0NodeContext` so it must be passed to the case class.

```
  // modules/l0/.../l0/Main.scala
  override def routes(implicit context: L0NodeContext[IO]): HttpRoutes[IO] =
CustomRoutes[IO](calculatedStateService).public

  // modules/l0/.../l0/CustomRoutes.scala
  case class CustomRoutes[F[_] : Async](calculatedStateService:
CalculatedStateService[F]) extends Http4sDsl[F] with PublicRoutes[F] {
    @derive(encoder, decoder)
    case class CalculatedStateResponse(
      ordinal        : Long,
      calculatedState: CheckInDataCalculatedState
    )

    private def getLatestCalculatedState: F[Response[F]] = {
      calculatedStateService.getCalculatedState
        .map(state => CalculatedStateResponse(state.ordinal.value.value,
state.state))
        .flatMap(Ok(_))
    }

    private val routes: HttpRoutes[F] = HttpRoutes.of[F] {
      case GET -> Root / "calculated-state" / "latest" => getLatestCalculatedState
    }
```

```
    val public: HttpRoutes[F] =
      CORS
        .policy
        .withAllowCredentials(false)
        .httpRoutes(routes)

    override protected def prefixPath: InternalUrlPrefix = "/"
  }
```

Custom Route Prefix

All custom defined routes exist under a prefix, shown in the example above as `Root`. By default this prefix is `/data-application`, so for example you might define an `addresses` route which would be found at `http://<base-url>:port/data-application/addresses`.

It is possible to override the default prefix to provide your own custom prefix by overriding the `routesPrefix` method.

For example, to use the prefix "/d" instead of "/data-application":

```
  override def routesPrefix: ExternalUrlPrefix = "/d"
```

Examples

For more complete examples of custom route implementations, see Example Codebases.

Edit this page

Quick Start Guide

This guide will walk you through the process of setting up a minimal development environment using the Euclid Development Environment project, installing the Metagraph Framework, and launching clusters. The process should take less than an hour, including installing dependencies.

> **Windows Support**
>
> Primary development focus for this SDK is based on UNIX-based operating systems like macOS or Linux. With that being said, Windows support is available using the Windows Subsystem for Linux (WSL) to emulate a UNIX environment. The following guide has been tested in that environment and works wells.
>
> See Install WSL for more detail in setting up WSL on your Windows machine.

Install Dependencies

Install Basic Dependencies

Many developers can skip this step because these dependencies are already installed.

- Node JS
- Docker
- Cargo
- Ansible
- Scala 2.13
- Jq

- Yq

Install argc

```
cargo install argc
```

Install Giter

```
cs install giter8
```

Configure Docker

The Euclid Development Environment starts up to 10 individual docker containers to create a minimal development environment which takes some significant system resources. Configure docker to make at least 8GB of RAM available. If you are using Docker Desktop, this setting can be found under Preferences -> Resources.

Create a Github Access Token

See instructions for how to create an access token. The token only needs `read:packages` scope. Save this token for later, it will be added as an environment variable.

Install

Clone

Clone the Euclid Development Environment project to your local machine.

```
git clone https://github.com/Constellation-Labs/euclid-development-environment
cd euclid-development-environment
```

See the Development Environment section for an overview of the directory structure of the project.

Configure

Edit the `github_token` variable within the `euclid.json` file with your Github Access Token generated previously. Update the `project_name` field to the name of your project.

Hydra

Familiarize yourself with the `hydra` CLI. We can use the `hydra` CLI tool to build the necessary docker containers and manage our network clusters.

```
scripts/hydra -h

USAGE: hydra <COMMAND>

COMMANDS:
  install                          Installs a local framework and detaches project
  install-template                 Installs a project from templates
  build                            Build containers
  start-genesis                    Start containers from the genesis snapshot
(erasing history) [aliases: start_genesis]
  start-rollback                   Start containers from the last snapshot
(maintaining history) [aliases: start_rollback]
  stop                             Stop containers
  destroy                          Destroy containers
  purge                            Destroy containers and images
  status                           Check the status of the containers
  remote-deploy                    Remotely deploy to cloud instances using Ansible
[aliases: remote_deploy]
  remote-start                     Remotely start the metagraph on cloud instances
using Ansible [aliases: remote_start]
  remote-status                    Check the status of the remote nodes
  update                           Update Euclid
  logs                             Get the logs from containers
  install-monitoring-service       Download the metagraph-monitoring-service
(https://github.com/Constellation-Labs/metagraph-monitoring-service) [aliases:
install_monitoring_service]
  remote-deploy-monitoring-service  Deploy the metagraph-monitoring-service to remote
host [aliases: remote_deploy_monitoring_service]
  remote-start-monitoring-service   Start the metagraph-monitoring-service on remote
host [aliases: remote_start_monitoring_service]
```

Install Project

Running the `install` command will do two things:

- Creates currency-l0 and currency-l1 projects from a g8 template and moves them to the `source/project` directory.
- Detach your project from the source repo.

Detaching your project from the source repo removes its remote git configuration and prepares your project to be included in your own version control. Once detached, your project can be updated with `hydra`.

```
scripts/hydra install
```

You can import a metagraph template from custom examples by using the following command:

```
scripts/hydra install-template
```

By default, we use the Metagraph Examples repository. You should provide the template name when running this command. To list the templates available to install, type:

```
scripts/hydra install-template --list
```

Build

Build your network clusters with hydra. By default, this builds `metagraph-ubuntu`, `metagraph-base-image`, and `prometheus` + `grafana` monitoring containers. These images will allow deploy the containers with metagraph layers: `global-l0`, `metagraph-l0`, `currency-l1`, and

`data-l1`. The `dag-l1` layer is not built by default since it isn't strictly necessary for metagraph development. You can include it on the `euclid.json` file.

Start the build process. This can take a significant amount of time... be patient.

```
scripts/hydra build
```

Run

After your containers are built, go ahead and start them with the `start-genesis` command. This starts all network components from a fresh genesis snapshot.

```
scripts/hydra start-genesis
```

Once the process is complete you should see output like this:

```
######################################################################
######################### METAGRAPH INFO ##########################

Metagraph ID: :your_id


Container metagraph-node-1 URLs
Global L0: http://localhost:9000/node/info
Metagraph L0: http://localhost:9200/node/info
Currency L1: http://localhost:9300/node/info
Data L1: http://localhost:9400/node/info


Container metagraph-node-2 URLs
Metagraph L0: http://localhost:9210/node/info
Currency L1: http://localhost:9310/node/info
Data L1: http://localhost:9410/node/info


Container metagraph-node-3 URLs
Metagraph L0: http://localhost:9220/node/info
Currency L1: http://localhost:9320/node/info
Data L1: http://localhost:9420/node/info
```

```
Clusters URLs
Global L0: http://localhost:9000/cluster/info
Metagraph L0: http://localhost:9200/cluster/info
Currency L1: http://localhost:9300/cluster/info
Data L1: http://localhost:9400/cluster/info

######################################################################
```

You can also check the status of your containers with the `status` command.

```
scripts/hydra status
```

Next Steps

You now have a minimal development environment installed and running 🎉

Send your first transaction

Set up the FE Developer Dashboard and send your hello world metagraph transaction.

Manual Setup

Prefer to configure your environment by hand? Explore manual setup.

Edit this page

Send a Transaction

In this guide, we will explore two of the tools that work together with the Euclid Developer Environment, then use them to send and track our first metagraph token transaction.

We will install the Developer Dashboard, send a transaction using an included script, and monitor our clusters using the Telemetry Dashboard.

Before You Start

This guide assumes that you have configured your local environment based on the Quick Start Guide and have at least your `global-l0`, `currency-l0`, `currency-l1`, and `monitoring` clusters running.

Install the SDK Developer Dashboard

The Developer Dashboard is a frontend dashboard built with NextJS and Tailwind CSS. It comes with default configuration to work with the Development Environment on install.

Clone the repository

```
git clone https://github.com/Constellation-Labs/sdk-developer-dashboard.git
cd sdk-developer-dashboard
```

Install dependencies

*Node 16 recommended

```
yarn install
```

```
npm install
```

Start the app

```
yarn dev
(or npm run dev)
```

View the Developer Dashboard

Open a brower window to `http://localhost:8080`.

Here, you can see both your currency and global clusters at work. You should see the snapshot ordinals for the Global L0 and the Currency L0 increment on your dashboard. Also notice that you can inspect each snapshot to see its contents. Any transactions sent on the network will appear in the tables below - there are separate tables for DAG and Metagraph Token transactions.

The dashboard is designed to work with the Euclid Development Environment default settings out-of-the-box but if you need to change network settings, they can be found in the `.env` file at the root of the project.

Send a Transaction

The Developer Dashboard comes pre-installed with scripts to send transactions to your running metagraph. The scripts use dag4.js to interact with the network based on the settings in your `.env` file.

Single Transaction

Single transactions can be sent on the command line for easy testing. The transaction below should succeed with the default configuration.

```
yarn metagraph-transaction:send --seed="drift doll absurd cost upon magic plate often
actor decade obscure smooth" --transaction='{"destination":
"DAG4o41NzhfX6DyYBTTXu6sJa6awm36abJpv89jB","amount":99, "fee":0}'
```

Bulk Transactions

You can send bulk transactions to the network by calling `send-bulk` and providing a path to a json file with transaction configuration. A sample JSON file is provided for you which will work with the default configuration.

```
yarn metagraph-transaction:send-bulk --
config="./scripts/send_transactions/batch_transactions.example.json"
```

View Transactions

Return to the dashboard and look in the Currency Transactions table. You should see the transactions you just sent. You can also view the contents of the snapshot that the transaction's block was included in.

Monitoring

Now that you have sent a transaction or two we can check on the stability of the network with the Telemetry Dashboard. The Telemetry Dashboard is composed of two containers included as part of the Development Environment: a Prometheus instance and a Grafana instance.

The dashboard is hosted on the Grafana instance which can be accessed at `http://localhost:3000/`.

The initial login and password are:

```
username: admin
password: admin
```

The Grafana instance includes two dashboards which can be found in the menu on the left. One dashboard monitors the Global L0 and DAG L1 (if you have it running). The other monitors the Currency L0 and Currency L1. More information can be found in the Telemetry Dashboard section.

Manual Setup

This guide walks through the detailed process of manually creating a minimal development environment using docker containers and manual configuration. Most developers will be more productive using the automatic setup and configuration of the Euclid Development Environment with the Hydra CLI. The following is provided for project teams looking to create their own custom configurations outside the default development environment.

Generate P12 Files

Generate your own `p12` files following the steps below: (java 11 must be installed)

1. Download the cl-keytool file here
2. We need to generate 3 `p12` files: 1 for Genesis Nodes (Global L0, Currency L0, and Currency L1 - 1), 1 for second node on cluster (Currency L1 - 2), 1 for third node on cluster (Currency L1 - 3).
3. Export the follow variables on your terminal with the values replaced to generate the first `p12` file.

```
export CL_KEYSTORE=":name-of-your-file.p12"
export CL_KEYALIAS=":name-of-your-file"
export CL_PASSWORD=":password"
```

1. Run the following instruction:

```
java -jar cl-keytool.jar generate
```

1. This will generate the first file for you

2. Change the variables CL_KEYSTORE, CL_KEYALIAS, and CL_PASSWORD and repeat the step 2 more times
3. At the end you should have 3 `p12` files

Common Steps

Create Containers

With Docker installed on your machine run:

```
docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -it -p 9000:9000 -p 9001:9001 -p
9002:9002--name :container_name_global_l0 --entrypoint "/bin/bash" ubuntu:20.04
docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -it -p 9100:9000 -p 9101:9001 -p
9102:9002--name :container_name_currency_l0 --entrypoint "/bin/bash" ubuntu:20.04
docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -it -p 9200:9000 -p 9201:9001 -p
9202:9002--name :container_name_currency_l1_1 --entrypoint "/bin/bash" ubuntu:20.04
docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -it -p 9300:9000 -p 9301:9001 -p
9302:9002--name :container_name_currency_l1_2 --entrypoint "/bin/bash" ubuntu:20.04
docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -it -p 9400:9000 -p 9401:9001 -p
9402:9002--name :container_name_currency_l1_3 --entrypoint "/bin/bash" ubuntu:20.04
```

Replace the `:containername*` with the name that you want for your container

Create a Docker Network

We need to create a docker custom network by running the following:

```
docker network create custom-network-tokens
```

Build the Container Libs

In your container run the following instructions:

```
apt-get update
```

```
apt install openjdk-11-jdk -y #jdk 11 to run the jars
apt-get install curl -y #install curl
apt-get install wget -y #install wget
apt-get install gnupg -y #used to add sbt repo
apt-get install vim -y #used to edit files, you can use the editor that you want to

echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | tee
/etc/apt/sources.list.d/sbt.list
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | tee
/etc/apt/sources.list.d/sbt_old.list
curl -sL "https://keyserver.ubuntu.com/pks/lookup?
op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | apt-key add

apt-get update
apt-get install sbt -y #install sbt
```

The instructions above install the dependencies to run correctly the node.

Tesselation repository

First, configure and export the `GITHUB_TOKEN` environment variable that will be required to pull some of the necessary packages. The access token needs only the `read:packages` scope. See how to create the personal access token here:

Creating a personal access token - GitHub Docs

Setup the `GITHUB_TOKEN` variable

```
export GITHUB_TOKEN=github_token
```

Then clone the repository:

```
git clone https://github.com/Constellation-Labs/tessellation.git
git checkout v2.0.0-alpha.2
```

Warning

Make sure you're using the latest version of Tessellation. You can find the most recent release in **here**.

Move to the tesselation folder and checkout to branch/version that you want. You can skip the `git checkout :version` if you want to use the develop default branch

```
cd tesselation
git checkout :version
```

Global L0

- Here is the instructions to run specifically Global L0 container.
- Move the `p12` file to container with the instruction:

```
docker cp :directory-of-p12-file container-name:file-name.p12
```

- Inside the docker container make sure that your p12 file exists correctly
- It should be at the root level (same level as the tessellation folder)
- Move to tessellation folder:

```
cd tessellation/
```

- Generate the jars

```
sbt core/assembly wallet/assembly
```

- Check the logs to see which version of global-l0 and wallet was published. It should be something like this:

```
/tessellation/modules/core/target/scala-2.13/tessellation-core-assembly-*.jar
```

- Move these jars to the root folder, like the example below

```
mv codebase/tessellation/modules/core/target/scala-2.13/tessellation-core-assembly-*
global-l0.jar
mv codebase/tessellation/modules/wallet/target/scala-2.13/tessellation-wallet-
assembly-* cl-wallet.jar
```

- Run the following command to get the clusterId (**store this information**):

```
java -jar cl-wallet.jar show-id
```

- Run the following command to get the clusterAddress (**store this information**):

```
java -jar cl-wallet.jar show-address
```

- Outside the container, run this following command to get your docker container IP

```
docker container inspect :container_name | grep -i IPAddress
```

- Outside the container, we need to join our container to the created network, you can do this with the following command (outside the container)

```
docker network connect custom-network-tokens :container_name
```

- You can check now your network and see your container there:

```
docker network inspect custom-network
```

- Fill the environment variables necessary to run the container (from your first `p12` file):

```
export CL_KEYSTORE=":name-of-your-file.p12"
export CL_KEYALIAS=":name-of-your-file"
export CL_PASSWORD=":password"
export CL_APP_ENV=dev
export CL_COLLATERAL=0
export CL_ENV=dev
```

- Create one empty genesis file in root directory too (you can add wallets and amounts if you want to):

```
touch genesis.csv
```

- Finally, run the jar:

```
java -jar global-l0.jar run-genesis genesis.csv --ip :ip_of_your_container
```

- Your should see something like this:

```
23:26:53.013 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App environment: Dev
23:26:53.052 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App version: 2.0.0-a
[WARNING] Your CPU is probably starving. Consider increasing the granularity
of your delays or adding more cedes. This may also be a sign that you are
unintentionally running blocking I/O operations (such as File or InetAddress)
without the blocking combinator.
[WARNING] Your CPU is probably starving. Consider increasing the granularity
of your delays or adding more cedes. This may also be a sign that you are
unintentionally running blocking I/O operations (such as File or InetAddress)
without the blocking combinator.
23:27:03.051 [io-compute-5] INFO  o.t.s.a.TessellationIOApp - Self peerId:
b1cf4d017eedb3e187b4d17cef9bdbcfdb2e57b26e346e9186da3a7a2b9110d73481fedbc6de23db51fb932371
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.esotericsoftware.kryo.util.UnsafeUtil (file:/coc
```

```
java.nio.DirectByteBuffer(long,int,java.lang.Object)
WARNING: Please consider reporting this to the maintainers of com.esotericsoftware.kryo.ut
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
WARNING: All illegal access operations will be denied in a future release
23:27:04.670 [io-compute-5] INFO  o.t.s.a.TessellationIOApp - Seedlist disabled.
23:27:18.263 [io-compute-1] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
0.0.0.0:9000
23:27:18.270 [io-compute-1] INFO  o.t.s.r.MkHttpServer - HTTP Server name=public started a
23:27:18.315 [io-compute-2] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
0.0.0.0:9001
23:27:18.316 [io-compute-2] INFO  o.t.s.r.MkHttpServer - HTTP Server name=p2p started at /
23:27:18.357 [io-compute-1] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
127.0.0.1:9002
23:27:18.359 [io-compute-1] INFO  o.t.s.r.MkHttpServer - HTTP Server name=cli started at /
23:27:20.400 [io-compute-3] INFO  o.t.s.i.c.d.N.$anon - Node state changed to=Ready{}
```

- That's all for the global-l0 container

Currency L0

- Here is the instructions to run specifically Currency L0 container.
- Move the `p12` file to container with the instruction:

```
docker cp :directory-of-p12-file container-name:file-name.p12
```

- Inside the docker container make sure that your p12 file exists correctly
- It should be at the root level (same level as the tessellation folder)
- Move to tessellation folder:

```
cd tessellation/
```

- Generate the jars

```
sbt currencyL0/assembly
```

- Check the logs to see which version of currency-l0 was published. It should be something like this:

```
/tessellation/modules/currency-l0/target/scala-2.13/tessellation-currency-l0-assembly-*.jar
```

- Move this jar to the root folder, like the example below

```
mv codebase/tessellation/modules/core/target/scala-2.13/tessellation-currency-l0-assembly-* currency-l0.jar
```

- Outside the container, run this following command to get your docker container IP

```
docker container inspect :container_name | grep -i IPAddress
```

- Outside the container, we need to join our container to the created network, you can do this with the following command (outside the container)

```
docker network connect custom-network-tokens :container_name
```

- You can check now your network and see your container there:

```
docker network inspect custom-network
```

- Fill the environment variables necessary to run the container (from your first `p12` file):

```
export CL_KEYSTORE=":name-of-your-file.p12"
export CL_KEYALIAS=":name-of-your-file"
export CL_PASSWORD=":password"
export CL_GLOBAL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
```

```
export CL_L0_TOKEN_IDENTIFIER=:id_got_of_command_cl_wallet_show_address
export CL_PUBLIC_HTTP_PORT=9000
export CL_P2P_HTTP_PORT=9001
export CL_CLI_HTTP_PORT=9002
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip-global-l0-container
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_APP_ENV=dev
export CL_COLLATERAL=0
```

- Create one genesis file in root directory too (you can add wallets and amounts if you want to):

```
touch genesis.csv
```

- You should edit this `genesis.csv` to add your addresses and amounts. You can use `vim` for that:

```
vim genesis.csv
```

- Example of genesis content:

```
DAG8pkb7EhCkT3yU87B2yPBunSCPnEdmX2Wv24sZ,1000000000000
DAG4o41NzhfX6DyYBTTXu6sJa6awm36abJpv89jB,1000000000000
DAG4Zd2W2JxL1f1gsHQCoaKrRonPSSHLgcqD7osU,1000000000000
```

- Finally, run the jar:

```
java -jar currency-l0.jar run-genesis genesis.csv --ip :ip_of_current_container
```

- Your should see something like this:

```
```

```
23:28:33.769 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App
environment: Dev
23:28:33.829 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App version:
2.0.0-alpha.2
23:29:25.489 [io-compute-2] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform -
Ember-Server service bound to address: 0.0.0.0:9000
23:29:25.520 [io-compute-2] INFO  o.t.s.r.MkHttpServer - HTTP Server name=public
started at /0.0.0.0:9000
23:29:25.606 [io-compute-2] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform -
Ember-Server service bound to address: 0.0.0.0:9001
23:29:25.608 [io-compute-2] INFO  o.t.s.r.MkHttpServer - HTTP Server name=p2p started
at /0.0.0.0:9001
23:29:25.795 [io-compute-3] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform -
Ember-Server service bound to address: 127.0.0.1:9002
23:29:25.796 [io-compute-3] INFO  o.t.s.r.MkHttpServer - HTTP Server name=cli started
at /127.0.0.1:9002
23:29:44.671 [io-compute-3] INFO  o.t.c.l.s.s.G.$anon - Genesis binary
3c02294a7a3c7b3a8f2af8c9633a82af46430cda7ffc2de0fc0c6f19afb497e0 and
57a4f918ce8228be1282834ece3e6f69ad87d69b42857dbb227b5e6441b25025 accepted and sent to
Global L0
```

- That's all for the currency-l0 container

Currency L1 - 1

- Here is the instructions to run specifically Currency L1 - 1 container.
- Move the `p12` file to container with the instruction:

```
docker cp :directory-of-p12-file container-name:file-name.p12
```

- Inside the docker container make sure that your p12 file exists correctly
- It should be at the root level (same level as the tessellation folder)
- Move to tessellation folder:

```
cd tessellation/
```

- Generate the jars

```
sbt currencyL1/assembly
```

- Check the logs to see which version of currency-l1 was published. It should be something like this:

```
/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-l1-assembly-*.jar
```

- Move this jar to the root folder, like the example below

```
mv codebase/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-l1-assembly-* currency-l1.jar
```

- Outside the container, run this following command to get your docker container IP

```
docker container inspect :container_name | grep -i IPAddress
```

- Outside the container, we need to join our container to the created network, you can do this with the following command (outside the container)

```
docker network connect custom-network-tokens :container_name
```

- You can check now your network and see your container there:

```
docker network inspect custom-network
```

- Fill the environment variables necessary to run the container (from your first `p12` file):

```
export CL_KEYSTORE=":name-of-your-file.p12"
export CL_KEYALIAS=":name-of-your-file"
export CL_PASSWORD=":password"
export CL_GLOBAL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_TOKEN_IDENTIFIER=:id_got_of_command_cl_wallet_show_address
export CL_PUBLIC_HTTP_PORT=9000
export CL_P2P_HTTP_PORT=9001
export CL_CLI_HTTP_PORT=9002
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip-global-l0-container
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_L0_PEER_HTTP_HOST=:ip-currency-l0-container
export CL_L0_PEER_HTTP_PORT=9000
export CL_APP_ENV=dev
export CL_COLLATERAL=0
```

- Finally, run the jar:

```
java -jar currency-l1.jar run-initial-validator  --ip :ip_of_current_container
```

- Your should see something like this:

```
23:31:34.892 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App environment: Dev
23:31:34.901 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App version: 2.0.0-a
23:31:38.257 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Self peerId: b1cf4d017eedb3e
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.esotericsoftware.kryo.util.UnsafeUtil (file:/coc
WARNING: Please consider reporting this to the maintainers of com.esotericsoftware.kryo.ut
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
WARNING: All illegal access operations will be denied in a future release
23:31:39.054 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Seedlist disabled.
23:31:49.892 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.895 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=public started a
23:31:49.917 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.918 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=p2p started at /
23:31:49.943 [io-compute-3] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.943 [io-compute-3] INFO  o.t.s.r.MkHttpServer - HTTP Server name=cli started at /
23:31:52.135 [io-compute-6] INFO  o.t.s.i.c.d.N.$anon - Node state changed to=Ready{}
23:31:57.435 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:31:57.635 [io-compute-3] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.598 [io-compute-1] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.658 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:06.858 [io-compute-0] INFO  o.t.d.l.StateChannel - Pulled following global snapshot:
SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdinal{value=11},lastSnapshotHash
```

```
23:32:06.959 [io-compute-0] INFO  o.t.d.l.StateChannel - Snapshot processing result:
DownloadPerformed{reference=SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdina
23:32:07.172 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:07.177 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:12.178 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:12.219 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
```

- That's all for the currency-l1-1 container

Currency L1 - 2

- Here is the instructions to run specifically Currency L1 - 2 container.
- Move the `p12` file to container with the instruction (second `p12` file):

```
docker cp :directory-of-p12-file-2 container-name:file-name.p12
```

- Inside the docker container make sure that your p12 file exists correctly
- It should be at the root level (same level as the tessellation folder)
- Move to tessellation folder:

```
cd tessellation/
```

- Generate the jars

```
sbt currencyL1/assembly
```

- Check the logs to see which version of currency-l1 was published. It should be something like this:

```
```

```
/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-l1-assembly-*.jar
```

- Move this jar to the root folder, like the example below

```
mv codebase/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-l1-assembly-* currency-l1.jar
```

- Outside the container, run this following command to get your docker container IP

```
docker container inspect :container_name | grep -i IPAddress
```

- Outside the container, we need to join our container to the created network, you can do this with the following command (outside the container)

```
docker network connect custom-network-tokens :container_name
```

- You can check now your network and see your container there:

```
docker network inspect custom-network
```

- Fill the environment variables necessary to run the container (from your first `p12` file):

```
export CL_KEYSTORE=":name-of-your-second-file.p12"
export CL_KEYALIAS=":name-of-your-second-file"
export CL_PASSWORD=":password"
export CL_GLOBAL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_TOKEN_IDENTIFIER=:id_got_of_command_cl_wallet_show_address
export CL_PUBLIC_HTTP_PORT=9000
export CL_P2P_HTTP_PORT=9001
export CL_CLI_HTTP_PORT=9002
```

```
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip-global-l0-container
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_L0_PEER_HTTP_HOST=:ip-currency-l0-container
export CL_L0_PEER_HTTP_PORT=9000
export CL_APP_ENV=dev
export CL_COLLATERAL=0
```

- Finally, run the jar:

```
java -jar currency-l1.jar run-validator  --ip :ip_of_current_container
```

- Your should see something like this:

```
23:31:34.892 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App environment: Dev
23:31:34.901 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App version: 2.0.0-a
23:31:38.257 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Self peerId: b1cf4d017eedb3e
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.esotericsoftware.kryo.util.UnsafeUtil (file:/coc
WARNING: Please consider reporting this to the maintainers of com.esotericsoftware.kryo.ut
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
WARNING: All illegal access operations will be denied in a future release
23:31:39.054 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Seedlist disabled.
23:31:49.892 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.895 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=public started a
23:31:49.917 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.918 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=p2p started at /
23:31:49.943 [io-compute-3] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.943 [io-compute-3] INFO  o.t.s.r.MkHttpServer - HTTP Server name=cli started at /
23:31:52.135 [io-compute-6] INFO  o.t.s.i.c.d.N.$anon - Node state changed to=Ready{}
23:31:57.435 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:31:57.635 [io-compute-3] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.598 [io-compute-1] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.658 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:06.858 [io-compute-0] INFO  o.t.d.l.StateChannel - Pulled following global snapshot:
SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdinal{value=11},lastSnapshotHash
23:32:06.959 [io-compute-0] INFO  o.t.d.l.StateChannel - Snapshot processing result:
DownloadPerformed{reference=SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdina
23:32:07.172 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:07.177 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:12.178 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:12.219 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
```

- That's all for the currency-l1-2 container

Currency L1 - 3

- Here is the instructions to run specifically Currency L1 - 2 container.
- Move the `p12` file to container with the instruction (third `p12` file):

```
docker cp :directory-of-p12-file-3 container-name:file-name.p12
```

- Inside the docker container make sure that your p12 file exists correctly
- It should be at the root level (same level as the tessellation folder)
- Move to tessellation folder:

```
cd tessellation/
```

- Generate the jars

```
sbt currencyL1/assembly
```

- Check the logs to see which version of currency-l1 was published. It should be something like this:

```
/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-l1-
assembly-*.jar
```

- Move this jar to the root folder, like the example below

```
mv codebase/tessellation/modules/currency-l1/target/scala-2.13/tessellation-currency-
```

```
l1-assembly-* currency-l1.jar
```

- Outside the container, run this following command to get your docker container IP

```
docker container inspect :container_name | grep -i IPAddress
```

- Outside the container, we need to join our container to the created network, you can do this with the following command (outside the container)

```
docker network connect custom-network-tokens :container_name
```

- You can check now your network and see your container there:

```
docker network inspect custom-network
```

- Fill the environment variables necessary to run the container (from your first `p12` file):

```
export CL_KEYSTORE=":name-of-your-third-file.p12"
export CL_KEYALIAS=":name-of-your-third-file"
export CL_PASSWORD=":password"
export CL_GLOBAL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_PEER_ID=:id_got_of_command_cl_wallet_show_id
export CL_L0_TOKEN_IDENTIFIER=:id_got_of_command_cl_wallet_show_address
export CL_PUBLIC_HTTP_PORT=9000
export CL_P2P_HTTP_PORT=9001
export CL_CLI_HTTP_PORT=9002
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip-global-l0-container
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_L0_PEER_HTTP_HOST=:ip-currency-l0-container
export CL_L0_PEER_HTTP_PORT=9000
export CL_APP_ENV=dev
export CL_COLLATERAL=0
```

- Finally, run the jar:

```
java -jar currency-l1.jar run-validator  --ip :ip_of_current_container
```

- Your should see something like this:

```
23:31:34.892 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App environment: Dev
23:31:34.901 [io-compute-blocker-3] INFO  o.t.s.a.TessellationIOApp - App version: 2.0.0-a
23:31:38.257 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Self peerId: b1cf4d017eedb3e
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.esotericsoftware.kryo.util.UnsafeUtil (file:/coc
WARNING: Please consider reporting this to the maintainers of com.esotericsoftware.kryo.ut
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
WARNING: All illegal access operations will be denied in a future release
23:31:39.054 [io-compute-1] INFO  o.t.s.a.TessellationIOApp - Seedlist disabled.
23:31:49.892 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.895 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=public started a
23:31:49.917 [io-compute-6] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.918 [io-compute-6] INFO  o.t.s.r.MkHttpServer - HTTP Server name=p2p started at /
23:31:49.943 [io-compute-3] INFO  o.h.e.s.EmberServerBuilderCompanionPlatform - Ember-Serv
23:31:49.943 [io-compute-3] INFO  o.t.s.r.MkHttpServer - HTTP Server name=cli started at /
23:31:52.135 [io-compute-6] INFO  o.t.s.i.c.d.N.$anon - Node state changed to=Ready{}
23:31:57.435 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:31:57.635 [io-compute-3] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.598 [io-compute-1] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:02.658 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:06.858 [io-compute-0] INFO  o.t.d.l.StateChannel - Pulled following global snapshot:
SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdinal{value=11},lastSnapshotHash
23:32:06.959 [io-compute-0] INFO  o.t.d.l.StateChannel - Snapshot processing result:
DownloadPerformed{reference=SnapshotReference{height=0,subHeight=11,ordinal=SnapshotOrdina
23:32:07.172 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:07.177 [io-compute-2] DEBUG o.t.d.l.StateChannel - Received block consensus input to
23:32:12.178 [io-compute-0] DEBUG o.t.d.l.d.c.b.Validator - Cannot start own consensus: No
23:32:12.219 [io-compute-0] DEBUG o.t.d.l.StateChannel - Received block consensus input to
```

- That's all for the currency-l1-3 container

Joining Currency L1 containers to build the cluster

- We need to join the 2 and 3 currency L1 container to the first one, to build the cluster.

- For that, we need to open another terminal instance and run

```
docker exec -it :l1-currency-2-container-name /bin/bash
```

- Then we need to call this:

```
curl -v -X POST http://localhost:9002/cluster/join -H \"Content-type:
application/json\" -d '{ \"id\":\":id_got_of_command_cl_wallet_show_id\", \"ip\":
\":ip_of_currency_l1_1_container\", \"p2pPort\": 9001 }'
```

- Repeat the same with the third Currency L1 container

- You now should have the cluster build, if you access the url:
  `http://localhost:9200/cluster/info` you should see the nodes

Next Steps

You should now have a minimal development environment installed and running 🎉

Send your first transaction

Set up the FE Developer Dashboard and send your hello world metagraph transaction.

Edit this page

Customize Rewards Logic

In this guide, we will walk through two different methods of customizing rewards logic within your metagraph.

Understanding Rewards

Rewards are emitted on every timed snapshot of the metagraph and increase the circulating supply of the metagraph token beyond the initial balances defined in genesis.csv. These special transaction types can be used to distribute your currency to fund node operators, or create fixed pools of tokens over time.

By default, no rewards are distributed by a metagraph using the Metagraph Framework which results in a static circulating supply. The rewards customizations described below create inflationary currencies - the rate of which can be controlled by the specific logic introduced. Similarly, a maximum token supply can easily be introduced if desired to prevent unlimited inflation.

The Rewards Function

The rewards function includes contextual information from the prior incremental update, including any data produced. Additionally, this function can include customized code capable of invoking any library function of your choice, allowing you to support truly custom use cases and advanced tokenomics structures. The following examples serve as a foundation for typical use cases, which you can expand upon and tailor to your project's needs.

Before You Start

This guide assumes that you have configured your local environment based on the Quick Start Guide and have at least your `global-l0`, `currency-l0`, `currency-l1` clusters configured.

We will be updating the code within your project in the L0 module. This can be found in:

```
source/project/<project_name>/modules/l0/src/main/Main.scala
```

Please note, the examples below show all logic within a single file to make copy/pasting the code as simple as possible. In a production application you would most likely want to split the code into multiple files.

Examples

These examples show different ways that rewards logic can be customized within your metagraph. The concepts displayed can be used independently or combined for further customization based on the business logic of your project.

Example: Distribute Rewards to Fixed Addresses

Add the following code to your L0 Main.scala file.

```
package com.my.currency.l0

import cats.effect.{Async, IO}
import org.tessellation.BuildInfo
import org.tessellation.currency.dataApplication.BaseDataApplicationL0Service
import org.tessellation.currency.l0.CurrencyL0App
import org.tessellation.currency.schema.currency.{
  CurrencyBlock,
  CurrencyIncrementalSnapshot,
  CurrencySnapshotStateProof,
  CurrencyTransaction
}
import org.tessellation.schema.address.Address
import org.tessellation.schema.balance.Balance
import org.tessellation.schema.cluster.ClusterId
import org.tessellation.schema.transaction.{
  RewardTransaction,
```

```
  TransactionAmount
}
import org.tessellation.sdk.domain.rewards.Rewards
import org.tessellation.sdk.infrastructure.consensus.trigger.ConsensusTrigger
import org.tessellation.security.SecurityProvider
import org.tessellation.security.signature.Signed

import eu.timepit.refined.auto._
import cats.syntax.applicative._

import java.util.UUID
import scala.collection.immutable.{SortedSet, SortedMap}

object RewardsMintForPredefinedAddresses {
  def make[F[_]: Async] =
    new Rewards[
      F,
      CurrencyTransaction,
      CurrencyBlock,
      CurrencySnapshotStateProof,
      CurrencyIncrementalSnapshot
    ] {
      def distribute(
        lastArtifact: Signed[CurrencyIncrementalSnapshot],
        lastBalances: SortedMap[Address, Balance],
        acceptedTransactions: SortedSet[Signed[CurrencyTransaction]],
        trigger: ConsensusTrigger
      ): F[SortedSet[RewardTransaction]] = SortedSet(
        Address("DAG8pkb7EhCkT3yU87B2yPBunSCPnEdmX2Wv24sZ"),
        Address("DAG4o41NzhfX6DyYBTTXu6sJa6awm36abJpv89jB")
      ).map(RewardTransaction(_, TransactionAmount(55_500_0000L))).pure[F]
    }
}

object Main
  extends CurrencyL0App(
    "custom-rewards-l0",
    "custom-rewards L0 node",
    ClusterId(UUID.fromString("517c3a05-9219-471b-a54c-21b7d72f4ae5")),
    version = BuildInfo.version
  ) {

  def dataApplication: Option[BaseDataApplicationL0Service[IO]] = None

  def rewards(implicit sp: SecurityProvider[IO]) = Some(
    RewardsMintForPredefinedAddresses.make[IO]
  )
}
```

The code distributes 5.55 token rewards on each timed snapshot to two hardcoded addresses:

- DAG8pkb7EhCkT3yU87B2yPBunSCPnEdmX2Wv24sZ
- DAG4o41NzhfX6DyYBTTXu6sJa6awm36abJpv89jB

These addresses could represent treasury wallets or manually distributed rewards pools.

Update the number of wallets and amounts to match your use-case.

Rebuild Clusters

Run the following commands to rebuild your clusters with the new code:

```
scripts/hydra destroy
scripts/hydra build --no_cache
```

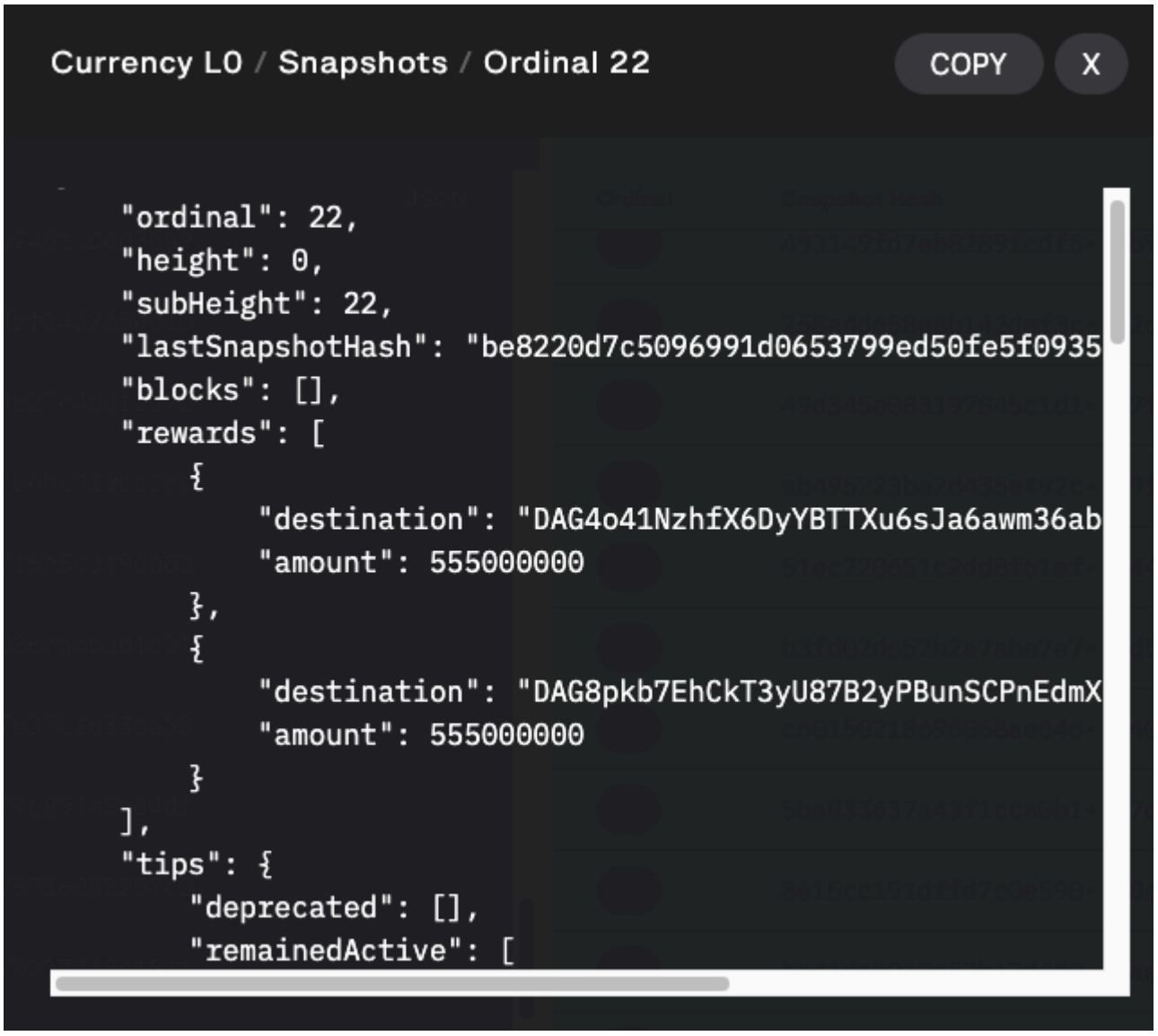
Once built, run hydra start to see your changes take effect.

```
scripts/hydra start-genesis
```

View Changes

Using the Developer Dashboard you should see the balances of the two wallets above increase by 5.5 tokens after each snapshot.

Inspecting the snapshot body, you should also see an array of "rewards" transactions present.



Example: Distribute Rewards to Validator Nodes

Add the following code to your L0 Main.scala file.

```
package com.my.currency.l0

import cats.effect.{Async, IO}
import cats.implicits.{toFoldableOps, toFunctorOps, toTraverseOps}
import org.tessellation.BuildInfo
import org.tessellation.currency.dataApplication.BaseDataApplicationL0Service
import org.tessellation.currency.l0.CurrencyL0App
import org.tessellation.currency.schema.currency.{CurrencyBlock,
CurrencyIncrementalSnapshot, CurrencySnapshotStateProof, CurrencyTransaction}
import org.tessellation.schema.address.Address
import org.tessellation.schema.balance.Balance
import org.tessellation.schema.cluster.ClusterId
import org.tessellation.schema.transaction.{RewardTransaction, TransactionAmount}
import org.tessellation.sdk.domain.rewards.Rewards
import org.tessellation.security.SecurityProvider
```

```
import org.tessellation.security.signature.Signed
import eu.timepit.refined.auto._
import org.tessellation.sdk.infrastructure.consensus.trigger.ConsensusTrigger

import java.util.UUID
import scala.collection.immutable.{SortedMap, SortedSet}

object RewardsMint1ForEachFacilitator {
  def make[F[_]: Async: SecurityProvider] =
    new Rewards[F, CurrencyTransaction, CurrencyBlock, CurrencySnapshotStateProof,
CurrencyIncrementalSnapshot] {
      def distribute(
        lastArtifact: Signed[CurrencyIncrementalSnapshot],
        lastBalances: SortedMap[Address, Balance],
        acceptedTransactions: SortedSet[Signed[CurrencyTransaction]],
        trigger: ConsensusTrigger
      ): F[SortedSet[RewardTransaction]] = {
        val facilitatorsToReward = lastArtifact.proofs.map(_.id)
        val addresses = facilitatorsToReward.toList.traverse(_.toAddress)
        val rewardsTransactions = addresses.map(addresses => {
          val addressesAsList = addresses.map(RewardTransaction(_,
TransactionAmount(1_000_0000L)))
          collection.immutable.SortedSet.empty[RewardTransaction] ++ addressesAsList
        })

        rewardsTransactions
      }
    }
}

object Main
  extends CurrencyL0App(
    "custom-rewards-l0",
    "custom-rewards L0 node",
    ClusterId(UUID.fromString("517c3a05-9219-471b-a54c-21b7d72f4ae5")),
    version = BuildInfo.version
  ) {

  def dataApplication: Option[BaseDataApplicationL0Service[IO]] = None

  def rewards(implicit sp: SecurityProvider[IO]) = Some(
    RewardsMint1ForEachFacilitator.make[IO]
  )
}
```

The code distributes 1 token reward on each timed snapshot to each validator node that participated in the most recent round of consensus.

Rebuild Clusters

Run the following commands to rebuild your clusters with the new code:

```
scripts/hydra destroy
scripts/hydra build --no_cache
```

Once built, run hydra start to see your changes take effect.

```
scripts/hydra start-genesis
```

View Changes

Using the Developer Dashboard you should see the balances of the wallets in each node in your L0 cluster above increase by 1 token after each snapshot.

Inspecting the snapshot body, you should also see an array of "rewards" transactions present.

Example: Distribute Rewards Based on API Data

Add the following code to your L0 Main.scala file.

```
package com.my.currency.l0

import cats.data.NonEmptyList
import cats.effect.{Async, IO}
import cats.implicits.catsSyntaxApplicativeId
import derevo.circe.magnolia.{decoder, encoder}
import derevo.derive
import org.tessellation.BuildInfo
import org.tessellation.currency.dataApplication.BaseDataApplicationL0Service
import org.tessellation.currency.l0.CurrencyL0App
import org.tessellation.currency.schema.currency.{CurrencyBlock,
CurrencyIncrementalSnapshot, CurrencySnapshotStateProof, CurrencyTransaction}
import org.tessellation.schema.address.Address
import org.tessellation.schema.balance.Balance
import org.tessellation.schema.cluster.ClusterId
import org.tessellation.schema.transaction.{RewardTransaction, TransactionAmount}
import org.tessellation.sdk.domain.rewards.Rewards
import org.tessellation.security.SecurityProvider
import org.tessellation.security.signature.Signed
```

```scala
import eu.timepit.refined.numeric.Positive
import eu.timepit.refined.refineV
import eu.timepit.refined.types.numeric.PosLong
import org.tessellation.sdk.infrastructure.consensus.trigger.ConsensusTrigger
import io.circe.parser.decode

import java.util.UUID
import scala.collection.immutable.{SortedMap, SortedSet}

object RewardsMintForEachAddressOnApi {
  private def getRewardAddresses: List[Address] = {

    @derive(decoder, encoder)
    case class AddressTimeEntry(address: Address, date: String)

    try {
      //Using host.docker.internal as host because we will fetch this from a docker container to a API that is on local machine
      //You should replace to your url
      val response = requests.get("http://host.docker.internal:8000/addresses")
      val body = response.text()

      println("API response" + body)

      decode[List[AddressTimeEntry]](body) match {
        case Left(e) => throw e
        case Right(addressTimeEntries) => addressTimeEntries.map(_.address)
      }
    } catch {
      case x: Exception => {
        println(s"Error when fetching API: ${x.getMessage}")
        List[Address]()
      }
    }
  }

  private def getAmountPerWallet(addressCount: Int): PosLong = {
    val totalAmount: Long = 100_000_0000L
    val amountPerWallet: Either[String, PosLong] = refineV[Positive](totalAmount / addressCount)

    amountPerWallet.toOption match {
      case Some(amount) => amount
      case None =>
        println("Error getting amount per wallet")
        PosLong(1)
    }
  }

  def make[F[_] : Async ] =
    new Rewards[F, CurrencyTransaction, CurrencyBlock, CurrencySnapshotStateProof, CurrencyIncrementalSnapshot] {
      def distribute(
```

```
                      lastArtifact: Signed[CurrencyIncrementalSnapshot],
                      lastBalances: SortedMap[Address, Balance],
                      acceptedTransactions: SortedSet[Signed[CurrencyTransaction]],
                      trigger: ConsensusTrigger
                    ): F[SortedSet[RewardTransaction]] = {

        val rewardAddresses = getRewardAddresses
        val foo = NonEmptyList.fromList(rewardAddresses)

        foo match {
          case Some(addresses) =>
            val amountPerWallet = getAmountPerWallet(addresses.size)
            val rewardAddressesAsSortedSet = SortedSet(addresses.toList: _*)

            rewardAddressesAsSortedSet.map(address => {
              val txnAmount = TransactionAmount(amountPerWallet)
              RewardTransaction(address, txnAmount)
            }).pure[F]

          case None =>
            println("Could not find reward addresses")
            val nodes: SortedSet[RewardTransaction] = SortedSet.empty
            nodes.pure[F]
        }
      }
    }
}

object Main
  extends CurrencyL0App(
    "custom-rewards-l0",
    "custom-rewards L0 node",
    ClusterId(UUID.fromString("517c3a05-9219-471b-a54c-21b7d72f4ae5")),
    version = BuildInfo.version
  ) {

  def dataApplication: Option[BaseDataApplicationL0Service[IO]] = None

  def rewards(implicit sp: SecurityProvider[IO]) = Some(
    RewardsMintForEachAddressOnApi.make[IO]
  )
}
```

The code distributes token rewards on each timed snapshot to each address that is returned from a custom API.

On this Repository you can take a better look at the template example and the custom API.

In the repository, the code will distribute the amount of 100 tokens between the number of returned wallets (in this case the maximum of 20 latest wallets)

Rebuild Clusters

Run the following commands to rebuild your clusters with the new code:

```
scripts/hydra destroy
scripts/hydra build --no_cache
```

Once built, run hydra start to see your changes take effect.

```
scripts/hydra start-genesis
```

View Changes

Using the Developer Dashboard you should see the balances of the wallets in each node in your L0 cluster above increase by (100 / :number_of_wallets) tokens after each snapshot.

Inspecting the snapshot body, you should also see an array of "rewards" transactions present.

Custom Data Validation

In this guide, we will walk through a Data Application and a simple example implementation of an IoT use case. Complete code for this guide can be found in the metagraph examples repo on Github. See the Water and Energy Use example.

> **Want more detail?**
>
> Looking for additional detail on Data Application development? More information is available in Data Application.

Before You Start

In order to get started, install dependencies as described in the Quick Start Guide. You will need at least the `global-l0`, `metagraph-l0`, and `metagraph-l1-data` containers enabled in your `euclid.json` file for this guide.

Example euclid.json values

```
  "github_token": "",
  "version": "0.9.1",
  "tessellation_version": "2.2.0",
  "project_name": "custom-project",
  "framework": {
    "name": "currency",
    "modules": [
      "data"
    ],
    "version": "v2.2.0",
    "ref_type": "tag"
  },
  "layers": [
    "global-l0",
    "metagraph-l0",
```

```
    "currency-l1",
    "data-l1"
  ],
```

Installing Templates with Hydra

To initiate a metagraph using a template, we provide several options in our GitHub repository. Follow these steps to utilize a template:

1. List Available Templates: First, determine th`e templates at your disposal by executing the command below:

```
./scripts/hydra install-template --list
```

2. Install a Template: After selecting a template, replace `:repo_name` with your chosen repository's name to install it. For instance:

```
./scripts/hydra install-template :repo_name
```

As a practical example, if you wish to install the `water-and-energy-usage` template, your command would look like this:

```
./scripts/hydra install-template water-and-energy-usage
```

This process will set up a metagraph based on the selected template.

Within your Euclid modules directory (source/project/water-and-energy-usage/modules) you will see three module directories: l0 (metagraph l0), l1 (currency l1), and data_l1 (data l1). Each module has a Main.scala file that defines the application that will run at each corresponding layer.

```
```

```
- source
  - project
    - water-and-energy-usage
      - modules
        - l0
        - l1
        - data_1
        - shared_data
      - project
```

Send Data

Edit the `send_data_transaction.js` script and fill in the `globalL0Url`, `metagraphL1DataUrl`, and `walletPrivateKey` variables. The private key can be generated with the `dag4.keystore.generatePrivateKey()` method if you don't already have one.

Once the variables are updated, save the file. You can now run `node send_data_transaction.js` to send data to the `/data` endpoint.

Check State Updates

Using the custom endpoint created in the data_l1 Main.scala `routes` method, we can check the metagraph state as updates are sent.

Using your browser, navigate to `<your L1 base url>/data-application/addresses` to see the complete state including all devices that have sent data. You can also check the state of an individual device using the `<your L1 base url>/data-application/addresses/:address` endpoint.

You should see a response like this:


```
{
    "DAG4bQGdnDJ5okVdsdtvJzBwQoPGjLNzN7HC1CBV": {
        "energy": {
            "usage": 7,
            "timestamp": 1689441998946
        },
```

```
            "water": {
                "usage": 7,
                "timestamp": 1689441998946
            }
        }
    }
```


Next Steps

This brief guide demonstrates the ability to update and view on-chain state based on the Metagraph Framework's Data Application layer. Detailed information about the framework methods used can be found in the example README file and in comments throughout the code. Also see additional break downs of the application lifecycle methods in the Data API section.

Working with p12 files

Generating p12 files

This guide will walk you through the process of creating your own custom p12 files. We will generate three files to match the original Euclid Development Environment project's configuration.

> **Caution**
>
> If using a Euclid Development Environment project, you must update your configuration to use your own custom p12 files. Projects submitted with the default p12 files that come with the project will be rejected.

Step 1: Download `cl-keytool.jar` Executable

Download the `cl-keytool.jar` executable. This is included as an asset with each release of Tessellation.

Step 2: Set Up Your Environment Variables

Modify the following variables with your custom details and export them to your environment:

```
export CL_KEYSTORE=":your_custom_file_name.p12"
export CL_KEYALIAS=":your_custom_file_alias"
export CL_PASSWORD=":your_custom_file_password"
```

Replace `:your_custom_file_name.p12`, `:your_custom_file_alias`, and `:your_custom_file_password` with your specific file name, alias, and password, respectively.

Step 3: Generate Your Custom .p12 File

Execute the following command to generate your custom .p12 file:

```
java -jar cl-keytool.jar generate
```

This will create a .p12 file in the directory from which the command was executed.

Step 4: Repeat the Process

Repeat steps 2 and 3 two more times to create a total of three custom p12 files. Remember to change the file name each time to avoid overwriting any existing files.

Finding Your Node IDs

Your node ID is the public key of your wallet which will be stored as a p12 file.

> **Caution**
>
> If using a Euclid Development Environment project, you must update your configuration to use your own custom p12 files. Projects submitted with the default p12 files that come with the project will be rejected.
>
> How to generate p12 files

Download the `cl-wallet.jar` executable. This is distributed as an asset with each release of Tessellation.

Editing the details of the following variables and export to your environment.

```
export CL_KEYSTORE=":your_file_name.p12"
export CL_KEYALIAS=":your_file_alias"
export CL_PASSWORD=":your_file_password"
```

Then you can run the following to get your node ID:

```
java -jar cl-wallet.jar show-id
```

Edit this page

Snapshot Fees

The Hypergraph charges fees for validating and storing metagraph snapshots, ensuring the network's continued functionality. These snapshot fees, along with node collateral requirements, are the only expenses metagraphs must pay in order to interface with the Hypergraph. This fee structure provides metagraphs with significant flexibility, enabling them to decide their own fee structures and data inclusion policies.

Metagraphs have the autonomy to choose whether to charge their end users directly, impose fees for specific data types, or even operate without user fees. They can also control which data is included in their snapshots, managing costs and determining the privacy level of their network.

Fees are calculated based on the size and computational cost of processing snapshots. Currently, all snapshots have a computational cost of 1, which means that snapshot size is the only active factor in determining snapshot fee cost. For detailed information on network fees, refer to the Network Fees Litepaper.

Owner and Staking Wallets

The Global L0 deducts snapshot fees from an "owner wallet," which is designated by a majority of metagraph validators and registered with the gL0. An additional wallet, known as the "staking wallet," can also be designated to reduce fees based on its balance at the time a snapshot is processed. These two wallets can be the same, but the addresses used for either the owner or staking wallets must be globally unique on the Hypergraph.

The owner and staking wallets are designated by signing a message to prove ownership of each wallet and creating a "Currency Message" for the metagraph. This Currency Message must be signed by a majority of the L0 nodes of the metagraph, then included in a metagraph

snapshot to be sent to the gL0 for registration and inclusion in a global snapshot. Owner and staking wallets can be changed at any time using the same process.

> **Assigning an owner wallet is required**
>
> Starting in Tessellation v2.7.0, the Hypergraph will reject snapshots sent by metagraphs that do not designate an owner wallet or if the designated wallet does not have sufficient funds to cover the cost of the current snapshot's fees. For this reason, assigning an owner wallet is required. Assigning a staking wallet is optional.

Assigning Metagraph Wallets with Euclid

Euclid simplifies the process of assigning both the owner and staking wallets with a few simple Hydra commands (v0.11.0 or later).

> **Local builds**
>
> Snapshot fees are turned off by default for local builds. Owner and staking details only need to be configured when deploying to MainNet or a public testnet (IntegrationNet, etc.).

First, update the `snapshot_fees` key in euclid.json with the name, alias, and password of the p12 file for each of the wallets. The p12 files should be stored in the `source/p12-files` directory and should have a unique name.

For example (replace with your own values):

```
{
  "snapshot_fees": {
    "owner": {
        "name": "metagraph_owner.p12",
        "alias": "metagraph_owner",
        "password": "pass1234"
    },
    "staking": {
        "name": "metagraph_staking.p12",
        "alias": "metagraph_staking",
```

```
            "password": "pass1235"
        }
      }
    }
```

Next, run `hydra remote-deploy` , this will deploy your owner and staking p12 files to the remote nodes.

If running your remote metagraph from genesis, no special configuration is required - owner and (optionally) staking wallet configuration will be automatically set for you if provided in euclid.json. Simply run `hydra remote-start` or `hydra remote-start --force_genesis` to start your metagraph from genesis.

If running an existing metagraph as rollback, run

```
hydra remote-start --force_owner_message --force_staking_message
```

to set or overwrite the fee wallet configuration for the metagraph.

The `--force_staking_message` parameter is optional and can be removed if not using a staking wallet.

To check that your configuration has been successfully updated, run

```
hydra remote-snapshot-fee-config
```

You should see an output like this:

```
OWNER
Owner Address: DAG3Z6oMiqXyi4SKEU4u4fwNiYAMYFyPwR3ttTSd
Owner Parent Ordinal: 0

STAKING
Staking Address: DAG5yuTbZP5Jq55yzdDkAisULFhkhPrW7XRqNY1D
Staking Parent Ordinal: 0
```

If both the Owner and Staking addresses are set then the addresses are properly assigned and the Hypergraph will process snapshot fees based on that configuration.

Overview

This tutorial will guide you through the process of deploying your Euclid metagraph project to a cloud provider and connecting to IntegrationNet or MainNet. We focus on AWS specifically but the basic principles would apply to any cloud provider.

Deploying a Metagraph with Euclid

Utilize Euclid for deploying metagraphs efficiently. Initiate deployment to your remote hosts, including all necessary files and dependencies, with the following command:

```
./scripts/hydra remote-deploy
```

To start your nodes, execute:

```
./scripts/hydra remote-start
```

For comprehensive guidance on utilizing these commands, consult the README file in the Euclid repository.

Additionally, we offer a demonstration video showcasing this functionality, available here.

Architecture

There are many kinds of potential deployment architectures possible for production deployments depending on project scaling needs. Here, we will focus on a deployment

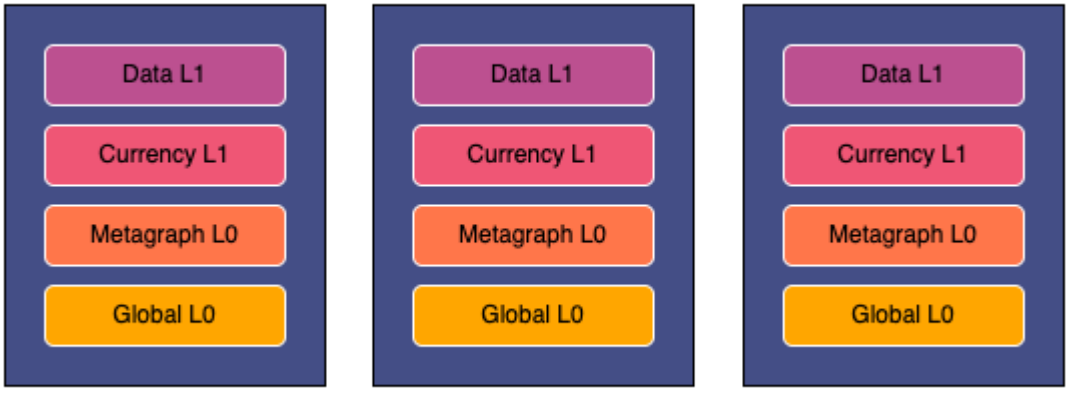
strategy that uses a minimal set of infrastructure components to simplify deployment and reduce cloud costs. For most projects, this offers a good starting point that can be expanded on later to meet specific project needs.

We will be deploying a Metagraph Framework metagraph using a Data Application. This type of metagraph consists of 4 layers in total:

- **Global L0:** Hypergraph node on IntegrationNet or MainNet
- **Metagraph L0:** Metagraph consensus layer
- **Currency L1:** Metagraph layer for accepting and validating token transactions
- **Data L1:** Metagraph layer for accepting and validating data updates

In this guide, we will deploy all 4 layers to each of 3 EC2 instances. In other words, we will only use 3 servers but each server will act as a node on each of the 4 layers. This allows all layers to have the minimum cluster size to reach consensus (3), while also being conscious of infrastructure costs by combining each layer onto the same EC2 instances. Each layer will communicate over custom ports which we will configure as part of this process.

Deployed Architecture:



Requirements

- AWS Account
- A metagraph project built and tested locally in Euclid
- At least 3 `p12` files. Refer to this guide on how to generate p12 files.

- Ensure that the ID of all your `p12` files is on the appropriate network seedlist (IntegrationNet or MainNet) otherwise, you won't be able to connect to the network. Check the seedlist to verify your IDs are included.

Guide

This guide will walk you through a series of steps to manually configure your nodes via the AWS console. We will configure AWS, build all code on a base instance that we will then convert to an AWS AMI to be used as a template for creating the rest of the nodes. This allows us to build once, and then duplicate it to all of the EC2 instances. Then we will configure each of the nodes with their own P12 file and other details specific to each node.

We will walk through the following steps:

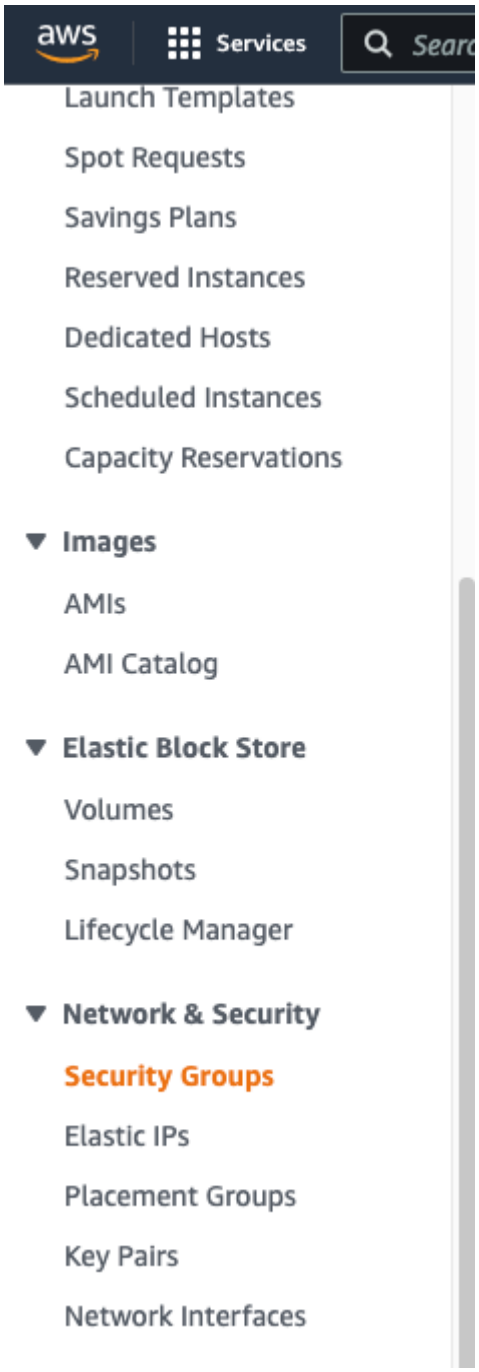
- Configure security groups: Create a security group for the nodes and open the proper network ports for communication.
- Setup SSH keys: Create SSH keys to securely connect to the nodes.
- Create a base instance: Build a server image as an AWS AMI to be reused for each of the nodes.
- Configure the base instance: Add all dependencies and upload metagraph project files to the base instance.
- Generate AMI: Convert the base instance into a reusable AMI.
- Generate EC2 Instances from AMI: Using the AMI created in previous steps as a template, generate all 3 EC2 instances.
- Configure Layers and Join: Configure each of the 4 layers and join to the network.

Security groups

Security groups act as virtual firewalls that control inbound and outbound traffic to your instances. Our 3 nodes will need to open up connection ports for SSH access, and for each of the 4 network layers to communicate over.

Create a Security Group

First, navigate to the `Security Groups` section in the Amazon EC2 console.



Click on `Create Security Group`

Create a new security group and provide a name, for example `MetagraphSecurityGroup`.

Add Inbound Rules

Inbound rules define which ports accept inbound connections on your node. We will need to open up ports for SSH access and for each of the metagraph layers.

Click **`Add Rule`** under the **`Inbound Rules`** section and add the following rules:

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	0.0.0.0/0	SSH access
Custom TCP	TCP	9000-9002	0.0.0.0/0	gL0 layer
Custom TCP	TCP	9100-9102	0.0.0.0/0	mL0 layer
Custom TCP	TCP	9200-9202	0.0.0.0/0	cL1 layer
Custom TCP	TCP	9300-9302	0.0.0.0/0	dL1 layer

Key pairs

Key pairs are a crucial part of securing your instances. They consist of a public key that AWS stores and a private key file that you store. The private key file is used to SSH into your instances securely.

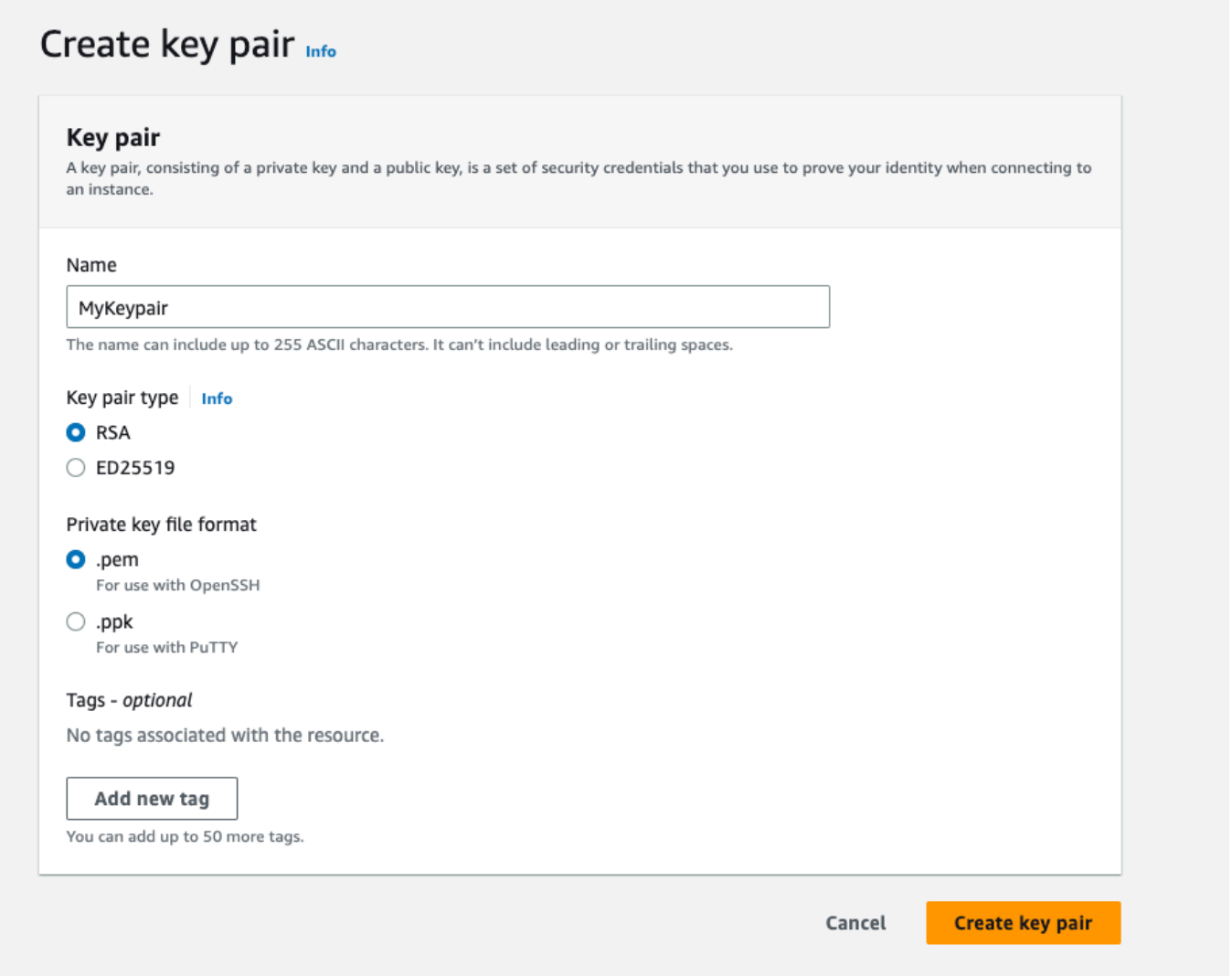
Use the following steps to create a keypair

Navigate to the `Key pairs` page on the Amazon EC2 console.

Click on `Create key pair`

Provide a unique name for your key pair, such as: `MetagraphKeyPair`

Your screen should now look similar to this:



After you click **`Create key pair`**, a new key pair will be generated, and your browser will automatically download a file that contains your private key.

> **Important**
>
> Safeguard this file as it will be necessary for SSH access to your instances. Do not share this file or expose it publicly as it could compromise the security of your instances.

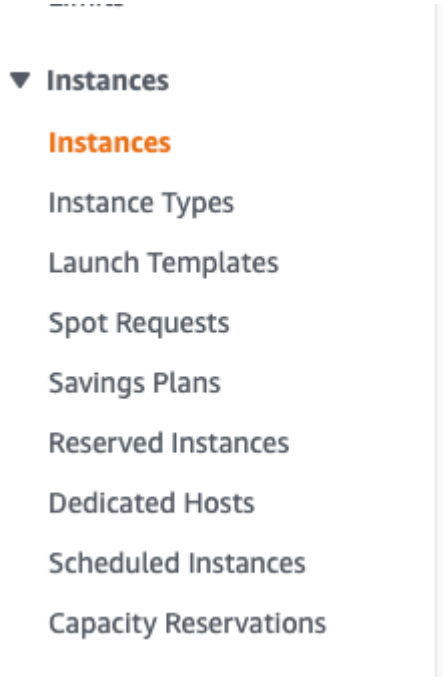
Store your keypair on your local machine in a secure location. You will need it to connect to your EC2 instances.

Generating base instance

In this section, we will create a single EC2 instance that we will use as a template for the other two EC2 instances. This allows us to perform these tasks once and then have the output replicated to all the instances.

Create a Base Instance

Navigate to the **`Instances`** section on the Amazon EC2 console.

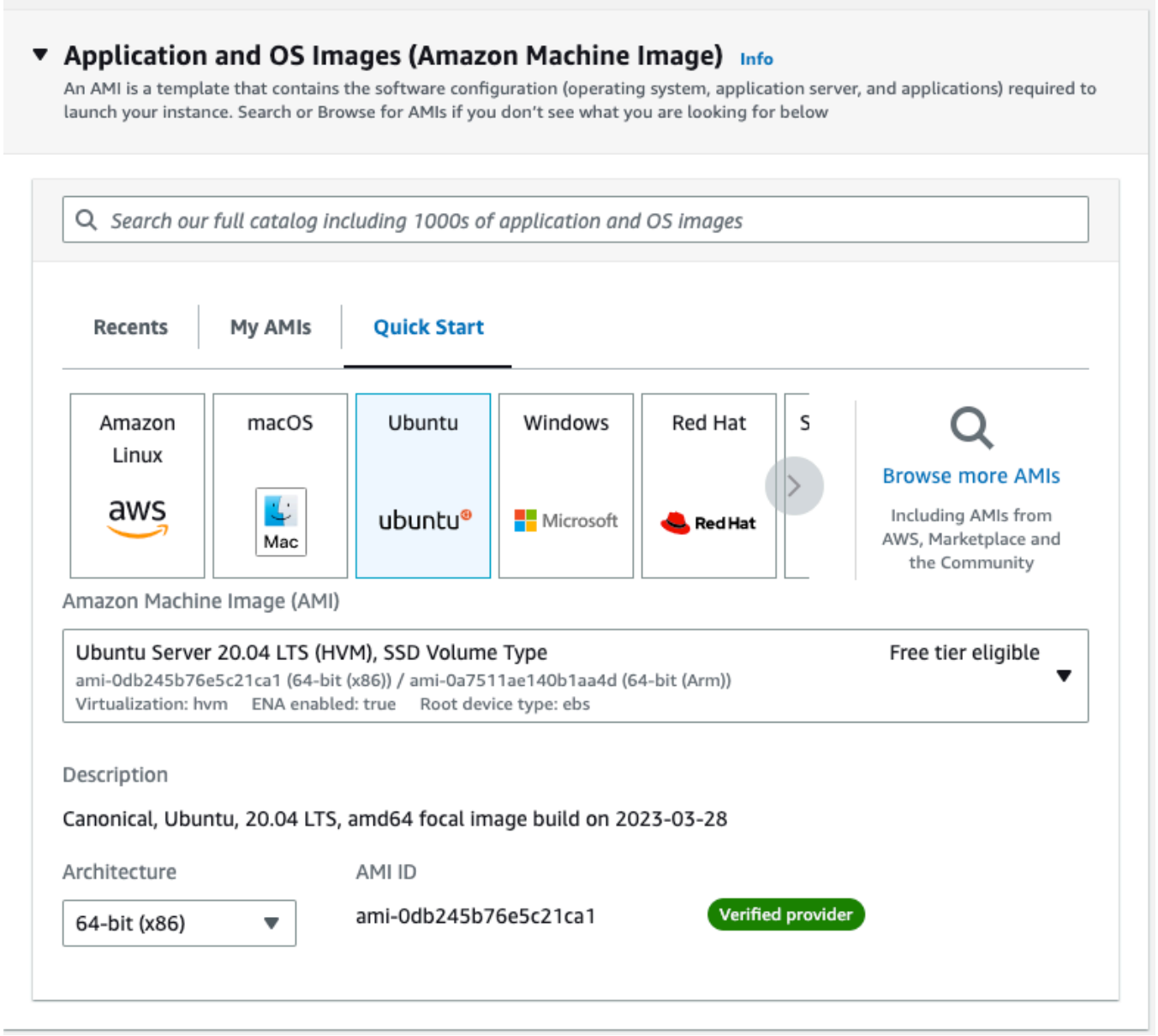


Click on **`Launch Instances`**.

Assign a name to your instance. For this guide, we will call it **`Metagraph base image`**.

Choose an AMI

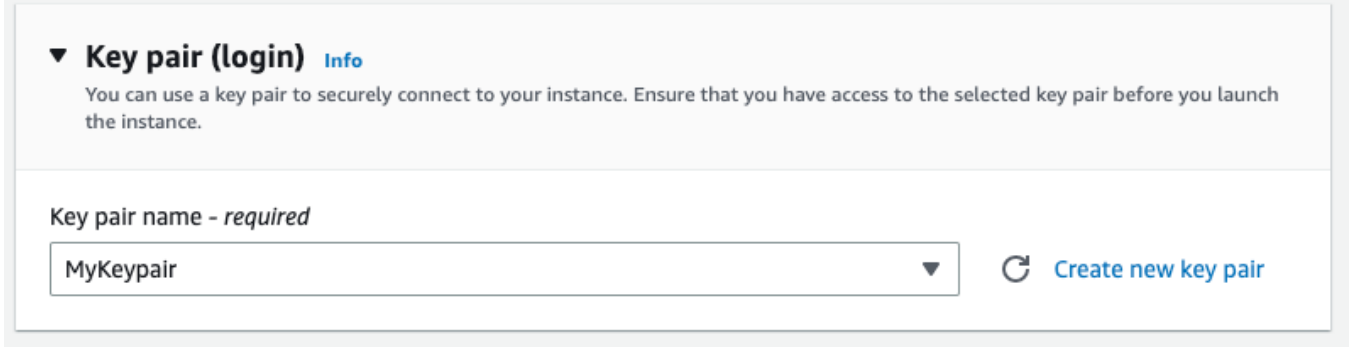
In the **`Choose an Amazon Machine Image (AMI)`** section, select `Ubuntu` and then `Ubuntu server 20.04`. You should keep `64-bit (x86)`.



For the instance type, choose a model with `4 vCPUs` and `16 GiB memory`. In this case, we'll use the `t2.xlarge` instance type.

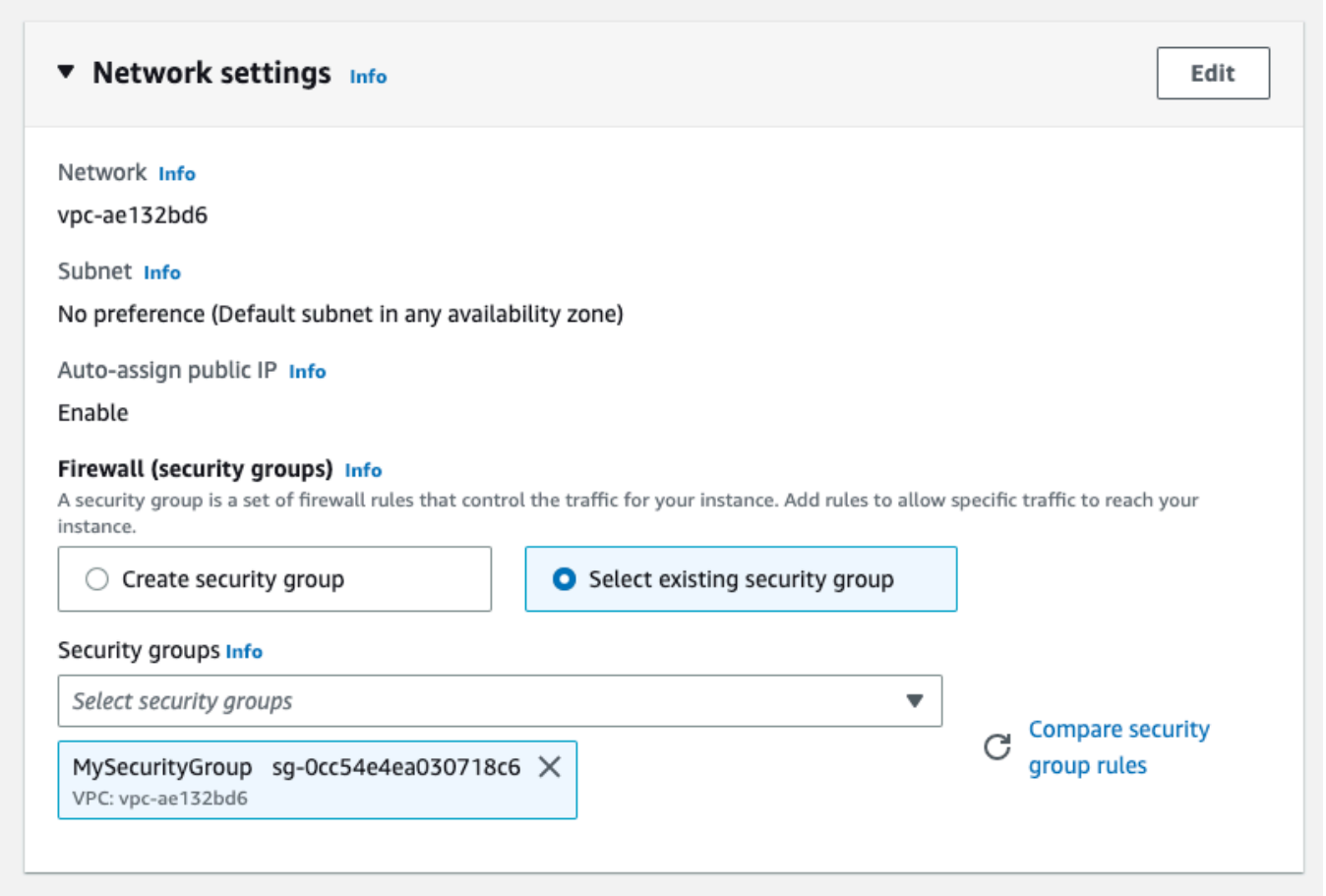
Select a Key Pair

In the `Configure Instance Details` step, select the key pair you created earlier in the `Key pair name` field.



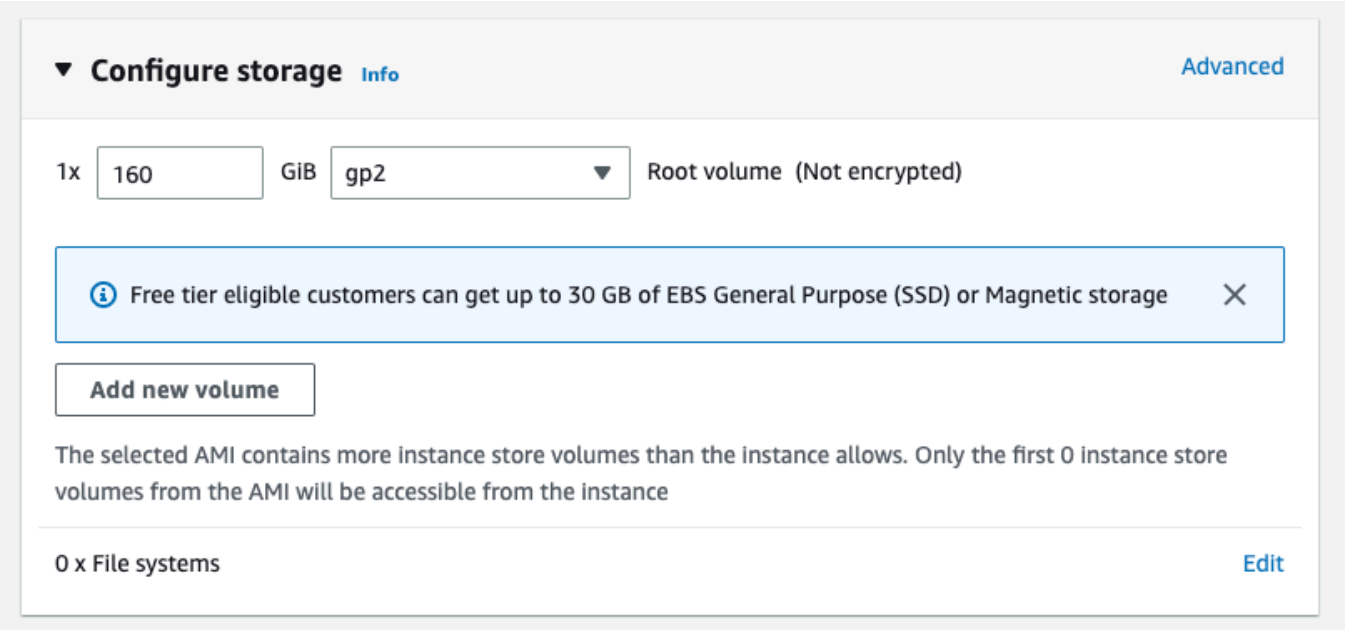
Select Security Group

In the `Network settings` section, you select the security group you created earlier.



Configure Storage

In the `Configure storage` section, you specify the amount of storage for the instance. For this tutorial, we'll set it to **`160 GiB`**.



Launch Instance

Finally, press `Launch instance`. Your base instance should now be running.

You can check the status of your instance in the **`Instances`** section of the Amazon EC2 console.

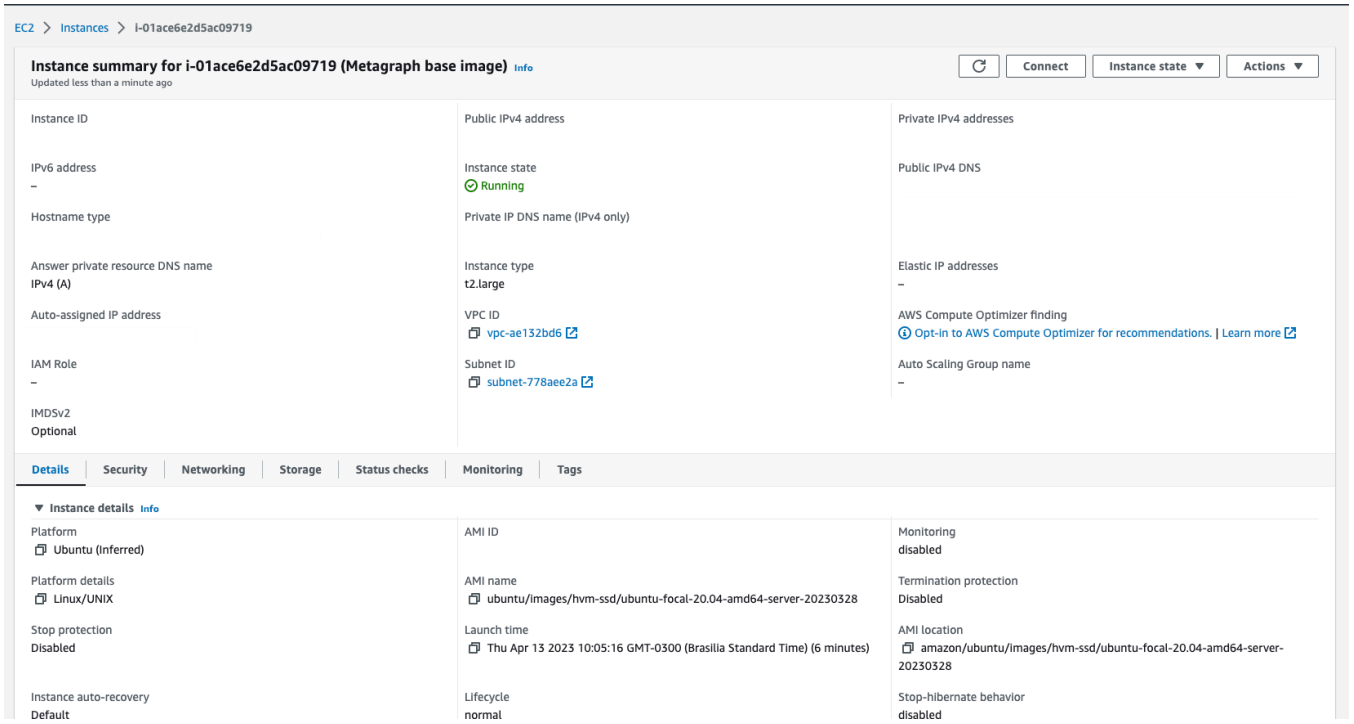
Edit this page

Configuring base instance

Connect to the instance

From your `Instances` page, click on your instance.

Then you should see something like this:



Click on the `Connect` button at the top of the page.

There are different ways to access the instance. In this example, we will connect using `ssh` using the file downloaded in the Key pairs step.

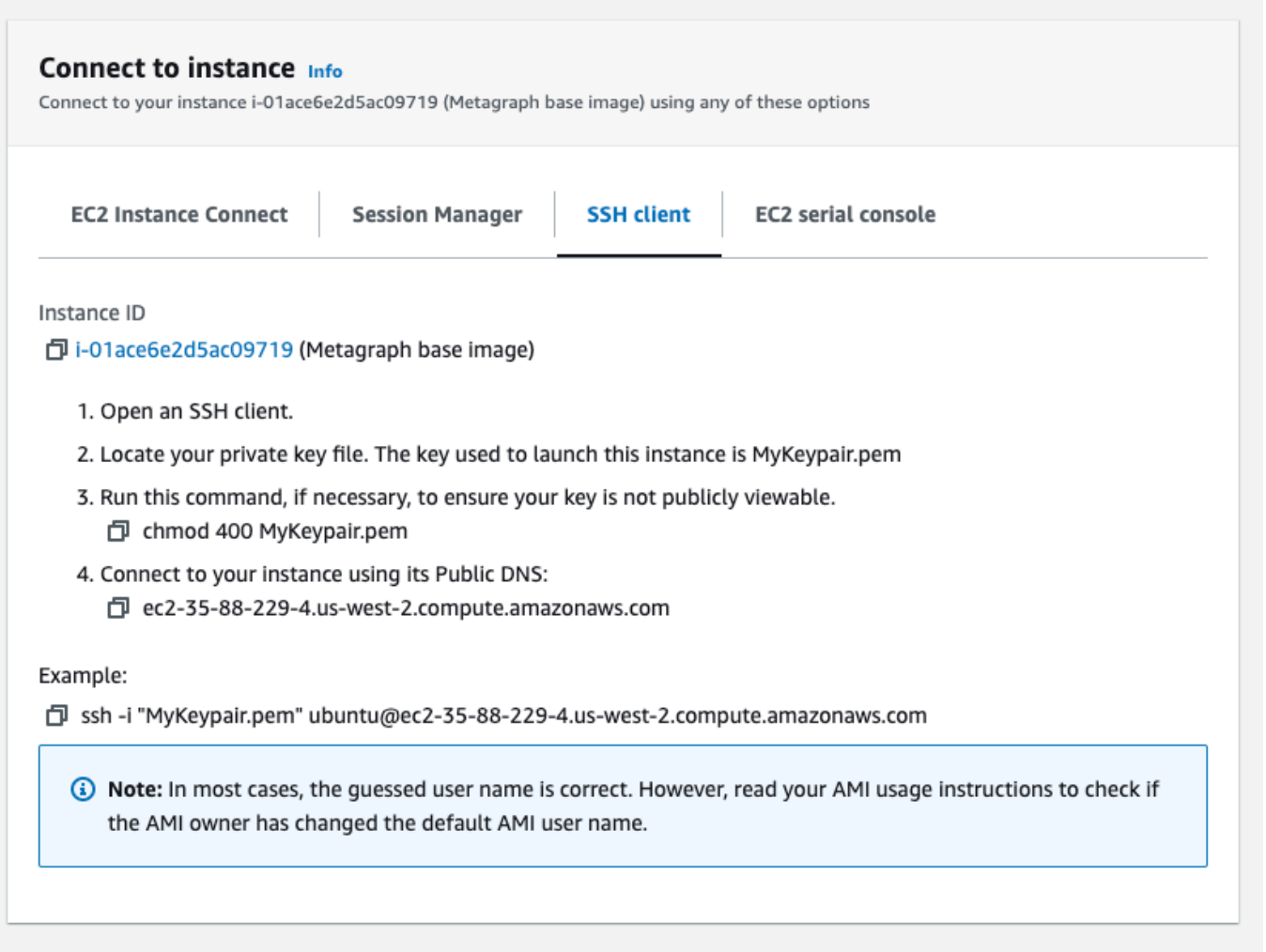
Grant privileges to the SSH key

```
chmod 400 MyKeypair.pem
```

Use the `ssh` command to connect to your instance

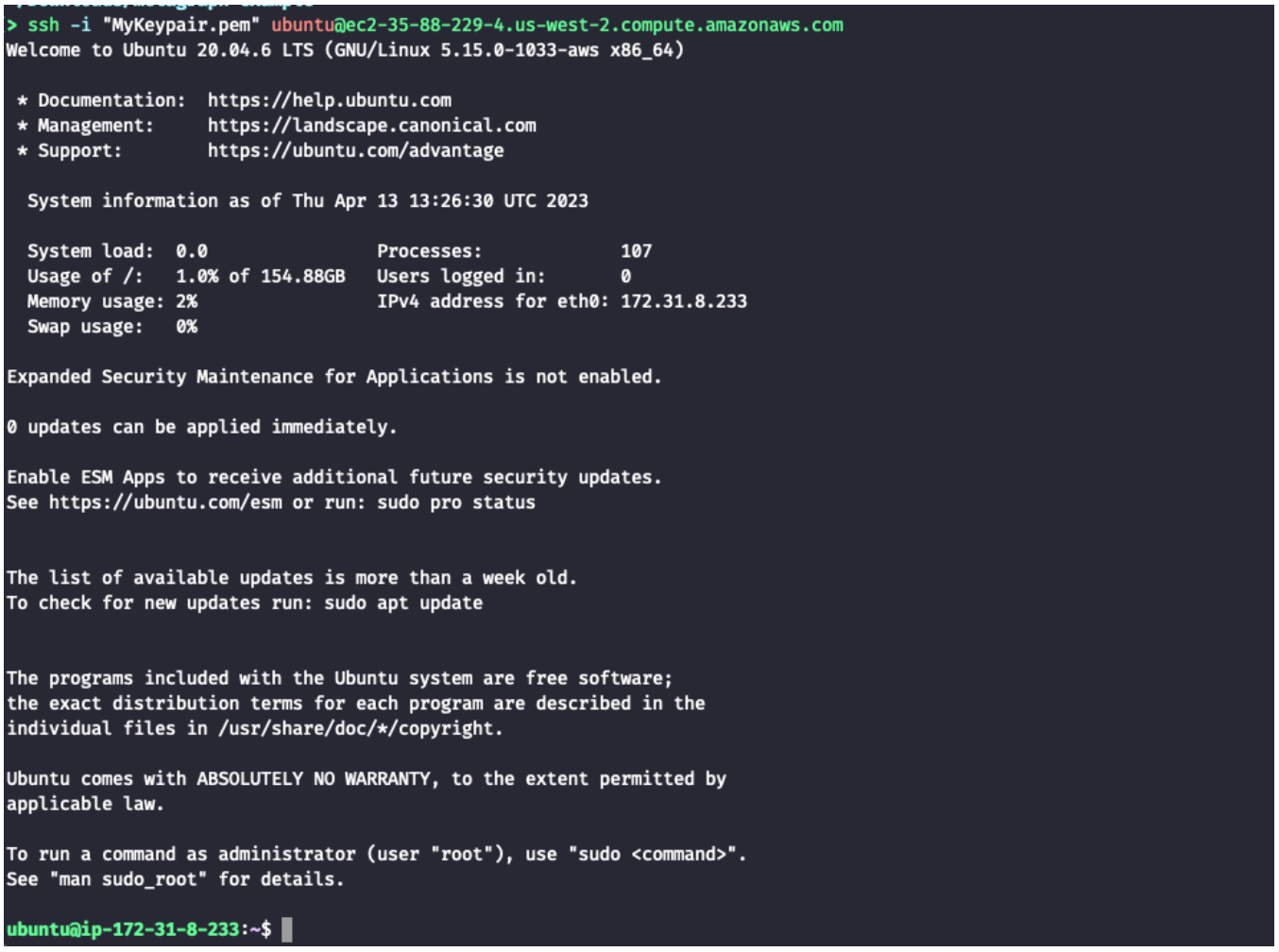
```
ssh -i "MyKeypair.pem" ubuntu@your_instance.aws-region.compute.amazonaws.com
```

The name/IP of the instance will be different, but you can get the instructions on how to connect via ssh in the **`Connect to your instance`** section of the EC2 Console.



If asked to confirm the fingerprint of the instance, type **`yes`**.

Once connected, you should see a screen similar to this:



Base instance setup

Now, you can begin setting up your instance.

Create base directory

Create a directory named `code` and navigate into it. This will be the base directory that we will work out of.

```
mkdir code
cd code/
```

Create layer directories

Create the following directories: `global-l0`, `metagraph-l0`, `currency-l1`, and `data-l1`. These will be the root directories for each of the layers.

```
mkdir global-l0
mkdir metagraph-l0
mkdir currency-l1
mkdir data-l1
```

Add Tessellation utilities to each directory

Replace "v2.2.0" with the latest version of Tessellation found here:

https://github.com/Constellation-Labs/tessellation/releases

```
cd global-l0

wget https://github.com/Constellation-Labs/tessellation/releases/download/v2.2.0/cl-node.jar
wget https://github.com/Constellation-Labs/tessellation/releases/download/v2.2.0/cl-wallet.jar
wget https://github.com/Constellation-Labs/tessellation/releases/download/v2.2.0/cl-keytool.jar

cp cl-wallet.jar metagraph-l0/cl-wallet.jar
cp cl-wallet.jar currency-l1/cl-wallet.jar
cp cl-wallet.jar data-l1/cl-wallet.jar

cp cl-keytool.jar metagraph-l0/cl-keytool.jar
cp cl-keytool.jar currency-l1/cl-keytool.jar
cp cl-keytool.jar data-l1/cl-keytool.jar
```

Install the necessary dependencies:

```
sudo apt-get update
sudo apt install openjdk-11-jdk -y
sudo apt-get install curl -y
sudo apt-get install wget -y
sudo apt-get install gnupg -y

sudo echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/sources.list.d/sbt.list
sudo echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.list.d/sbt_old.list
sudo curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
```

```
sudo apt-get update

sudo apt-get install sbt -y
```

Generate Metagraph JAR Files

For each of the metagraph layers, code from your project must be compiled into executable jar files. During local development with Euclid these files are compiled for you and stored within the `infra` directory of your project code. You can move these locally tested JAR files directly onto your base instance for deployment (recommended for this tutorial).

After ensuring that your project is ready for deployment, navigate to the following directory in your local Euclid codebase: `infra -> docker -> shared -> jars`

Within this directory, you will find the following JARs:

```
- `metagraph-l0.jar`
- `metagraph-l1-currency.jar`
- `metagraph-l1-data.jar`
```

Use `scp` to copy the files to your metagraph layer directories:

```
scp -i "MyKeypair.pem" your_jar_directory/metagraph-l0.jar ubuntu@ec2-your-ip.your-region.compute.amazonaws.com:code/metagraph-l0/metagraph-l0.jar
scp -i "MyKeypair.pem" your_jar_directory/metagraph-l1-currency.jar ubuntu@ec2-your-ip.your-region.compute.amazonaws.com:code/currency-l1/currency-l1.jar
scp -i "MyKeypair.pem" your_jar_directory/metagraph-l1-data.jar ubuntu@ec2-your-ip.your-region.compute.amazonaws.com:code/data-l1/data-l1.jar
```

Alternative Option

Alternatively, you could choose to generate the JARs on the base instance itself. If you choose that route, you can follow the steps in the following guide.

Generating JARs on Base Instance

Setting up the Genesis File

The genesis file is a configuration file that sets initial token balances on your metagraph at launch, or genesis. This allows your project to start with any configuration of wallet balances you choose, which will only later be updated through token transactions and rewards distributions.

Genesis file

If you already have your genesis file used for testing on Euclid, you can upload the file here.

```
scp -i "MyKeypair.pem" your_genesis_file.csv ubuntu@ec2-your-ip.your-region.compute.amazonaws.com:code/metagraph-l0/genesis.csv
```

Generating metagraphID

Before connecting your metagraph to the network, we will generate its' ID and save the output locally. This ID is a unique key used by the Global L0 store state about your metagraph.

> **Info**
>
> When deploying to MainNet, your metagraphID must be added to the metagraph seedlist before you will be able to connect. Provide the metagraphID generated below to the Constellation team to be added to the seedlist.
>
> IntegrationNet does not have a metagraph seedlist so you can connect easily and regenerate your metagraphID if needed during testing.

Generate your metagraphID

```
cd ~/code/metagraph-l0

export CL_KEYSTORE=test.p12
export CL_KEYALIAS=test
export CL_PASSWORD=test
export CL_PUBLIC_HTTP_PORT=9100
export CL_P2P_HTTP_PORT=9101
export CL_CLI_HTTP_PORT=9102
export CL_GLOBAL_L0_PEER_HTTP_HOST=localhost
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export
CL_GLOBAL_L0_PEER_ID=e2f4496e5872682d7a55aa06e507a58e96b5d48a5286bfdff7ed780fa464d9e789b27
export CL_APP_ENV=integrationnet

java -jar cl-keytool.jar generate

nohup java -jar metagraph-l0.jar create-genesis genesis.csv > metagraph-l0.log 2>&1 &

rm test.p12
```

View Genesis Output

You will find the following files in your directory:

- `genesis.snapshot`
- `genesis.address`

The `genesis.address` file contains your metagraphID, which should resemble a DAG address: `DAG...`. The `genesis.snapshot` file contains snapshot zero of your metagraph which will be used when connecting to the network for the first time.

Your base instance is now fully configured

The following sections will cover creating each EC2 instance from this base instance and configuring each individually. You can skip ahead to the Generating AMI section.

Generating metagraph JARs on the base instance

> **Warning**
>
> This guide describes generating metagraph JAR files directly on a node instance. For most use cases, it is recommended to build JAR files using Euclid instead, or to modify this process as part of an automated CI deployment.

This guide will give you the step by step on how to generate the metagraph JARs directly on your base instance as an alternative to uploading files tested in Euclid.

Setting up the Tessellation repository

Clone the Tessellation repository and checkout the integrationnet node version. You can find the integrationnet node version using the `/node/info` endpoint in any existing node of the network.


```
// 20230413104057
// http://54.71.12.221:8000/node/info

{
  "state": "Ready",
  "session": 1681321171161,
  "clusterSession": 1681314958617,
  "version": "2.0.0-alpha.3",
  "host": "54.71.12.221",
  "publicPort": 8000,
  "p2pPort": 8001,
  "id": "f810cbfd727785e3ce83d984e8086bb259705e80fb2d5525419a21d8dc475d1b31f0d55241ca6cbc0ad43203df292417c88dc30581cbb1c3b2dd6e8922136653"
}
```


```
git clone https://github.com/Constellation-Labs/tessellation.git
cd tessellation
git checkout v2.2.0
```

Warning

Make sure you're using the latest version of Tessellation. You can find the most recent release in **here**.

- Create a GitHub token to build and publish the JARs.
- Instructions on creating an access token can be found in the GitHub documentation. The token should only have the `read:packages` scope.
- Ensure you have a clean `.m2` directory:

```
rm -r ~/.m2
```

- Publish locally (on the `.m2` directory) the JARs that will be used for the Metagraph instances:

```
export GITHUB_TOKEN=:your_token
sbt clean
```

```
sbt shared/publishM2 kernel/publishM2 keytool/publishM2 sdk/publishM2 dagL1/publishM2
currencyL0/publishM2 currencyL1/publishM2
```

Info

This part of the process can take some time.

- Return to the root directory:

```
cd ..
```

Metagraph project

There are two options for the Metagraph project. You can either create the project from scratch on the instance or upload your existing project to the instance.

Creating project from scratch

- Install the coursier and giter8 packages:

```
curl -fL https://github.com/coursier/coursier/releases/latest/download/cs-x86_64-pc-linux.gz | gzip -d > cs && chmod +x cs && ./cs set
```

```
source ~/.profile
./cs install giter8
```

- Now create the Metagraph project. We will use the `gitter` lib for that
- **We should provide the same version as the Tessellation checkout above**

```
g8 Constellation-Labs/currency --tag v2.2.0 --name="my-project" --tessellation_version="2.0.0" --include_data_l1="yes"
```

> **Info**
>
> The command above includes the data-l1 layer with `--include_data_l1="yes"`. You can omit this parameter if your implementation does not require this layer.

Uploading project

- You can upload your project tested on Euclid to the `base-instance`
- To upload your project, you can use `scp` with the `-r` flag:

```
scp -i "MyKeypair.pem" -r your_project_directory ubuntu@ec2-your-ip.your-region.compute.amazonaws.com:code
```

Compiling JARs

- Compile the metagraph JARs: `metagraph-l0`, `metagraph-l1`, and `data-l1`

```
cd your_project_directory/
sbt clean currencyL0/assembly currencyL1/assembly dataL1/assembly
cd ..
```

- Move and rename the JARs to the metagraph directories:

```
cd ..
mv your_project_directory/modules/l0/target/scala-2.13/my-project-currency-l0-assembly-0.1.0-SNAPSHOT.jar metagraph-l0/metagraph-l0.jar
mv your_project_directory/modules/l1/target/scala-2.13/my-project-currency-l1-assembly-0.1.0-SNAPSHOT.jar currency-l1/currency-l1.jar
mv your_project_directory/modules/data_l1/target/scala-2.13/my-project-data-l1-assembly-0.1.0-SNAPSHOT.jar data-l1/data-l1.jar
```

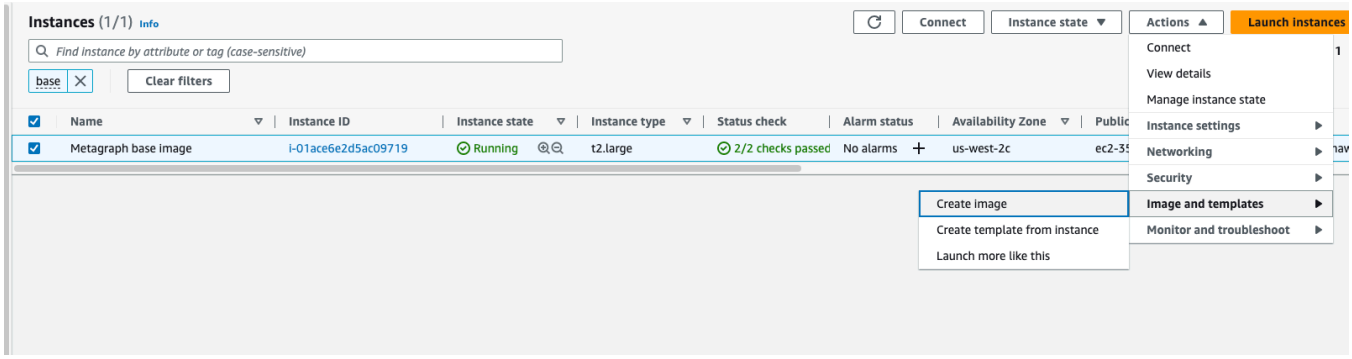
> **Info**
>
> If you are not using the data-l1 layer, feel free to exclude `dataL1/assembly` and the subsequent `mv my-project/modules/data_l1/target/scala-2.13/my-project-data-l1-assembly-0.1.0-SNAPSHOT.jar data-l1.jar` steps.

Generating AMI (Image) from Base Instance

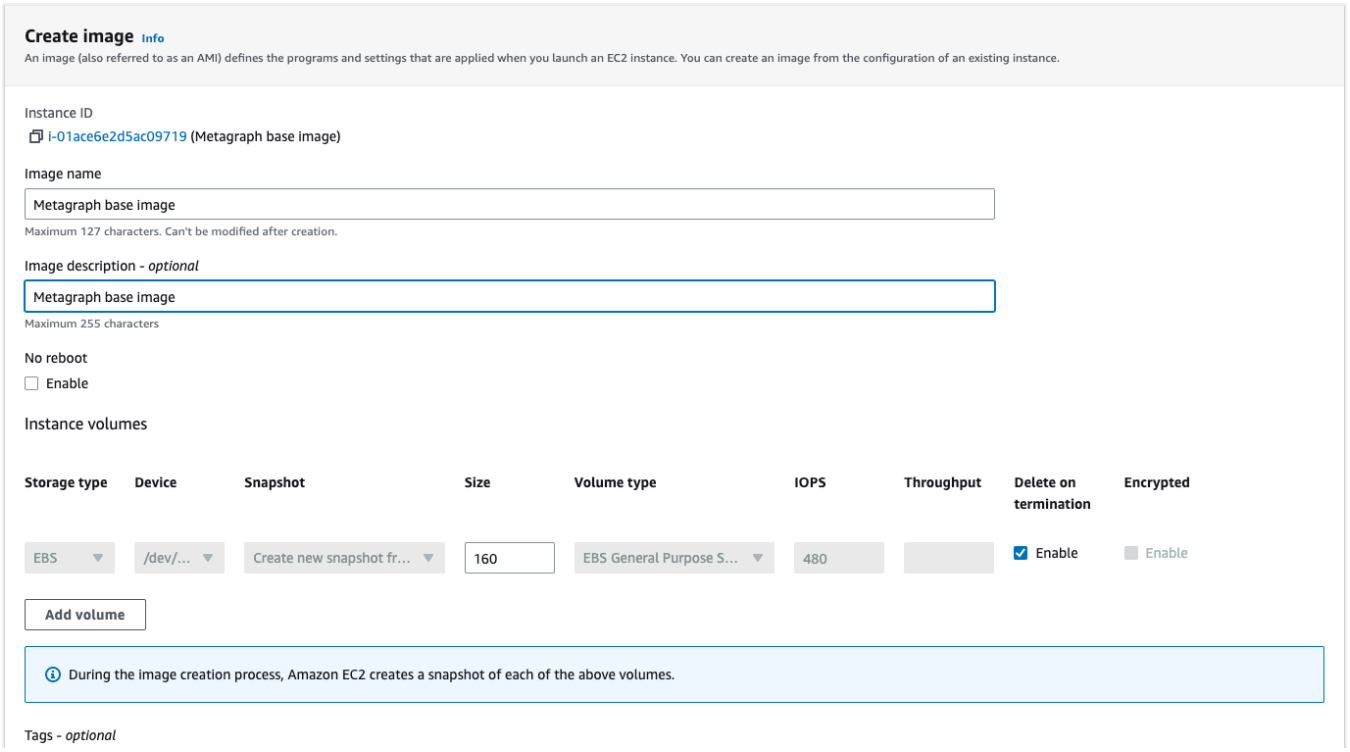
Now that our base instance is configured, we can generate an AMI (Amazon Machine Image) from the instance. The AMI will allow us to create our other two EC2 instances as exact copies of the one we've already configured.

Create the image

To generate the AMI, select your instance and then actions → Image and templates → Create Image

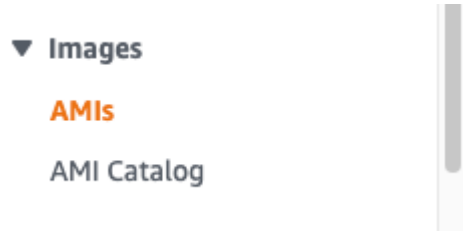


We can repeat the same name when configuring the image:

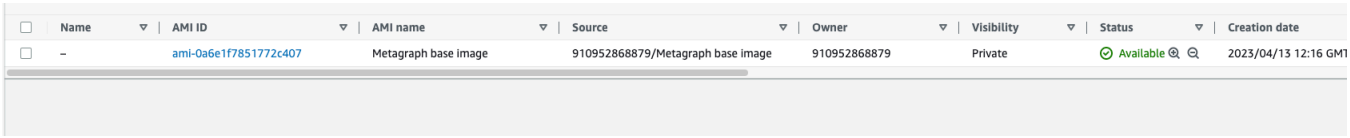


Press Create image

This step will take some time but you can follow progress on the AMIs page.



Wait until the image is in available status



Delete base instance

Once the image is ready we can delete the instance used to generate the image. To do this, go back to the instances page, select the instance, and then press `Terminate instance`.

Edit this page

Launching instances from AMI

The AMI created in the previous step can now be used to generate each of our 3 EC2 instances for our metagraph.

Visit the AMI page

Select the AMI we created previously and press the `Launch instance from AMI` button.

Configure Instance

Name your instance, select the `Instance Type` as `t2.xlarge`, choose your `Key pair`, and select the appropriate `Security Groups`.

Launch Instance

Press the `Launch Instance` button.

Repeat

Perform the above steps 3 times to create 3 EC2 instances from the AMI.

Connect to Instances

Find the ip address of each instance in the EC2 dashboard and connect using your previously generated SSH key. You should be able to access all 3 instances and confirm they are properly configured.

```
ssh -i "MyKeypair.pem" ubuntu@ip.of.your.instance
```

Edit this page

Configuring P12 Files

P12 Files

P12 files contain the public/private keypair for your node to connect to the network which is protected by an alias and a password. These files are necessary for all layers to communicate with each other and validate identity of other nodes. In this step, we will move p12 files to each of the 3 nodes so that they are available when starting each layer of the network.

This guide will use just 3 p12 files in total (1 for each server instance) which is the minimal configuration. For production deployments, it is recommended that each layer and instance has its own p12 file rather than sharing between the layers.

Transfer p12 files

Run the following command to transfer a p12 file to each layer's directory for a single EC2 instance.

Replace `:p12_file.p12` and `your_instance_ip` with your actual p12 file and node IP.

```
scp -i "MyKeypair.pem" :p12_file.p12 your_instance_ip:code/global-l0
scp -i "MyKeypair.pem" :p12_file.p12 your_instance_ip:code/metagraph-l0
scp -i "MyKeypair.pem" :p12_file.p12 your_instance_ip:code/currency-l1
scp -i "MyKeypair.pem" :p12_file.p12 your_instance_ip:code/data-l1
```

Repeat this process for each of your 3 instances.

Make sure to use a different P12 for instance when repeating the above steps.

Your P12 files will now be available on your nodes and you can move on the starting up each layer.

Edit this page

Start Global L0 Instances

In the following sections, we will SSH into each of our 3 servers and configure each layer and then join it to the network. Note that both IntegrationNet and MainNet have seedlists for the Global L0 layer. Make sure your node IDs have been added to the seedlist prior to joining, otherwise you will not be allowed to join.

Setup Global L0

SSH into one of your EC2 instances and move to the `global-l0` directory.

```
ssh -i "MyKeypair.pem" ubuntu@your_instance_ip"
cd code/global-l0
```

Set environment variables

Export the following environment variables, changing the values to use your p12 file's real name, alias, and password.

```
export CL_KEYSTORE=":p12_file.p12"
export CL_KEYALIAS=":p12_file_alias"
export CL_PASSWORD=":p12_password"
```

Obtain public IP

Obtain the public IP of your cluster by using the following command.

```
curl ifconfig.me
```

Download the latest seedlist

Download the latest seedlist from either IntegrationNet or MainNet.

For IntegrationNet, the seedlist is kept in an S3 bucket and can be downloaded directly.

```
wget https://constellationlabs-dag.s3.us-west-1.amazonaws.com/integrationnet-seedlist
```

For MainNet, the seedlist is stored in Github as a build asset for each release. Make sure to fill in the latest version below to get the correct seedlist.

```
wget https://github.com/Constellation-Labs/tessellation/releases/download/v2.2.1/mainnet-seedlist
```

Start your node

The following command will start your Global L0 node in validator mode.

```
nohup java -jar cl-node.jar run-validator --ip :instance_public_ip --public-port 9000 --p2p-port 9001 --cli-port 9002 --collateral 0 --seedlist integrationnet-seedlist -e integrationnet  > logs.log 2>&1 &
```

Check logs

You should see a new directory `logs` with a `app.log` file. Check the logs for any errors.

Join the network

Now that the node is running, we need to join it to a node on the network. You can find a node to connect to using the network load balancer at

```
https://l0-lb-integrationnet.constellationnetwork.io/node/info
```

Run the following command with the **`id`**, **`ip`**, and **`p2pPort`** parameters updated.

```
curl -v -X POST http://localhost:9002/cluster/join -H "Content-type:
application/json" -d '{ "id":":integrationnet_node_id", "ip":
":integrationnet_node_ip", "p2pPort": :integrationnet_node_p2p_port }'
```

Check connection

Verify that your node is connected to the network with the `/node/info` endpoint on your node. It can be accessed at the following url. You should see `state: Ready` if your node has successfully connected to the network.

```
http://your_node_id:9000/node/info
```

Repeat

Repeat the above steps for each of your 3 nodes before moving on to start your metagraph layers.

Edit this page

Start Metagraph L0 Instances

In this section, we will start each of our metagraph L0 instances and join them to the Global L0 network.

Setup Metagraph L0

SSH into one of your EC2 instances and move to the `metagraph-l0` directory.

```
ssh -i "MyKeypair.pem" ubuntu@your_instance_ip"
cd code/metagraph-l0
```

Set environment variables

Export the following environment variables, changing the values to use your p12 file's real name, alias, and password.

```
export CL_KEYSTORE=":p12_file_used_on_seedlist_1.p12"
export CL_KEYALIAS=":p12_file_used_on_seedlist_1"
export CL_PASSWORD=":file_password_1"
```

Also export the following environment variables, filling in `CL_GLOBAL_L0_PEER_ID` with the public ID of your Global L0 node which can be obtained from the `/node/info` endpoint at the end of the previous step. This is also the pub ID of your p12 file.

```
export CL_PUBLIC_HTTP_PORT=9100
export CL_P2P_HTTP_PORT=9101
export CL_CLI_HTTP_PORT=9102
export CL_GLOBAL_L0_PEER_HTTP_HOST=localhost
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
```

```
export CL_GLOBAL_L0_PEER_ID=:local_global_l0_id
CL_APP_ENV=integrationnet
export CL_COLLATERAL=0
```

Start your metagraph L0 node (genesis)

> **Note**
>
> Run this command only on the first of your instances. When you repeat these steps for the 2nd and 3rd instance, use the `run-validator` joining process below instead.

Use the following command to start the metagraph L0 process in genesis mode. This should only be done once to start your network from the genesis snapshot. In the future, to restart the network use `run-rollback` instead to restart from the most recent snapshot. Fill in the `:instance_ip` variable with the public IP address of your node.

```
nohup java -jar metagraph-l0.jar run-genesis genesis.snapshot --ip :instance_ip > metagraph-l0-logs.log 2>&1 &
```

You can check if your metagraph L0 successfully started using the `/cluster/info` endpoint or by checking logs.

```
http://:your_ip:9100/cluster/info
```

Start your metagraph L0 node (validator)

The 2nd and 3rd nodes should be started in validator mode and joined to the first node that was run in genesis or run-rollback mode. All other steps are the same.

```
nohup java -jar metagraph-l0.jar run-validator --ip :ip > metagraph-l0-logs.log 2>&1 &
```

Once the node is running in validator mode, we need to join it to the first node using the following command

```
curl -v -X POST http://localhost:9102/cluster/join -H "Content-type:
application/json" -d '{ "id":":metagraph_node_1_id", "ip": "metagraph_node_1_ip",
"p2pPort": 9101 }'
```

You can check if the nodes successfully started using the `/cluster/info` endpoint for your metagraph L0. You should see nodes appear in the list if all started properly.
http://:your_ip:9100/cluster/info

Repeat

Repeat the above steps for each of your 3 nodes before moving on to start your metagraph L1 layers. Note that the startup commands differ between the three nodes. The first node should be started in genesis or run-rollback mode. The second and third nodes should be started in validator mode and joined to the first node.

Start Currency L1 Instances

In this section, we will start each of our currency L1 instances and join them to the metagraph L0 network.

Setup Currency L1

SSH into one of your EC2 instances and move to the `currency-l1` directory.

```
ssh -i "MyKeypair.pem" ubuntu@your_instance_ip"
cd code/currency-l1
```

Set environment variables

Export the following environment variables, changing the values to use your p12 file's real name, alias, and password.

```
export CL_KEYSTORE=":p12_file_used_on_seedlist_1.p12"
export CL_KEYALIAS=":p12_file_used_on_seedlist_1"
export CL_PASSWORD=":file_password_1"
```

Also export the following environment variables, filling in the following:

- `CL_GLOBAL_L0_PEER_ID`: The public ID of your Global L0 node which can be obtained from the `/node/info` endpoint of your Global L0 instance (http://your_node_id:9000/node/info).
- `CL_L0_PEER_ID`: The public ID of the metagraph l0 node which is the same as `CL_GLOBAL_L0_PEER_ID` above if you're using the same p12 files for all layers.
- `CL_GLOBAL_L0_PEER_HTTP_HOST`: The public IP of this node (points to global-l0 layer).

- `CL_L0_PEER_HTTP_HOST` : The public IP of this node (points to metagraph-l0 layer).
- `CL_L0_TOKEN_IDENTIFIER` : The metagraph ID in your address.genesis file.

```
export CL_PUBLIC_HTTP_PORT=9200
export CL_P2P_HTTP_PORT=9201
export CL_CLI_HTTP_PORT=9202
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip_from_metagraph_l0_node_1_global_l0
export CL_GLOBAL_L0_PEER_ID=:id_from_metagraph_l0_node_1_global_l0
export CL_L0_PEER_HTTP_HOST=:ip_from_metagraph_l0_node_1_metagraph_l0
export CL_L0_PEER_HTTP_PORT=9100
export CL_L0_PEER_ID=:id_from_metagraph_l0_node_1_metagraph_l0
export CL_L0_TOKEN_IDENTIFIER=:**METAGRAPH_ID**
export CL_APP_ENV=integrationnet
export CL_COLLATERAL=0
```

Start your currency L1 node (initial)

> **Note**
>
> Run this command only on the first of your instances. When you repeat these steps for the 2nd and 3rd instance, use the `run-validator` joining process below instead.

Run the following command, filling in the public ip address of your instance.

```
nohup java -jar currency-l1.jar run-initial-validator --ip :instance_ip > metagprah-l1-logs.log 2>&1 &
```

Check if your Currency L1 successfully started: http://:your_ip:9200/cluster/info

Start your currency L1 node (validator)

The 2nd and 3rd nodes should be started in validator mode and joined to the first node that was run in initial-validator mode. All other steps are the same.

```
```

```
nohup java -jar currency-l1.jar run-validator --ip :ip > currency-l1-logs.log 2>&1 &
```

Run the following command to join, filling in the `id` and `ip` of your first currency L1 node.

```
curl -v -X POST http://localhost:9202/cluster/join -H "Content-type:
application/json" -d '{ "id":":id_from_currency_l1_1", "ip": ":ip_from_currency_l1",
"p2pPort": 9201 }'
```

You can check if the nodes successfully started using the `/cluster/info` endpoint for your metagraph L0. You should see nodes appear in the list if all started properly.
http://:your_ip:9200/cluster/info

Repeat

Repeat the above steps for each of your 3 currency L1 nodes before moving on to start your data L1 layer. Note that the startup commands differ between the three nodes. The first node should be started in initial-validator mode. The second and third nodes should be started in validator mode and joined to the first node.

Start Data L1 Instances

In this section, we will start each of our data L1 instances and join them to the metagraph L0 network.

Setup Data L1

SSH into one of your EC2 instances and move to the `data-l1` directory.

```
ssh -i "MyKeypair.pem" ubuntu@your_instance_ip"
cd code/data-l1
```

Set environment variables

Export the following environment variables, changing the values to use your p12 file's real name, alias, and password.

```
export CL_KEYSTORE=":p12_file_used_on_seedlist_1.p12"
export CL_KEYALIAS=":p12_file_used_on_seedlist_1"
export CL_PASSWORD=":file_password_1"
```

Also export the following environment variables, filling in the following:

- `CL_GLOBAL_L0_PEER_ID`: The public ID of your Global L0 node which can be obtained from the `/node/info` endpoint of your Global L0 instance (http://your_node_id:9000/node/info).
- `CL_L0_PEER_ID`: The public ID of the metagraph l0 node which is the same as `CL_GLOBAL_L0_PEER_ID` above if you're using the same p12 files for all layers.
- `CL_GLOBAL_L0_PEER_HTTP_HOST`: The public IP of this node (points to global-l0 layer).

- `CL_L0_PEER_HTTP_HOST` : The public IP of this node (points to metagraph-l0 layer).
- `CL_L0_TOKEN_IDENTIFIER` : The metagraph ID in your address.genesis file.

```
export CL_PUBLIC_HTTP_PORT=9300
export CL_P2P_HTTP_PORT=9301
export CL_CLI_HTTP_PORT=9302
export CL_GLOBAL_L0_PEER_HTTP_HOST=:ip_from_metagraph_l0_node_1_global_l0
export CL_GLOBAL_L0_PEER_HTTP_PORT=9000
export CL_GLOBAL_L0_PEER_ID=:id_from_metagraph_l0_node_1_global_l0
export CL_L0_PEER_HTTP_HOST=:ip_from_metagraph_l0_node_1_metagraph_l0
export CL_L0_PEER_HTTP_PORT=9100
export CL_L0_PEER_ID=:id_from_metagraph_l0_node_1_metagraph_l0
export CL_L0_TOKEN_IDENTIFIER=:**METAGRAPH_ID**
export CL_APP_ENV=integrationnet
export CL_COLLATERAL=0
```

Start your data L1 node (initial)

> **Note**
>
> Run this command only on the first of your instances. When you repeat these steps for the 2nd and 3rd instance, use the `run-validator` joining process below instead.

Run the following command, filling in the public ip address of your instance.

```
nohup java -jar data-l1.jar run-initial-validator --ip :instance_ip > metagprah-l1-logs.log 2>&1 &
```

Check if your data L1 node successfully started: http://:your_ip:9300/cluster/info

Start your data L1 node (validator)

The 2nd and 3rd nodes should be started in validator mode and joined to the first node that was run in initial-validator mode. All other steps are the same.

```
```

```
nohup java -jar data-l1.jar run-validator --ip :ip > data-l1-logs.log 2>&1 &
```

Run the following command to join, filling in the `id` and `ip` of your first data L1 node.

```
curl -v -X POST http://localhost:9302/cluster/join -H "Content-type:
application/json" -d '{ "id":":id_from_data_l1_1", "ip": ":ip_from_data_l1",
"p2pPort": 9301 }'
```

Repeat

Repeat the above steps for each of your 3 data L1 nodes before moving on to start your data L1 layer. Note that the startup commands differ between the three nodes. The first node should be started in initial-validator mode. The second and third nodes should be started in validator mode and joined to the first node.

Verify

If you followed all steps, your metagraph is now fully deployed.

You can check the status of each of the node layers using their IP address and layer port number.

Ports

- Global L0: 9000
- Metagraph L0: 9100
- Currency L1: 9200
- Data L1: 9300

Endpoints

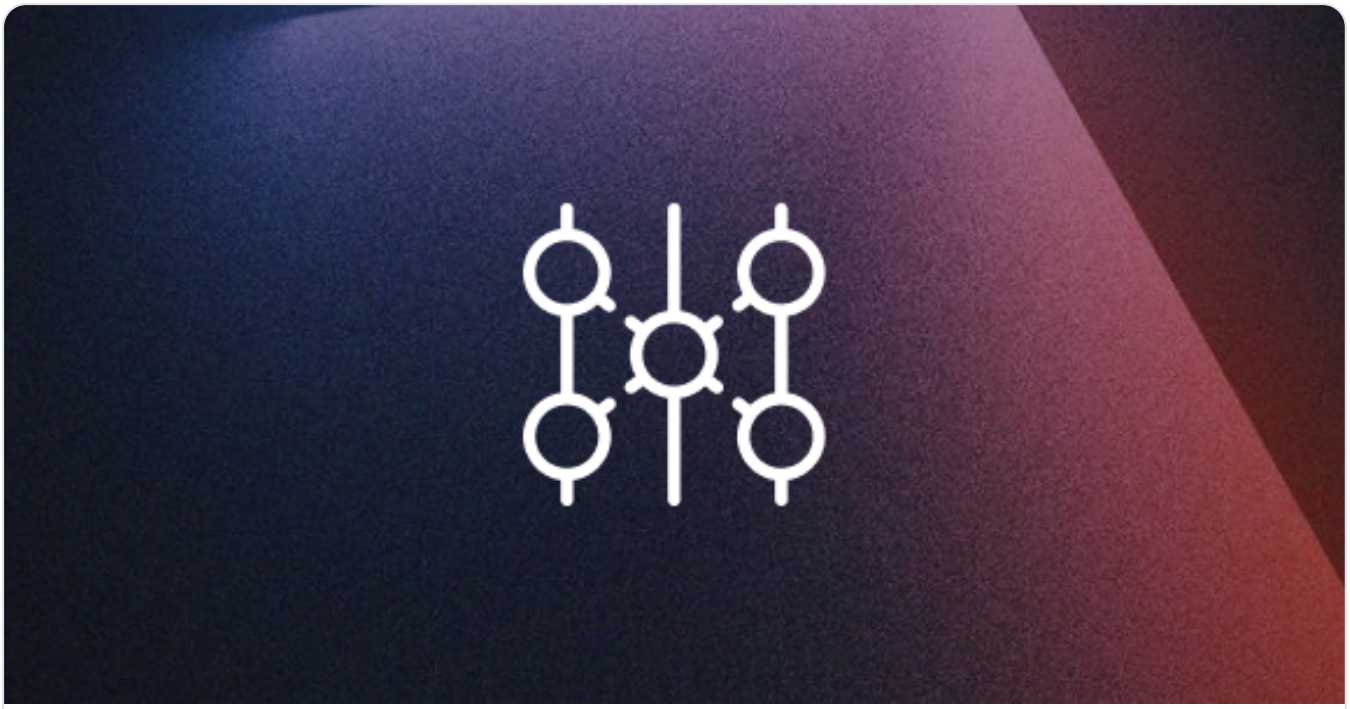
- `/cluster/info`: View nodes joined to the current layer's cluster

- `/node/info`: View info about a specific node and its status

Edit this page

Network APIs

Metagraph networks (metagraph L0, currency L1, and data L1) and Hypergraph Networks (Global L0 and DAG L1) are all accessible via REST API endpoints. The endpoints share a similar structure and composition for easy integration into wallets, exchanges, and dApps.



Network APIs

Explore network APIs

Edit this page

Example Codebases

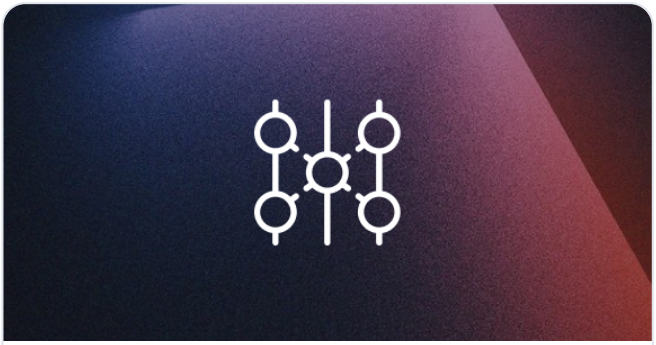
The Euclid SDK is designed to provide developers with the tools they need to build robust and scalable decentralized applications on the Constellation Network. To help you get started, we have curated a list of exemplary codebases that you can explore and learn from. These codebases are open-source projects that demonstrate various aspects of using the Euclid SDK in real-world scenarios.

Codebases

Metagraph Examples

The Metagraph Examples repository contains several minimalist metagraph codebases designed to demonstrate specific metagraph features in a simplified context. All projects in this repo can be installed with `hydra install-template`

Displays: many concepts.



Dor Metagraph

This repository is the codebase of the Dor Metagraph, the first metagraph to launch to MainNet. The Dor Metagraph ingests foot traffic data from a network of IoT sensors.

Displays: strategies for processing binary data types using decoders, reward distribution, and separation of public/private data using calculated state.

EL PACA Metagraph

This repository is the codebase of the EL PACA

Edit this page