# System Architecture Document

version 1.0

for

# Maelstrom

prepared by

| name | email |
|---|---|
| Iulian Rotaru | iulian@rotaru.fr |

# Document History

| Date | Version | Description | Author |
|---|---|---|---|
| 15/12/2018 | 1.0 | Filled Version 1.0 | Iulian Rotaru |
| | | | |
| | | | |
| | | | |

# Table of Contents

# 1. Introduction

The introduction of the Software Architecture Document provides an overview of the entire document.

## 1.1. Purpose

The Software Architecture Document provides a technical definition of the Maelstrom System.

## 1.2. Scope

The Software Architecture Document is describing in a technical manner the Maelstrom Daemon and the Maelstrom CLI. The document was created for technical members of the team working on Maelstrom.

## 1.3. Definitions, acronyms, and abbreviations

| | |
|---|---|
| SAD | Software Architecture Document |
| DB | Database |
| CSV | Comma-Separated Values |
| SRS | System Requirements Specification |
| UC | Use Case |
| CLI | Command Line Interface |

# 2. Architectural representation

The architectural representation describes the top-level architectural style of the system. It follows the classic 4+1 view model

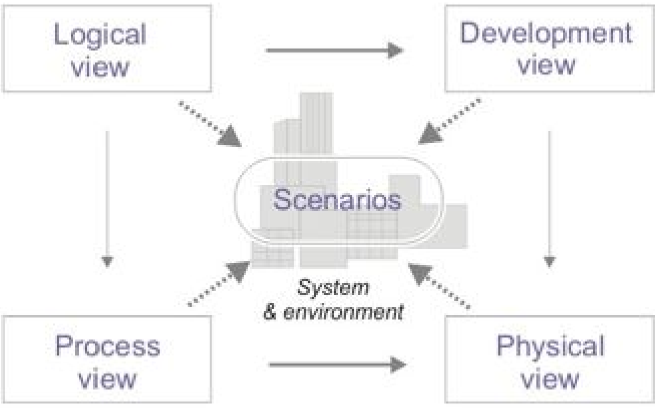


*Figure 2.1: The 4+1 view model.*

## 2.1. Logical view

This section is made for the Software Designers. The Logical View is concerned with the functionality that the system provides to end-users, in our case the Administrator.

### 2.1.1. Layers and tiers

The following diagram defines the logical architecture of the system. The system follows the 3-tier architecture pattern.

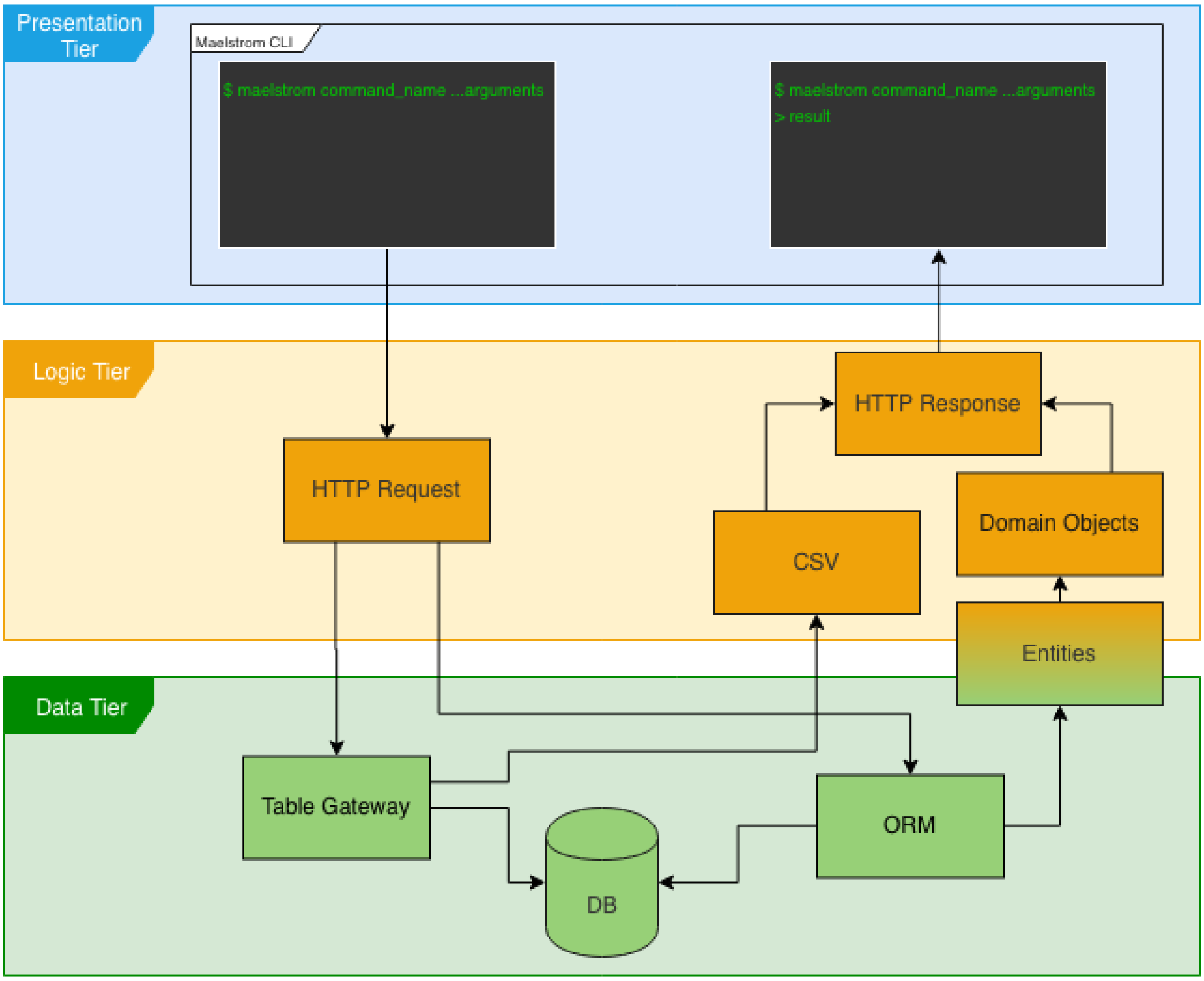


*Figure 2.1.1.1: 3-tier architecture*

### 2.1.2. Subsystems

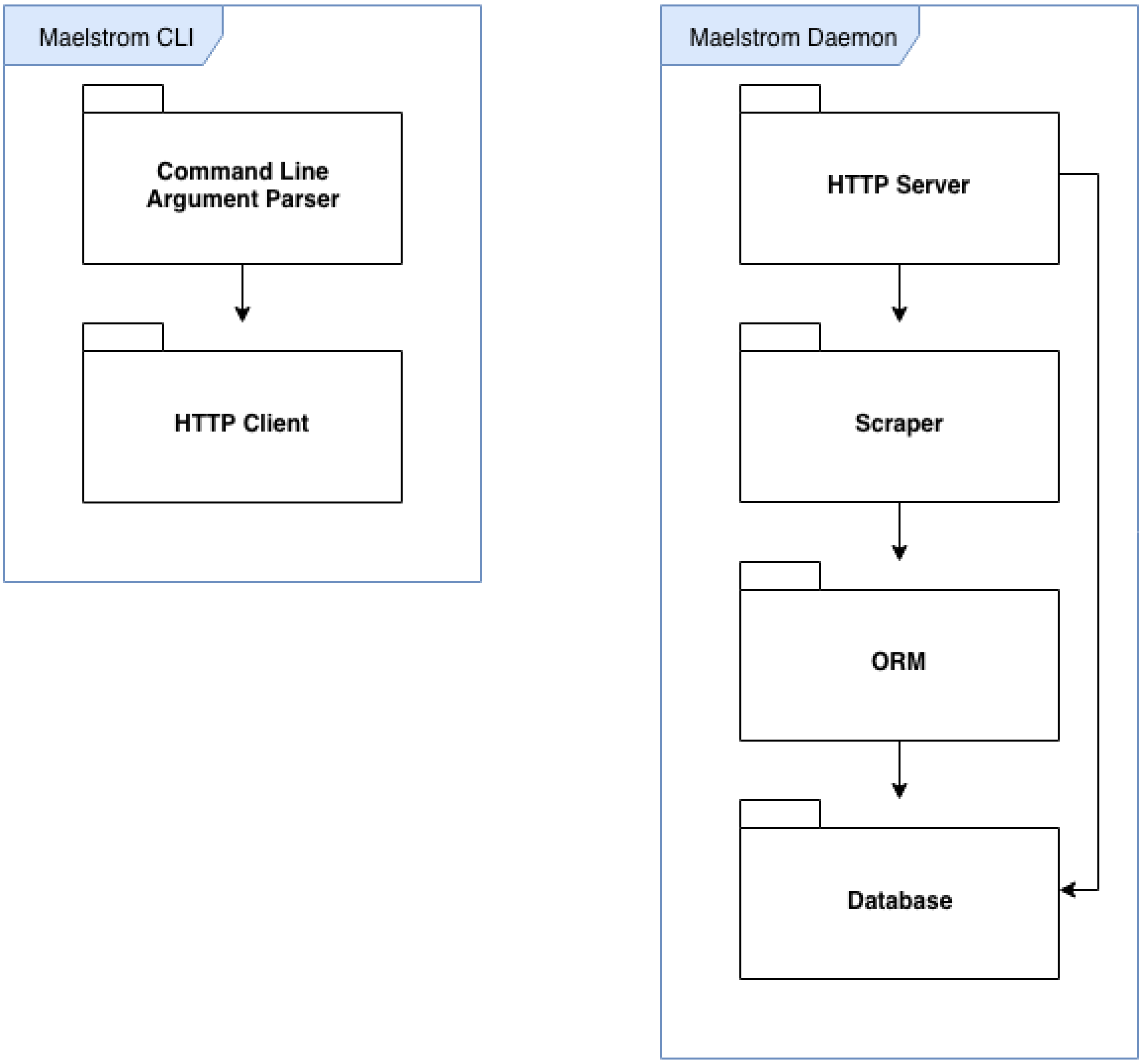


*Figure 2.1.2.1.Figure 2.1.2.2: Maelstrom CLI Subsystems,Maelstrom Daemon Subsystems*

### 2.1.3. Use case realizations

Use case realizations define detailed internal interaction and communications between entities for user use cases only.

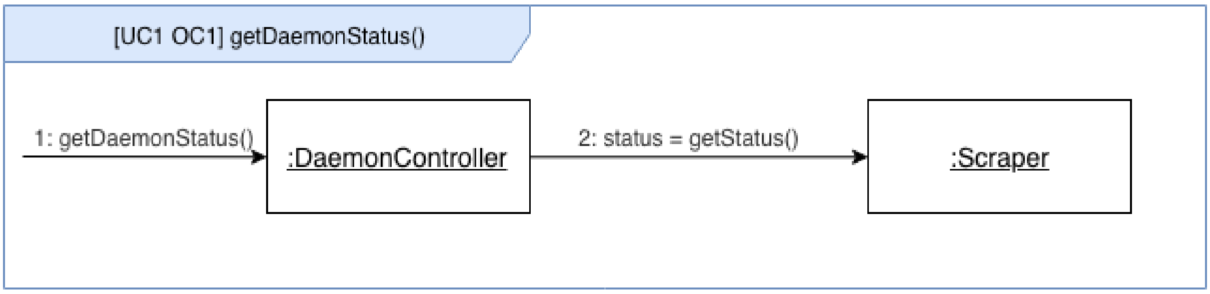


*Figure 2.1.3.1: [UC1 OC1] getDaemonStatus() communication diagram*

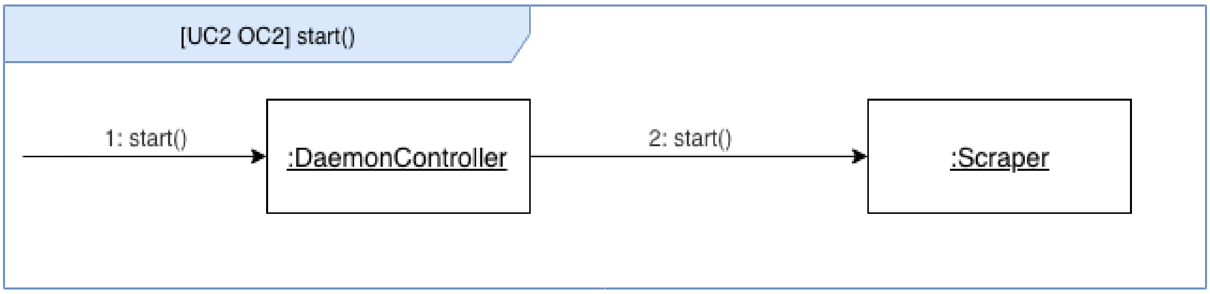


*Figure 2.1.3.2: [UC2 OC2] start() communication diagram*

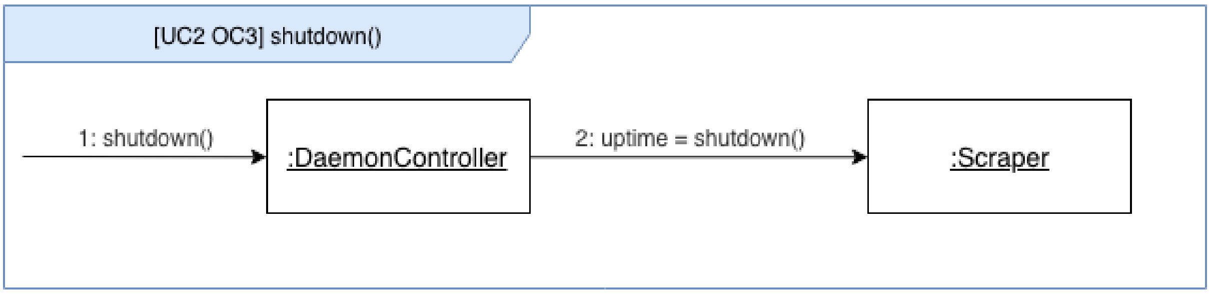


*Figure 2.1.3.3: [UC2 OC3] shutdown() communication diagram*

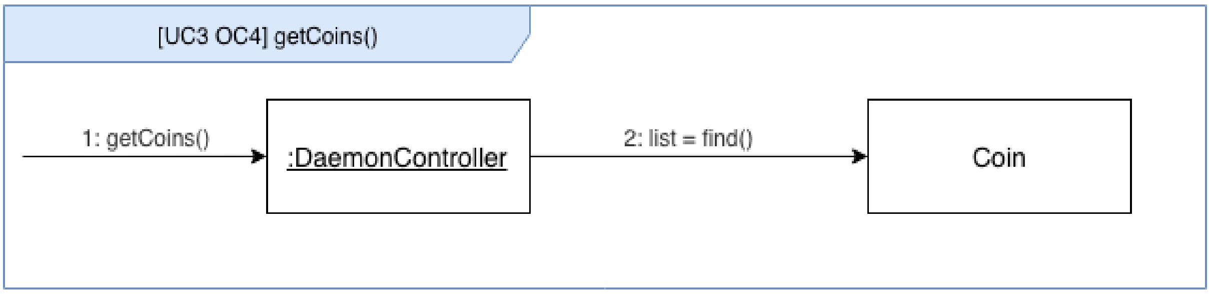


*Figure 2.1.3.4: [UC3 OC4] getCoins() communication diagram*

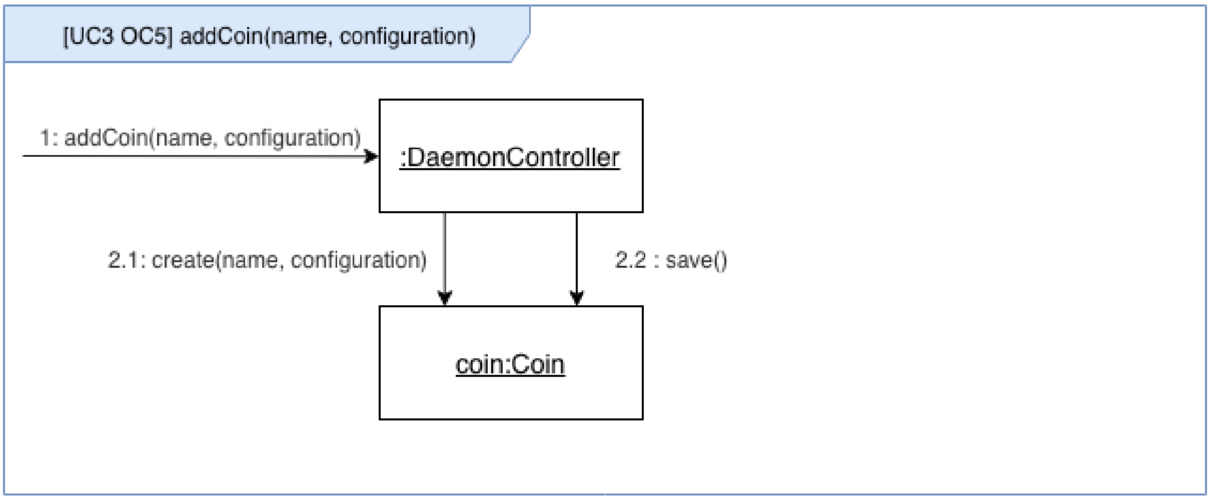

[UC3 OC5] addCoin(name, configuration)
1: addCoin(name, configuration)
:DaemonController
2.1: create(name, configuration)
2.2 : save()
coin:Coin

*Figure 2.1.3.5: [UC3 OC5] addCoin(name, configuration) communication diagram*

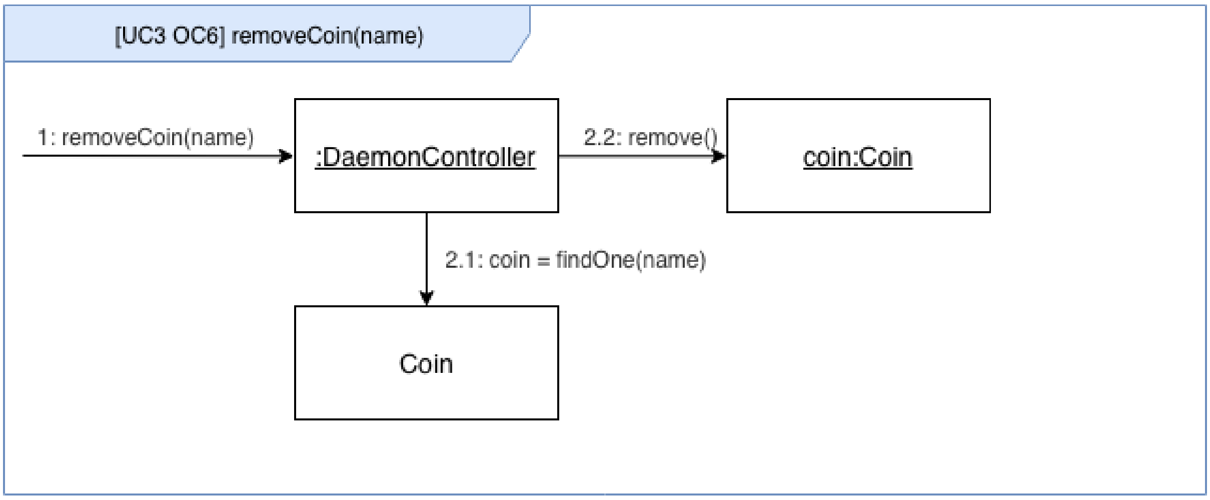


*Figure 2.1.3.6: [UC3 OC6] removeCoin(name) communication diagram*

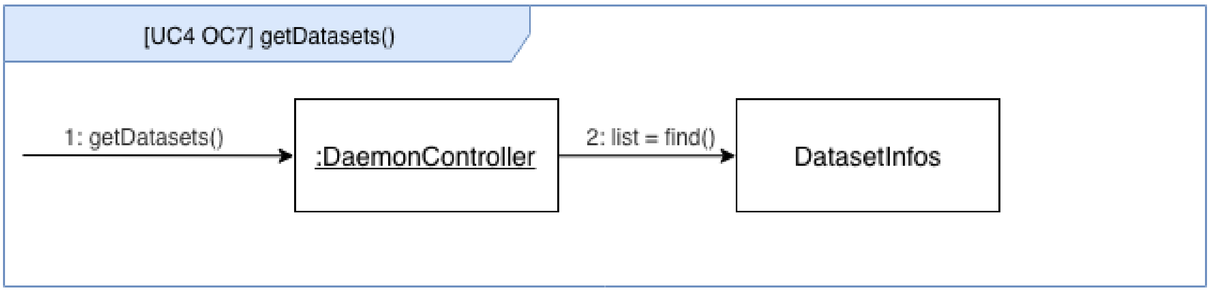


*Figure 2.1.3.7: [UC4 OC7] getDatasets() communication diagram*

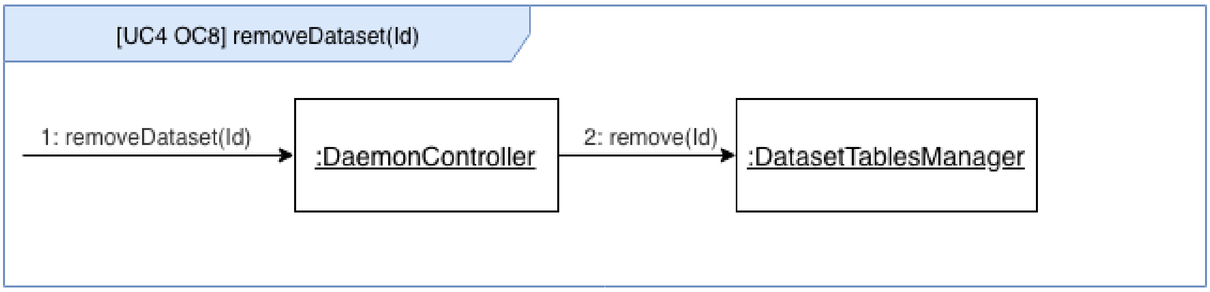


*Figure 2.1.3.8: [UC4 OC8] removeDataset(Id) communication diagram*

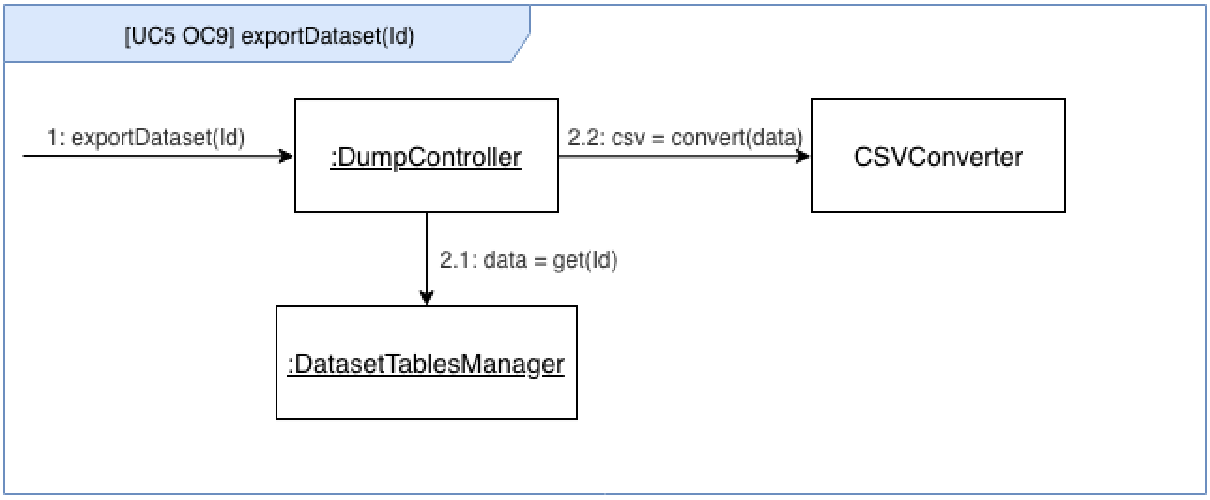


*Figure 2.1.3.9: [UC4 OC9] exportDataset(Id) communication diagram*

## 2.2. Development view

This section was made for Developers. It captures the different classes from the system and show their visibility between each other.

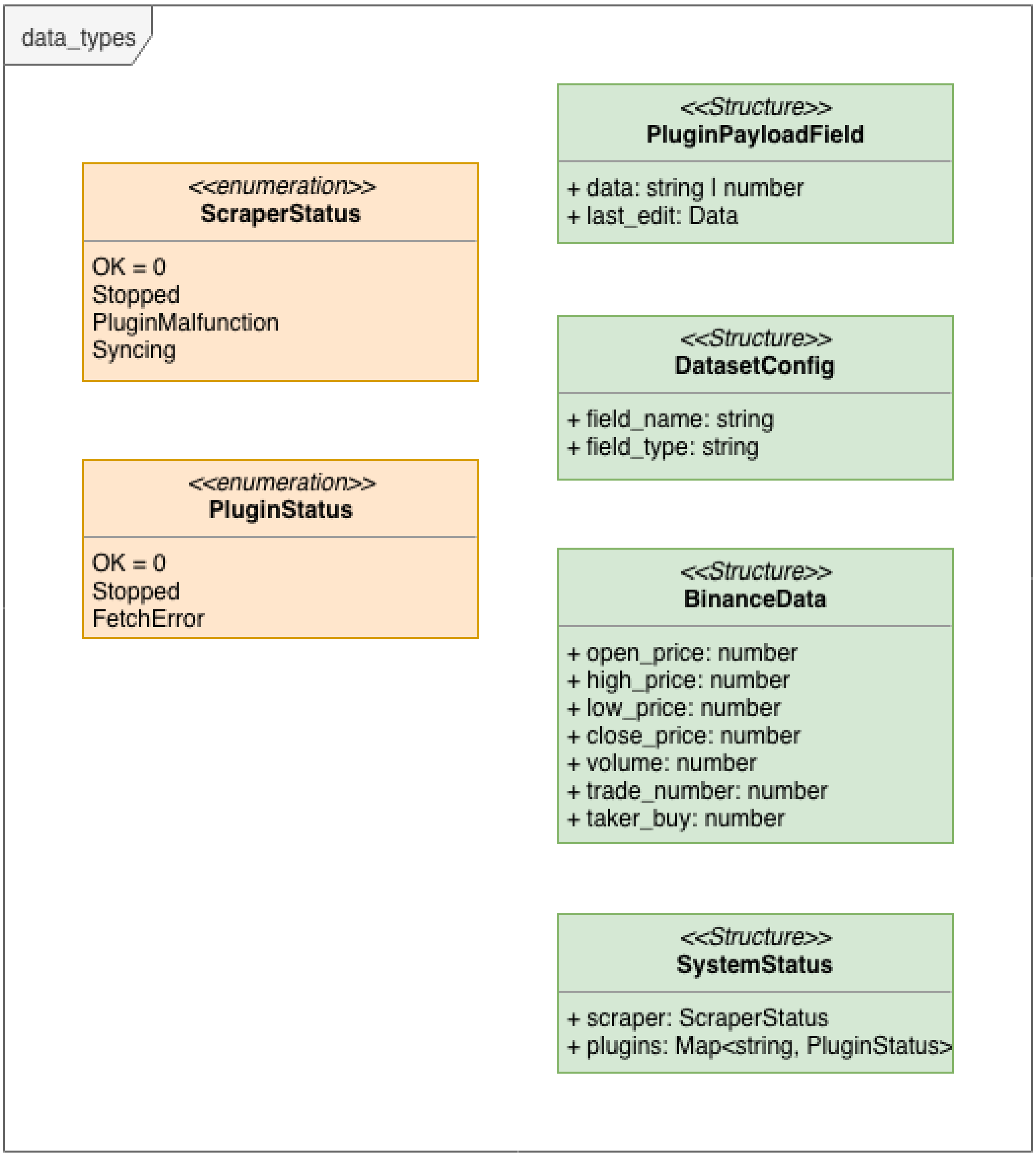


*Figure 2.2.1: data_types package*

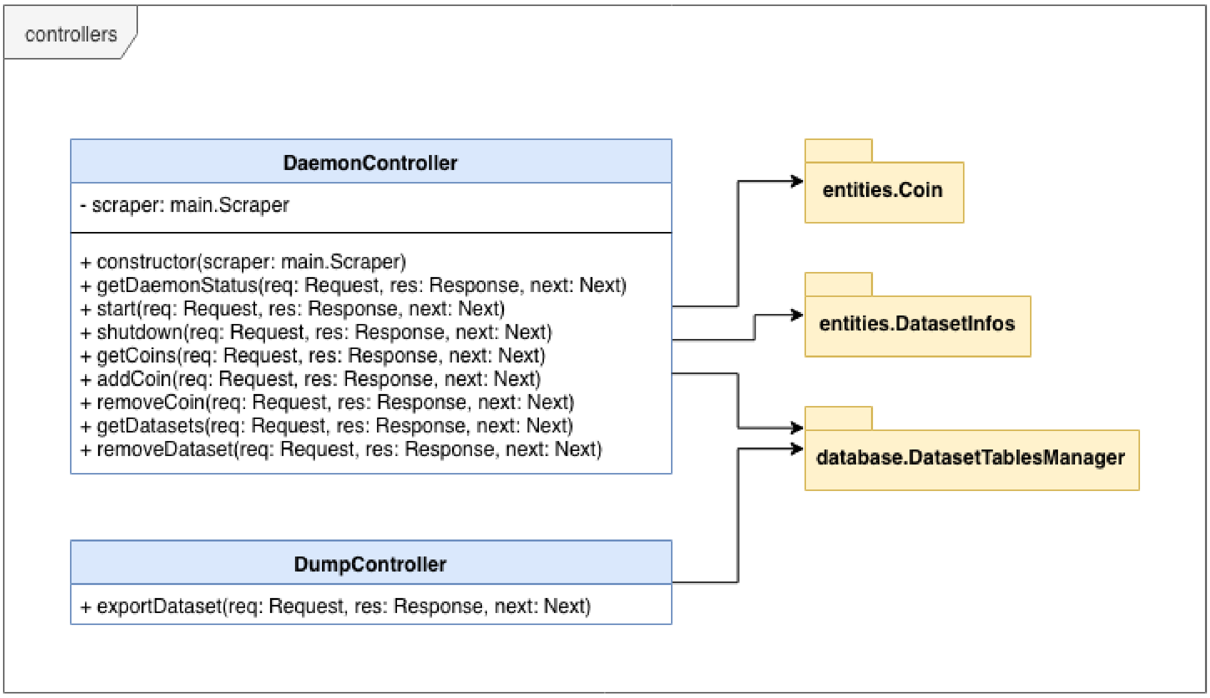

controllers
DaemonController
- scraper: main.Scraper
+ constructor(scraper: main.Scraper)
+ getDaemonStatus(req: Request, res: Response, next: Next)
+ start(req: Request, res: Response, next: Next)
+ shutdown(req: Request, res: Response, next: Next)
+ getCoins(req: Request, res: Response, next: Next)
+ addCoin(req: Request, res: Response, next: Next)
+ removeCoin(req: Request, res: Response, next: Next)
+ getDatasets(req: Request, res: Response, next: Next)
+ removeDataset(req: Request, res: Response, next: Next)
DumpController
+ exportDataset(req: Request, res: Response, next: Next)
entities.Coin
entities.DatasetInfos
database.DatasetTablesManager

*Figure 2.2.2: controllers package*

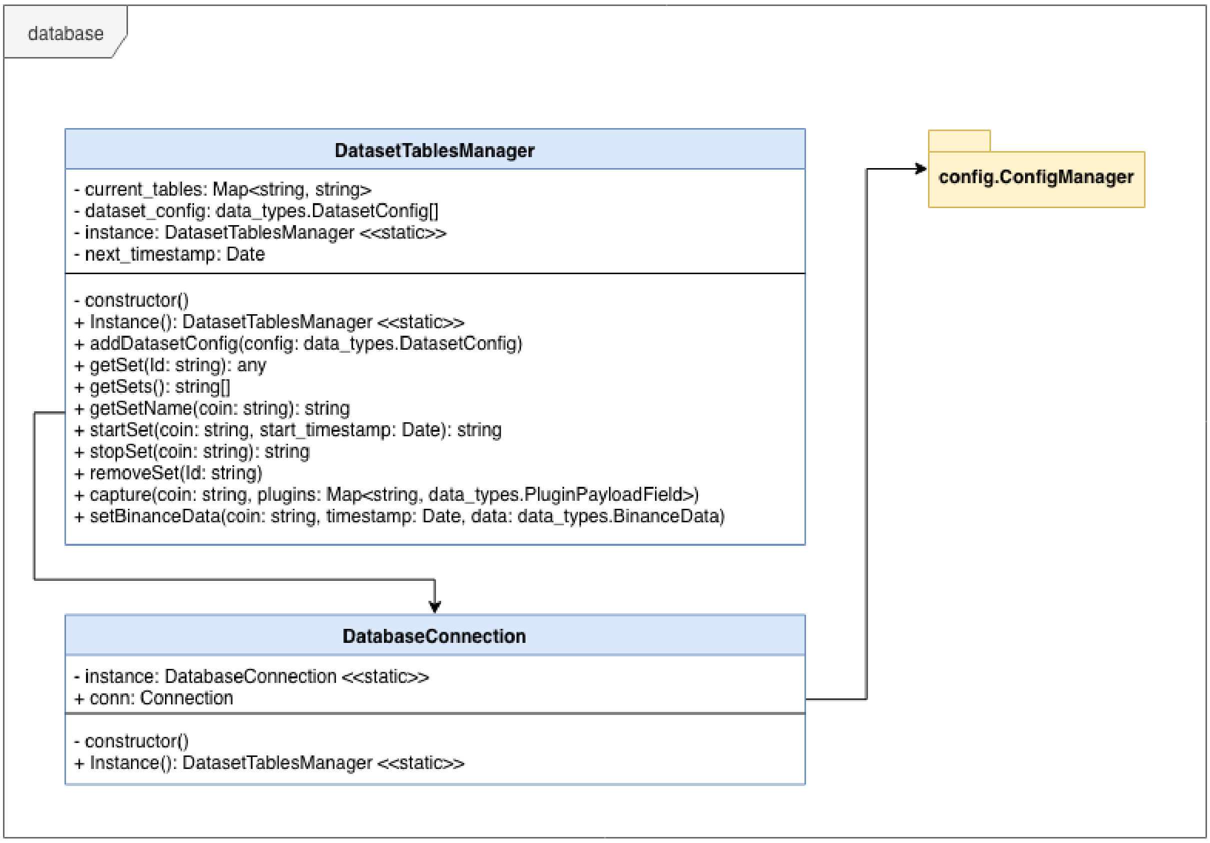

database
DatasetTablesManager
- current_tables: Map<string, string>
- dataset_config: data_types.DatasetConfig[]
- instance: DatasetTablesManager <<static>>
- next_timestamp: Date
- constructor()
+ Instance(): DatasetTablesManager <<static>>
+ addDatasetConfig(config: data_types.DatasetConfig)
+ getSet(Id: string): any
+ getSets(): string[]
+ getSetName(coin: string): string
+ startSet(coin: string, start_timestamp: Date): string
+ stopSet(coin: string): string
+ removeSet(Id: string)
+ capture(coin: string, plugins: Map<string, data_types.PluginPayloadField>)
+ setBinanceData(coin: string, timestamp: Date, data: data_types.BinanceData)
DatabaseConnection
- instance: DatabaseConnection <<static>>
+ conn: Connection
- constructor()
+ Instance(): DatasetTablesManager <<static>>
config.ConfigManager

*Figure 2.2.3: database package*

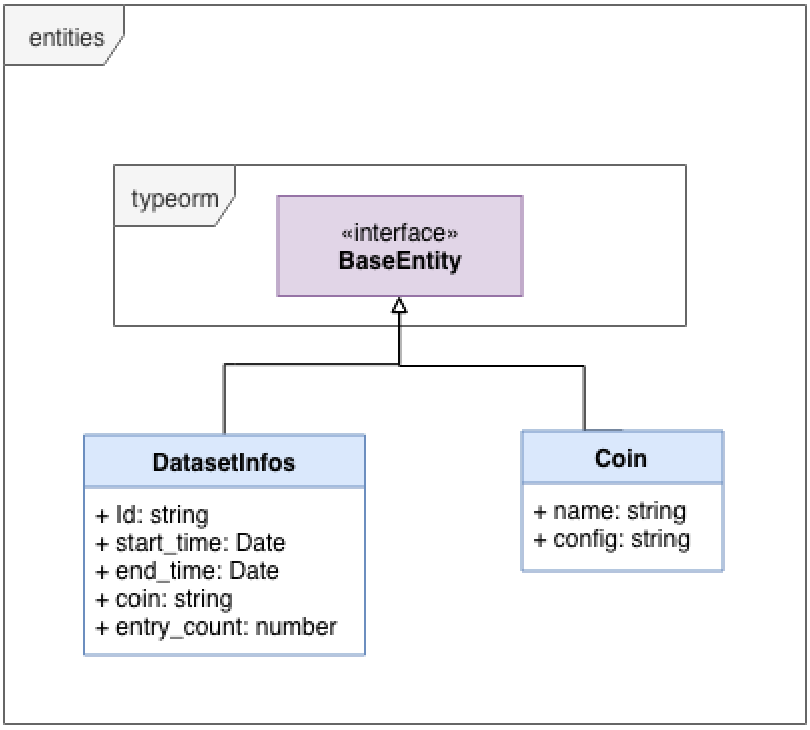


*Figure 2.2.4: entities package*

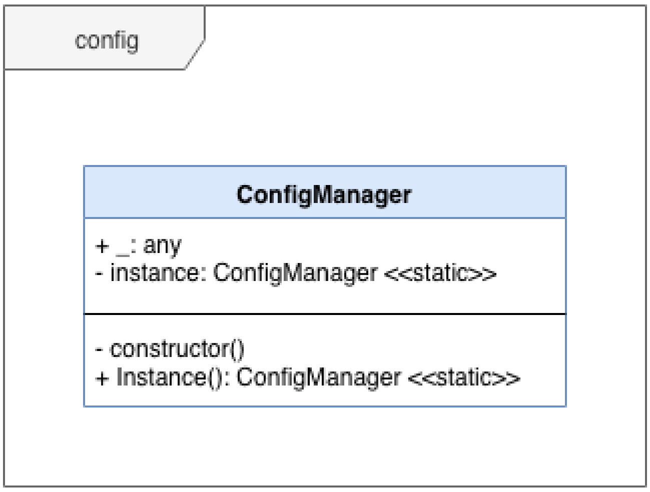

config
ConfigManager
+ _: any
- instance: ConfigManager <<static>>
- constructor()
+ Instance(): ConfigManager <<static>>

*Figure 2.2.5: config package*

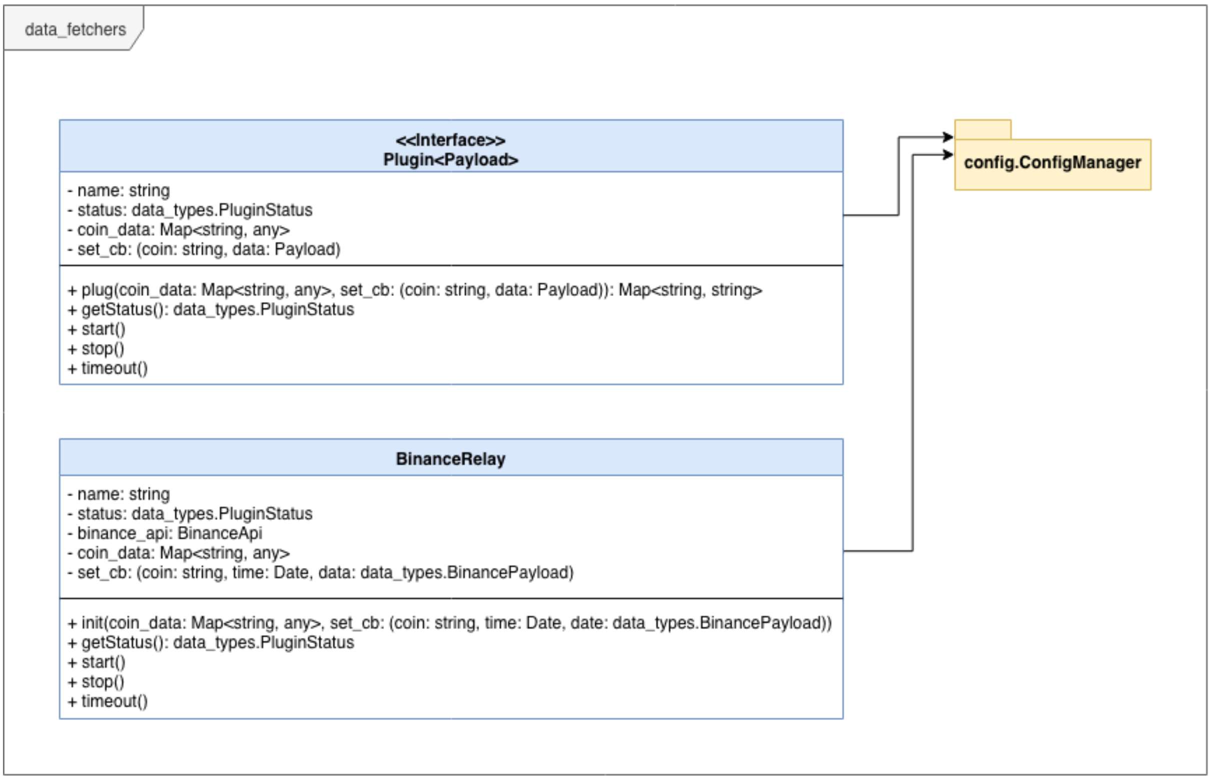

data_fetchers
<<Interface>>
Plugin<Payload>
- name: string
- status: data_types.PluginStatus
- coin_data: Map<string, any>
- set_cb: (coin: string, data: Payload)
+ plug(coin_data: Map<string, any>, set_cb: (coin: string, data: Payload)): Map<string, string>
+ getStatus(): data_types.PluginStatus
+ start()
+ stop()
+ timeout()
BinanceRelay
- name: string
- status: data_types.PluginStatus
- binance_api: BinanceApi
- coin_data: Map<string, any>
- set_cb: (coin: string, time: Date, data: data_types.BinancePayload)
+ init(coin_data: Map<string, any>, set_cb: (coin: string, time: Date, date: data_types.BinancePayload))
+ getStatus(): data_types.PluginStatus
+ start()
+ stop()
+ timeout()
config.ConfigManager

*Figure 2.2.6: data_fetchers package*

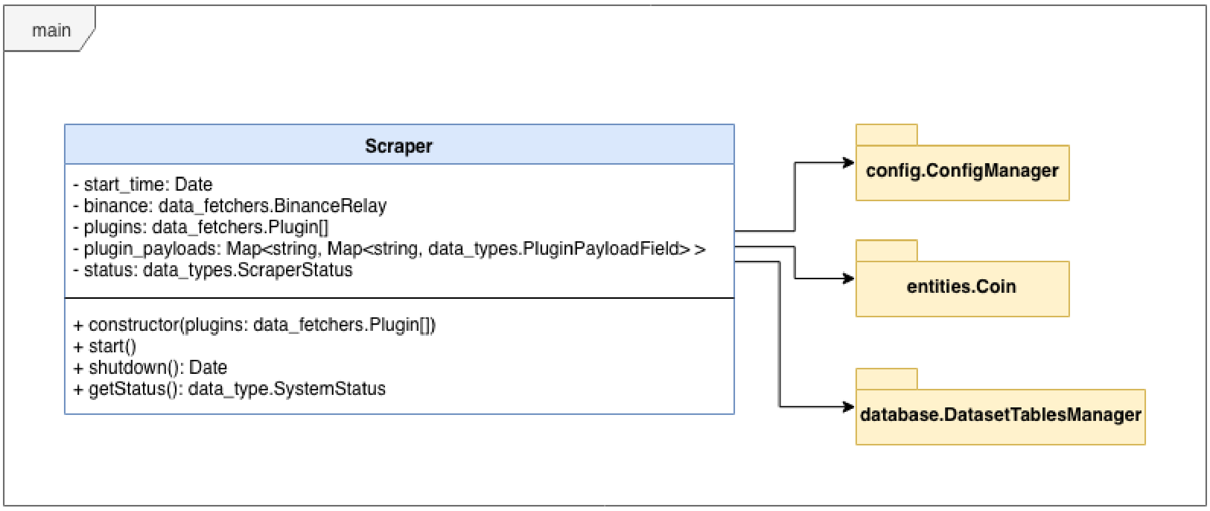

main
Scraper
- start_time: Date
- binance: data_fetchers.BinanceRelay
- plugins: data_fetchers.Plugin[]
- plugin_payloads: Map<string, Map<string, data_types.PluginPayloadField> >
- status: data_types.ScraperStatus
+ constructor(plugins: data_fetchers.Plugin[])
+ start()
+ shutdown(): Date
+ getStatus(): data_type.SystemStatus
config.ConfigManager
entities.Coin
database.DatasetTablesManager

*Figure 2.2.7: main package*

## 2.2.1. Reuse of components and frameworks

There is no component reuse.

## 2.3. Process view

This section is made for System Integrators and deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. This view addresses concurrency, distribution, processes, performances, and scalability.

The System is composed by a single threaded Node.js process running on a machine. This process communicates with a CLI via HTTP requests, exposing a REST API.

## 2.4. Physical view

This section is made for Deployment Managers and System Administrators. This view is concerned with the topology of software components on the physical layer, as well as the physical connections between these components.

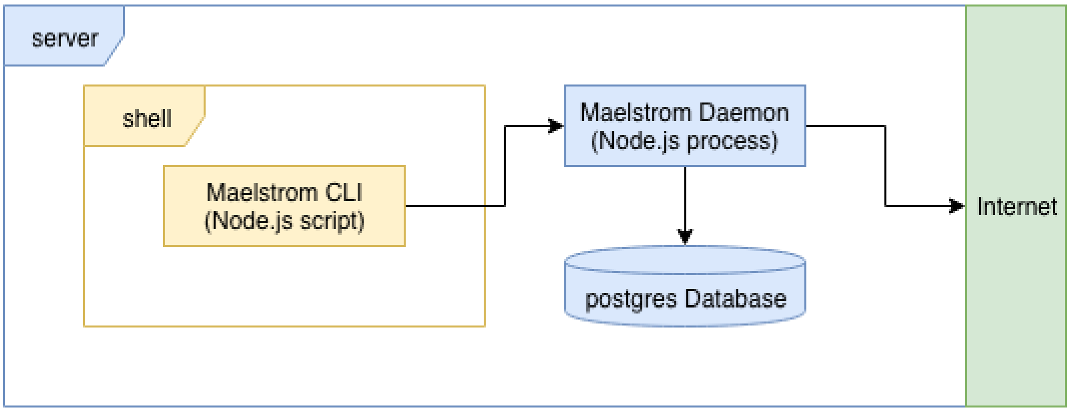

server
shell
Maelstrom CLI (Node.js script)
Maelstrom Daemon (Node.js process)
postgres Database
Internet

*Figure 2.4.1: Physical View Diagram*

## 2.5. Use case view

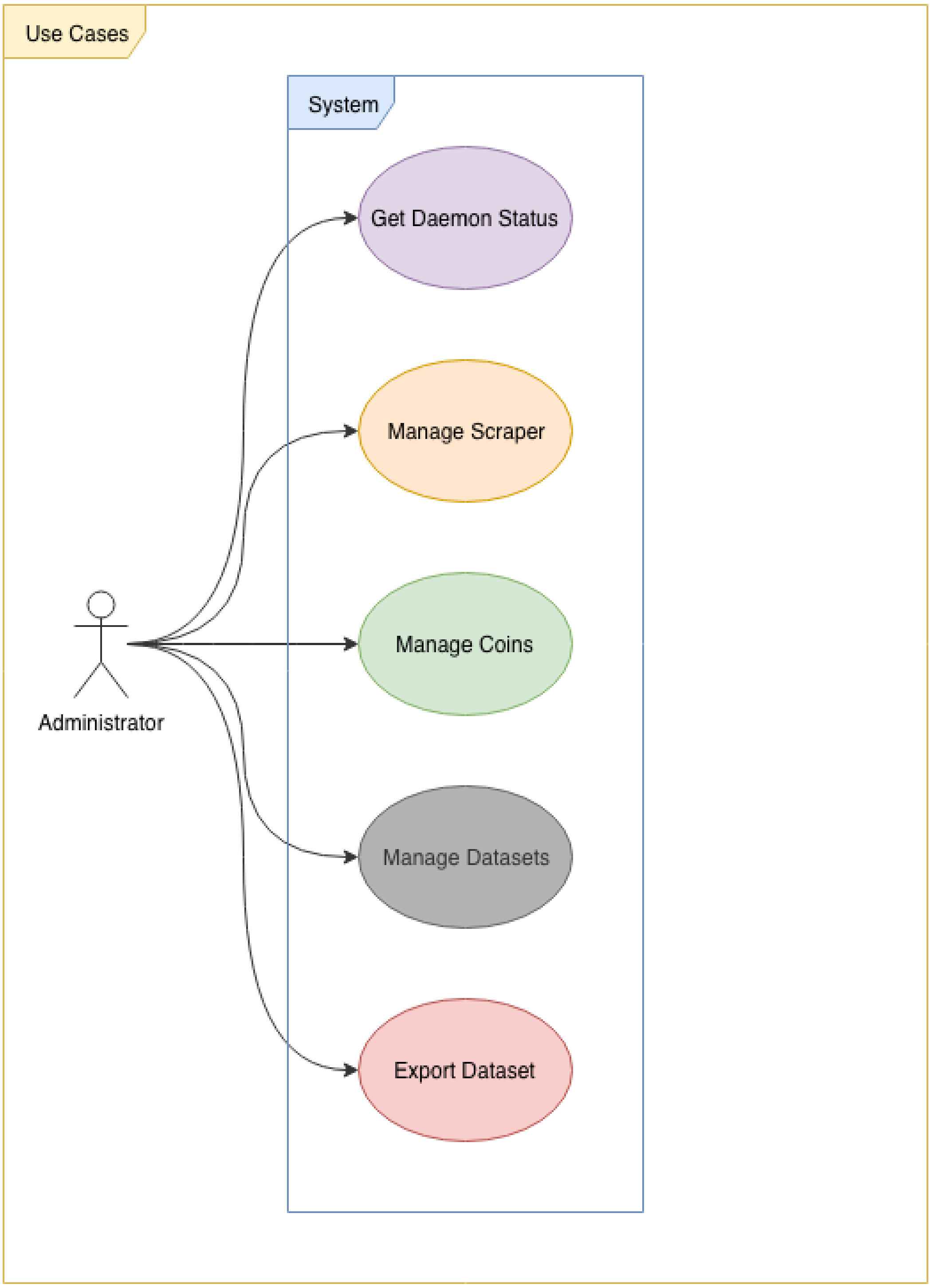
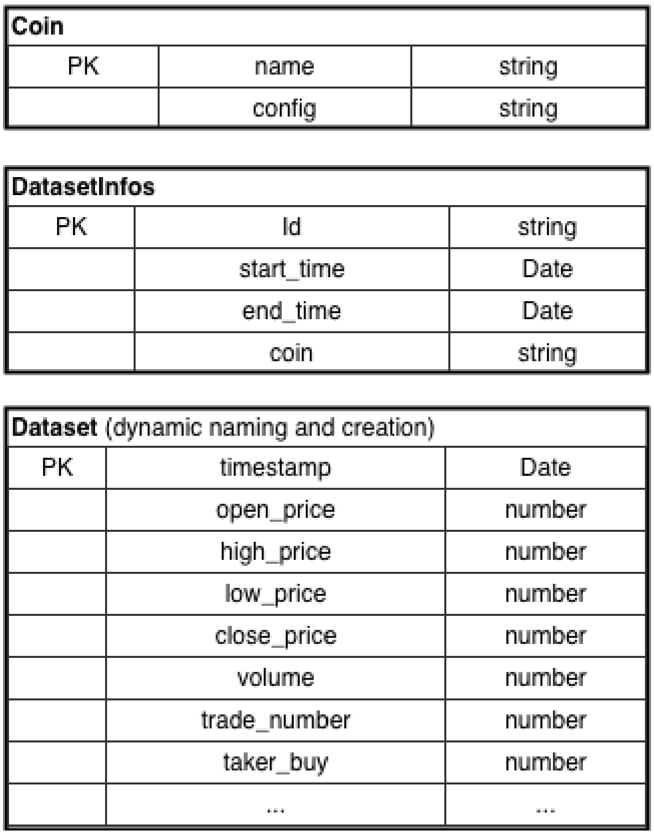


*Figure 2.5.1: Use case view*

## 2.6. Data view

This section was made for Data Specialists and Database Administrators. It describes the architecturally significant persistent elements in the data model.



| **Coin** | | |
|---|---|---|
| PK | name | string |
| | config | string |

| **DatasetInfos** | | |
|---|---|---|
| PK | Id | string |
| | start_time | Date |
| | end_time | Date |
| | coin | string |

| **Dataset** (dynamic naming and creation) | | |
|---|---|---|
| PK | timestamp | Date |
| | open_price | number |
| | high_price | number |
| | low_price | number |
| | close_price | number |
| | volume | number |
| | trade_number | number |
| | taker_buy | number |
| | ... | ... |

Figure 2.6.1: Data View Diagram

# 3. Architectural requirements: goals and constraints

Requirements are already described in SRS. This section describes key requirements and constraints that have a significant impact on the architecture.

## 3.1. Functional requirements

Refer to Use Cases or Use Case scenarios which are relevant with respect to the software architecture. The Use Cases referred to should contain central functionality, many architectural elements or specific delicate parts of the architecture.

| **Source** | **Name** | **Architectural relevance** | **Addressed in:** |
|---|---|---|---|
| UC2 | Manage Scraper | Ability to start or stop the daemon impacts how the main class is going to behave | 2.1.3, 2.2 |

# 4. Quality

A description of how the software architecture contributes to the quality attributes of the system as described in the ISO-9126 (I) standard.

| | |
|---|---|
| Scalabity | none |
| Reliability, Availability | Scraper was made to be fault tolerant, and should behave properly even if a third-party actor is not responding correctly. |
| Portability | none |
| Security | The Daemon can only be controlled from a local CLI instance. The Daemon will only listen for incoming messages on the local machine. |