Graph-based Provenance Metadata Database Documentation

Hochschule Karlsruhe
University of
Applied Sciences

Summer semester 2024

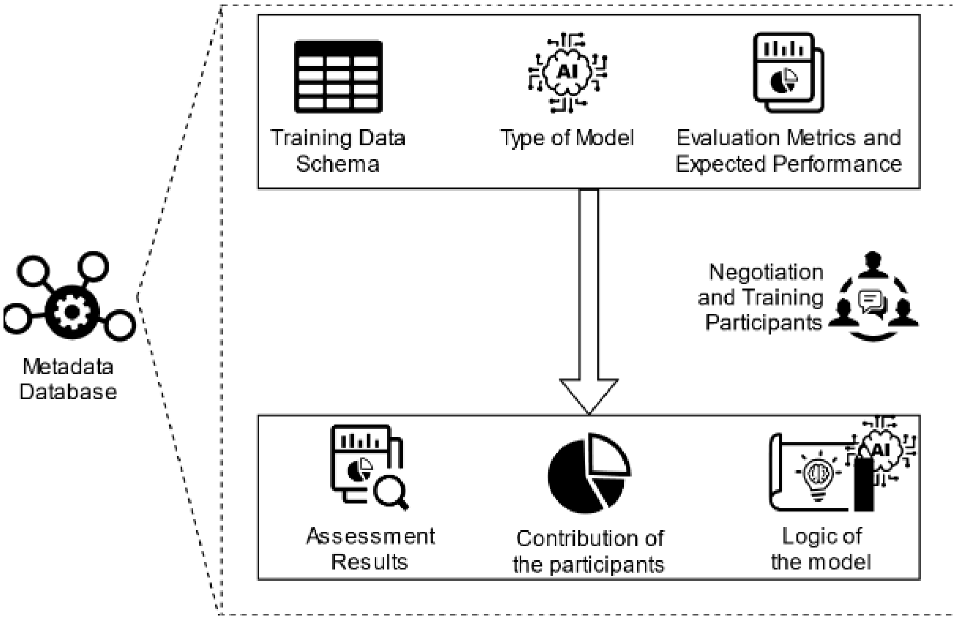
Developed by Philip Mader

Supervised by José Antonio Peregrina Pérez & Prof. Dr. rer. nat. Christian Zirpins

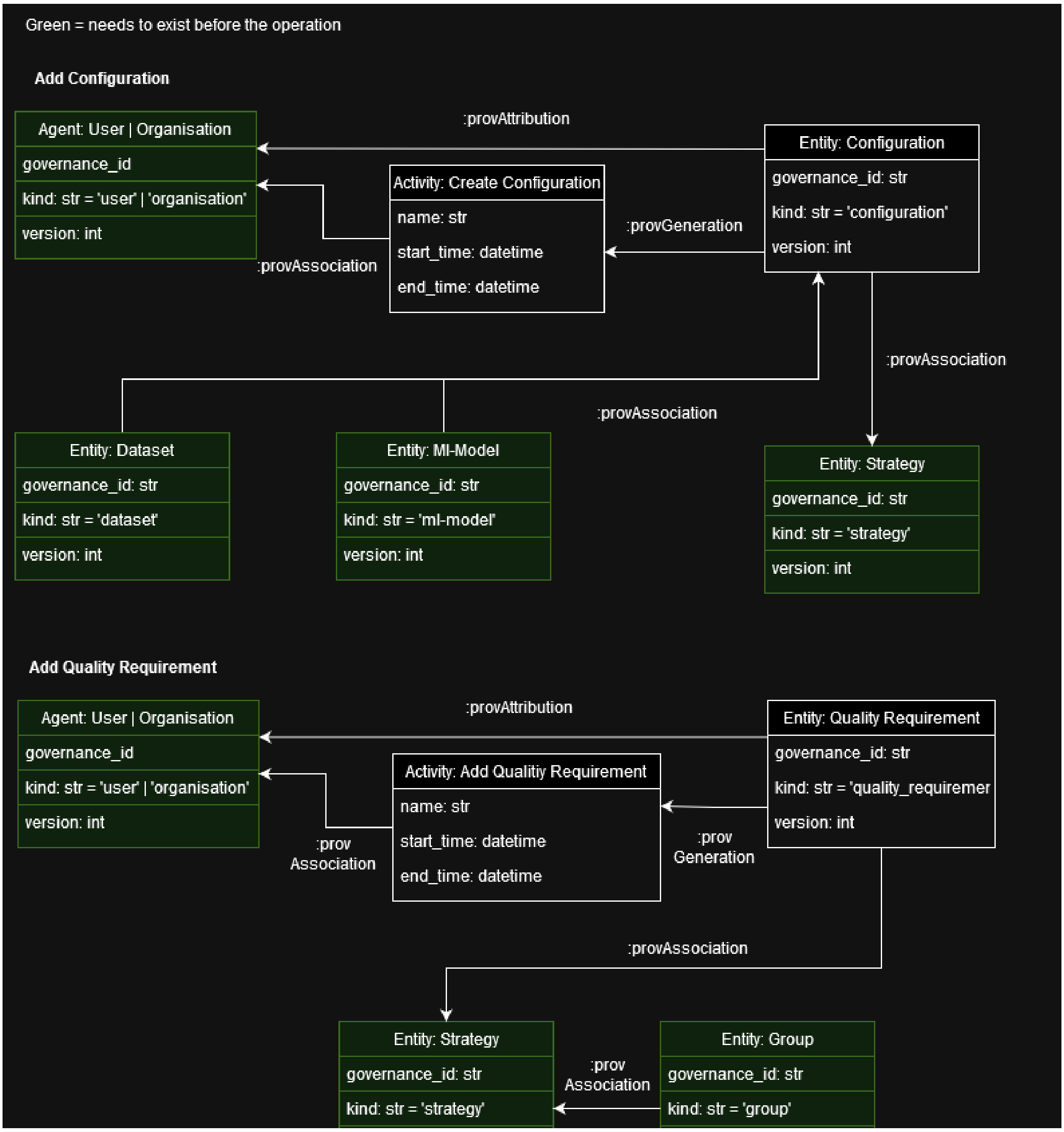
Table of Contents

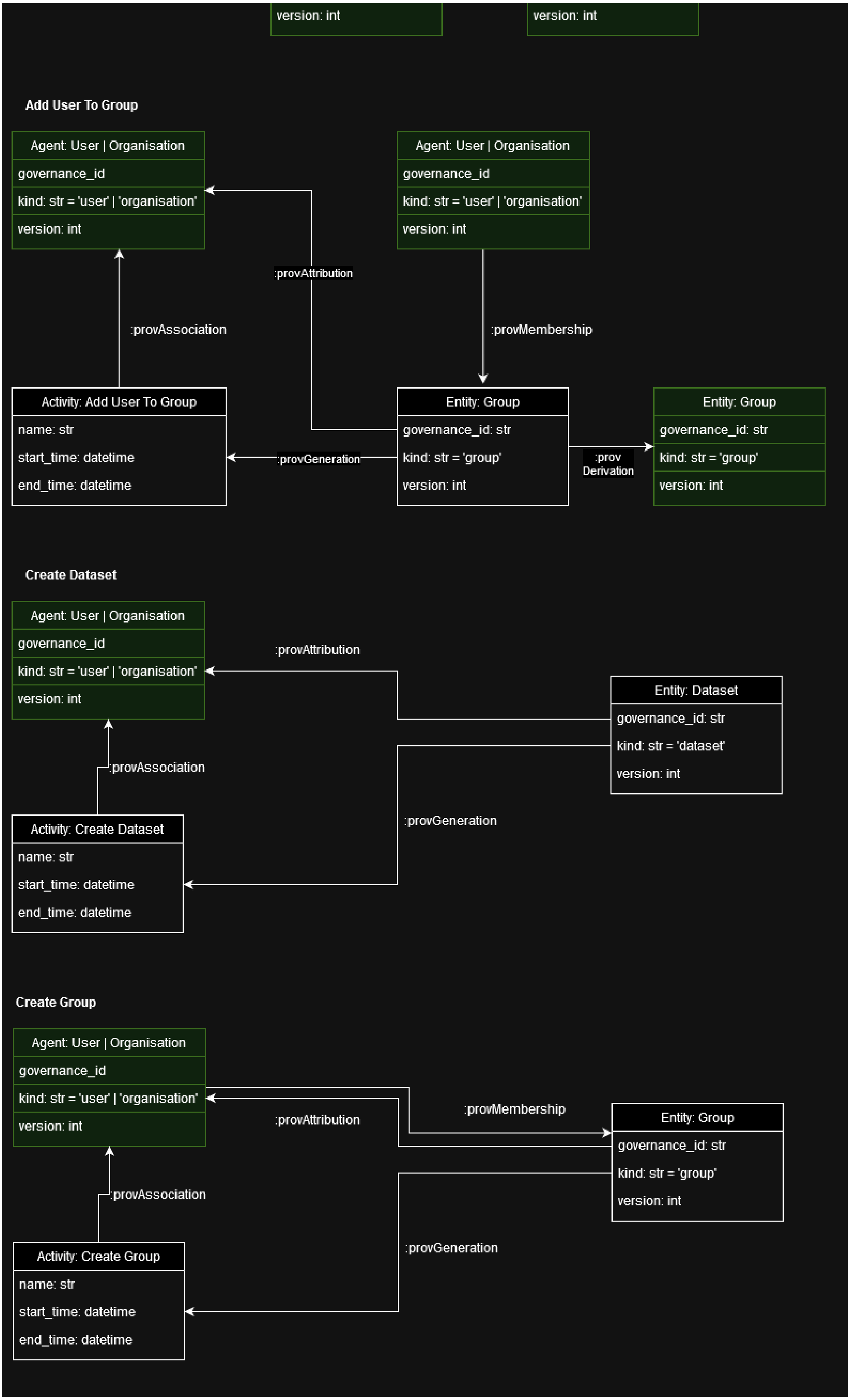
Introduction

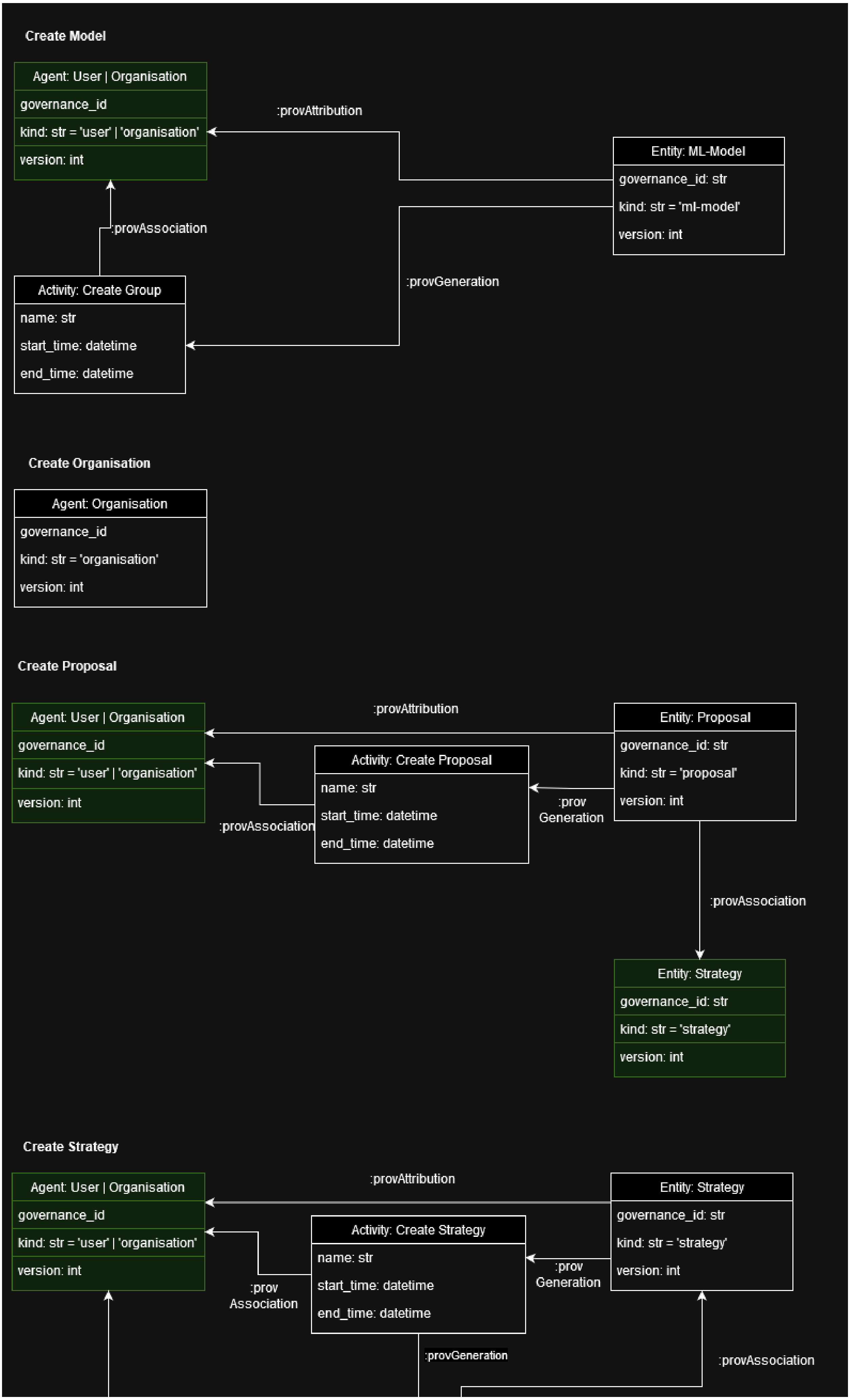
The goal of this project is to manage provenance metadata within the Aura project. In this case provenance metadata refers to a record of operations taken by both data providers and engineers in the `Data Governance Cockpit`. This data can then be used to create a link between the operations and evaluations of the model created by the federated learning process. This way if, for example, the evaluation results worsen significantly there is a database containing a record of all actions that can be queried in convenient ways to find the cause of the bad results.

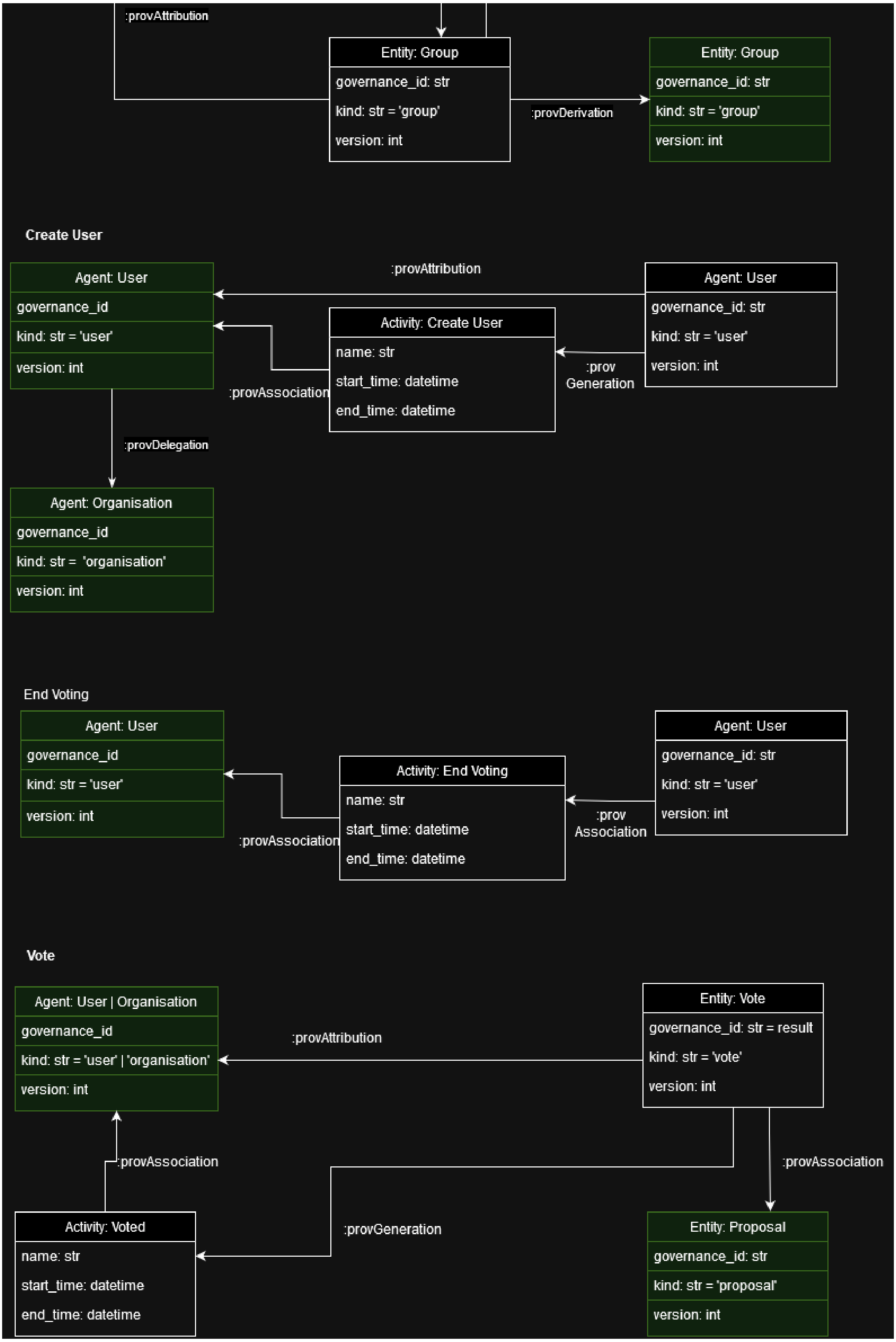

Training Data Schema
Type of Model
Evaluation Metrics and Expected Performance
Negotiation and Training Participants
Metadata Database
Assessment Results
Contribution of the participants
Logic of the model

To accomplish this the software within this package generates provenance metadata from the operations in the `Data Governance Cockpit` and saves it in an instance of Neo4j. Specifically the newly created database entries for each write operation are as follows:




version: int
version: int
Add User To Group
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
:provAttribution
:provAssociation
:provMembership
Activity: Add User To Group
name: str
start_time: datetime
end_time: datetime
:provGeneration
Entity: Group
governance_id: str
kind: str = 'group'
version: int
:prov Derivation
Entity: Group
governance_id: str
kind: str = 'group'
version: int
Create Dataset
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
:provAttribution
Entity: Dataset
governance_id: str
kind: str = 'dataset'
version: int
:provAssociation
Activity: Create Dataset
name: str
start_time: datetime
end_time: datetime
:provGeneration
Create Group
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
:provMembership
:provAttribution
Entity: Group
governance_id: str
kind: str = 'group'
version: int
:provAssociation
Activity: Create Group
name: str
start_time: datetime
end_time: datetime
:provGeneration


Create Model
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
:provAttribution
Entity: ML-Model
governance_id: str
kind: str = 'ml-model'
version: int
:provAssociation
:provGeneration
Activity: Create Group
name: str
start_time: datetime
end_time: datetime
Create Organisation
Agent: Organisation
governance_id
kind: str = 'organisation'
version: int
Create Proposal
:provAttribution
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
Activity: Create Proposal
name: str
start_time: datetime
end_time: datetime
Entity: Proposal
governance_id: str
kind: str = 'proposal'
version: int
:provAssociation
:prov Generation
:provAssociation
Entity: Strategy
governance_id: str
kind: str = 'strategy'
version: int
Create Strategy
:provAttribution
Agent: User | Organisation
governance_id
kind: str = 'user' | 'organisation'
version: int
Activity: Create Strategy
name: str
start_time: datetime
end_time: datetime
Entity: Strategy
governance_id: str
kind: str = 'strategy'
version: int
:prov Association
:prov Generation
:provGeneration
:provAssociation

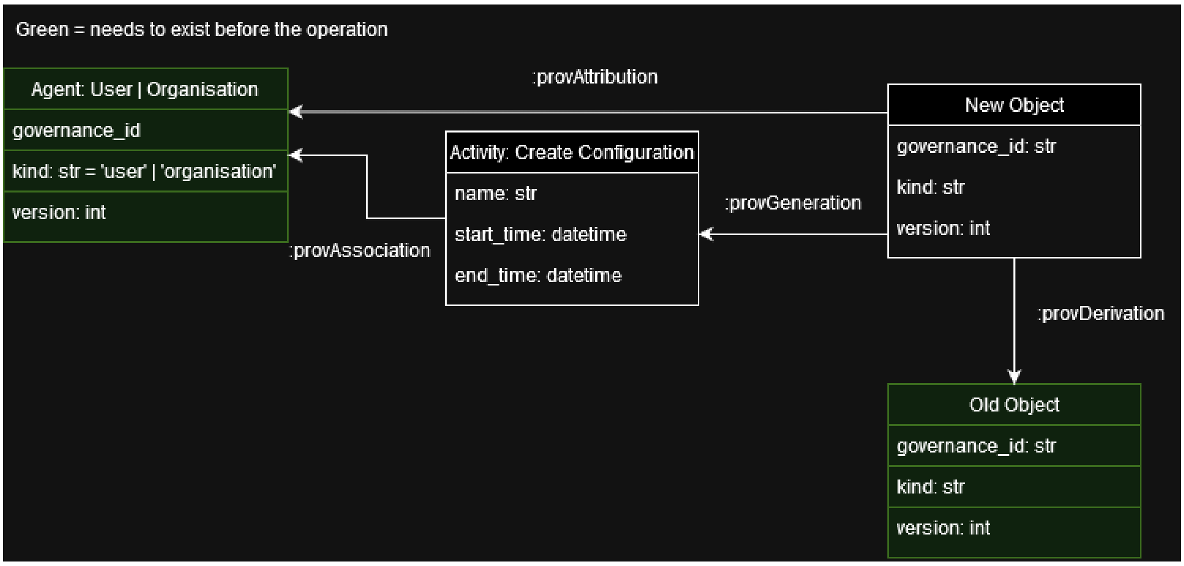


The entries in the database are structured according to PROV-O and contain the following data:

AGENT	ENTITY	ACTIVITY

	AGENT	ENTITY	ACTIVITY
Objects	Organisation, User	Group, ML-Model, Dataset, Quality-Requirements, Proposals, Votes	N/A (Part of PROV-O. Does not directly correlate with any object)
Attributes	Governance-ID, Kind, Version, Timestamp	Governance-ID, Kind, Version, Timestamp	Name, Affected_objects Start-time, End-time

Additionally, when ann object is updated the following data will be created:



To accomplish this the metadata package contains three main components. The `metadata_middleware`, `middleware_api` and `neo4j_connection`. The middleware generates the metadata and uses `neo4j_connection` to write it into the database. The api uses `neo4j_connection` to query the database and return the result. More information can be found in the respective sections.
The package is organized in the following way:

```
ProvenanceMetadataNeo4J/
`-- metadata/
    |-- api/
    |   |-- models/
    |   |   |-- activity_model.py, agent_model.py, entity_models.py
    |   |   `-- grouped _activity_model.py, num_of_actions_model.py
    |   |-- routers/
    |   |   |-- action_router.py, config_router.py, dataset_router.py, dev_router.py
    |   |   |-- group_router.py, model_router.py, organisation_router.py
    |   |   `-- proposal_router.py, strategy_router.py, user_router.py
    |   |-- database_connection.py
    |   `-- metadata_api.py
    |-- dbmanager/
    |   |-- exceptions/
    |   |   |-- group_does_not_exist.py, object_does_not_exist.py, no_actions.py
    |   |   `-- relationship_does_not_exist.py, strategy_does_not_exist.py,
    |   |   `-- user_does_not_exist.py, version_does_not_exist.py
    |   |-- queries/
    |   |   `-- create.py, delete.py, interactions.py, retrieve.py, update.py
    |   `-- neo4j_connection.py
    |-- middleware/
    |   |-- operations/
    |   |   |-- add_config.py, add_quality_requirement.py, add_user_to_group.py
    |   |   |-- create_dataset.py, create_group.py, create_ml_model.py
    |   |   |-- create_organisation.py, create_proposal.py, create_strategy.py
    |   |   |-- create_user.py, delete_config.py, delete_dataset.py, delete_group.py
    |   |   |-- delete_ml_model.py, delete_organisation.py, delete_proposal.py,
    |   |   |-- delete_qr.py delete_strategy.py, delete_user.py, delete_vote.py, end_voting.py
    |   |   |-- middleware_operation.py, update_config.py, update_dataset.py
    |   |   `-- update_group.py, update_model.py, update_organisation.py, update_qr.py
    |   |   `-- update_strategy.py,update_user.py vote.py
    |   |-- metadata_middleware.py
    |   `-- middleware_operations_manager.py
    |-- const.py
    `-- token.py
```

Return to ToC

Usage

The tools provided in this package are included in the `Data Governance Full Stack`. Therefore, installing them works the same way as installing the Full Stack.

1. Clone the `Data_Governance_Full_Stack` repository including its submodules:

```
git clone https://github.com/JsAntoPe/Data_Governance_Full_Stack.git
cd Data_Governance_Full_Stack

git submodule init
git submodule update
```

2. Build and start the docker-compose-container:

```
docker compose build --no-cache
docker compose up -d
```

Return to ToC

Requirements

In order to use the tools provided in this package the following packages need to be installed:

- `fastapi`
- `httpx`
- `neo4j`
- `pyjwt`
- `uvicorn`

Which can be done like this:

```
pip install fastapi
pip install httpx
pip install neo4j
pip install pyjwt
pip install uvicorn
```

The documentation for these libraries can be found here:

- fastapi docs
- httpx docs
- neo4j docs
- pyjwt docs
- uvicorn docs

Please note that docker automatically creates the virtual environment including all libraries when building the Full-Stack-Container. Therefore, when using docker it is not necessary to manually install the libraries.

Return to ToC

Important information for developers

MongoDB-ids vs governance_ids

Within this project there are two kinds of ids for objects managed by the `Data Governance Cockpit`.

1. MongoDB-ids which are unique to each entry and therefore two versions of the same object have different ids.
2. governance_ids which are used by keycloak and do stay the same between versions.

The metadata uses governance_ids to identify object and then an additional attribute for versions so they to can be differentiated. However, there are multiple scenarios within the cockpit where MongoDB-ids are used and the `metadata-middleware` must resolve them. To do this there is a functions in the `middleware_operation` class which can be used like this:

```
gov_id: str = await self.get_governance_id('MongoDB-id', 'kind_of_object')
```

Note, that, because all middleware operations are subclasses of `MiddlewareOperation`, this function can be used in every middleware operation.

Return to ToC

OAuth2 tokens for keycloak

The previous section talked about the `get_governance_id` function. It should be added that the way this functions retrieves governance_ids is by making a REST-call to the `Data Governance Cockpit`. In order to make this call a OAuth2 token is required. This token is acquired in the `get_token` function in `token.py`. Additionally, the constants such as grant_type, password etc. can be found in `const.py`. It is advised that before changing this functions one should familiarize themselves with the `Keycloak Admin REST API`. The documentation of which can be found here.

Return to ToC

Neo4j and metadata_api instances

The Neo4j instance uses the port 7687 for the Bolt protocol and the port 7474 to access the browser GUI. To access the database the following credentials are required:

- user: neo4j
- password: password

The metadata_api uses the port 5001.

Return to ToC

Metadata Middleware

Description

In order to store the providence metadata there needs to be some connections between the `Data Governance Cockpit` and the database. This connection is here provided by a custom middleware for the REST-API that runs in the governance cockpit.
A `middleware` is a function that gets called every time a `FastAPI` receives a request. This function is also given access to the Request- and Response-objects of the API-call.
This middleware uses the information it is given to generate the provenance metadata and then store it in the database.

The middleware generates the metadata in the following way:

- Agent: governance_id of the user that makes the API-Call
- Activity: derived from the combination of method and path of the request
- Entity: Either taken from Request- or Response-body

Return to ToC

Requirements

The middleware uses the following packages:

- `fastapi`
- `httpx`
- `neo4j`
- `pyjwt`

Which can be installed like this:

```
pip install fastapi
pip install httpx
pip install neo4j
pip install pyjwt
```

Return to ToC

Usage

In order to use the middleware it must be added to a FastAPI instance.
This can be done like this:

```
from fastapi import FastAPI
from ProvenanceMetadataNeo4J.metadata import MetadataMiddleware

app = FastAPI()
app.add_middleware(MetadataMiddleware)
```

Return to ToC

Operations

The middleware supports the following operations:

Operation	Method	Path	Description
`add_config`	POST	/groups/{group_id}/strategies/{strategy_id}/configurations	adds a configurations to the given strategy
`add_user_to_group`	POST	/groups/{group_id}/add/{user_id}	adds a user to a group
`create_dataset`	POST	/datasets	creates a dataset
`create_group`	POST	/groups	creates a group
`create_ml_model`	POST	/ml-models	creates a ml-model
`create_organisation`	POST	/organisations	creates an organisation
`create_proposal`	POST	/proposals	creates a proposal
`create_strategy`	POST	/strategies	creates a strategy
`create_user`	POST	/users	creates a user
`delete_config`	DELETE	/groups/{group_id}/strategies/{strategy_id}/configurations/{configuration_id}	deletes a configuration
`delete_dataset`	DELETE	/datasets	deletes a dataset
`delete_group`	DELETE	/groups	deletes a group

Operation	Method	Path	Description
`delete_ml_model`	DELETE	/ml-models	deletes a ml-model
`delete_organisation`	DELETE	/organisations	deletes an organisation
`delete_proposal`	DELETE	/proposals/{proposal_id}	deletes a proposal
`delete_strategy`	DELETE	/groups/{group_id}/strategies/{strategy_id}	deletes a strategy
`delete_qr`	DELETE	/strategies/{strategy_id}/quality_requirements/{quality_requirement_id}	deletes a quality_requirement
`delete_user`	DELETE	/users	deletes a users
`delete_vote`	DELETE	/proposals/{proposal_id}/votes/{member_id}	deletes a vote
`end_voting`	GET	/proposals/{strategy_id}/count_votes	ends voting
`update_config`	PUT	/groups/{group_id}/strategies/{strategy_id}/configurations/{config_id}/mlmodel /groups/{group_id}/strategies/{strategy_id}/configurations/{config_id}/dataset	updates a configuration
`update_dataset`	PUT	/datasets/{dataset_id}/features	updates a dataset
`update_group`	PUT	/groups/governance_id/{group_id}	updates a group
`update_model`	PUT	/ml-models/{model_id}	updates a model
`update_organisation`	PUT	/organisations/{user_id}	updates an organisation
`update_qr`	PUT	/strategies/{strategy_id}	updates a quality requirement
`update_strategy`	PUT	/strategies/{strategy_id}/quality_requirements/{quality_requirement_id}	updates a strategy
`update_user`	PUT	/users/{user_id}	updates a user
`reset`	POST	/dev/reset	deletes all database entries
`vote`	POST	/proposals/{proposal_id}/votes	logs a vote for a proposal

Return to ToC

Expansion

The functionality of the middleware ca be expanded in the following way:

1. Create a file called [name_of_the_operation].py in `metadata/middleware/operations`
2. Implement the Operation. The following template can be used for this

```
from typing import Any

from ...dbmanager.neo4j_connection import Neo4JConnection
from .middleware_operation import MiddlewareOperation


# TODO Add the name of your Operation
class OPERATION(MiddlewareOperation):
    """
    OPERATION
    """
    # TODO add regex for method@path here
    regex: str = ''

    def __init__(self, db: Neo4JConnection):
        """
        Constructor for Middleware Operation
        :param db: neo4j database connection
        """
        super().__init__(self.regex, db)

    async def execute(self,
                      operation: str,
                      user_responsible: str,
                      response_json: Any
                      ) -> None:
        # Todo add your code here
        pass
```

3. Add the newly created operations to the `middleware_operations_manager`. Specifically add it to the `self.operations` list.

Return to ToC

Neo4JConnection

Description

The class `Neo4JConnection` manages access to the Neo4j-database and store the provenance metadata in the database. It contains implementations for database operations that are used by either the middleware and the API.

Return to ToC

Requirements

Neo4JConnection uses the following packages:

- `httpx`
- `neo4j`

Which can be installed like this:

```
pip install httpx
pip install neo4j
```

Return to ToC

Usage

The `Neo4JConnection` class can be used like this

1. Import `Neo4JConnection`
2. Connect to the database via the constructor
3. Use the implemented operations
4. Close the database connection

One way to use the `Neo4JConnection` class, would be like this:

```
from metadata.dbmanager.neo4j_connection import Neo4JConnection

db: Neo4JConnectionMW = Neo4JConnection('neo4j://127.0.0.1:7687')
db.reset()
db.create_user('example_user')
print(db.get_all_users())
db.close()
```

Return to ToC

Operations

`Neo4JConnection` implements the following public basic functions:

Signature	Parameters	Return-value	Description
`__init__`	uri: str authorization: tuple = None database: str = None	None	constructor for Neo4JConnection, creates connection to the database
`close`	none	bool: success	closes the connection to the database
`reconnect`	none	none	reconnects to the database

`Neo4JConnection` implements the following public functions to query the database:

Signature	Parameters	Return-value	Description
`get_actions`	user_id: str start_time: datetime = None end_time: datetime = None fetch_user_info: bool = True	list[dict]: actions	returns all actions taken by a user if one is given. if a start_time is given then it will return all actions taken since if an end_time is given it will return all actions taken until if both a start and end_time are given all actions in that interval will be returned or raises a `UserDoesNotExistException` if user_id is not an existing user
`get_actions_for_object`	gov_id: str	list[dict]: actions	returns all activities for the given object
`get_all_datasets`	none	list[dict]: datasets	returns all datasets as a list
`get_all_groups`	none	list[dict]: groups	returns a list of all groups
`get_all_models`	none	list[dict]: models	returns all models as a list
`get_all_organisations`	none	list[dict]: organisations	returns a list of all organisations
`get_all_proposals`	none	list[dict]: proposal	returns a list of all proposal
`get_all_strategies`	none	list[dict]: strategies	returns all strategies for a given group as a list
`get_all_users`	None	list[dict]: users	returns a list of all users
`get_configurations`	strategy_id: str	list[dict]: strategies	returns all configurations for a given strategy as a list or raises a `StrategyDoesNotExistException` if strategy_id does not exist
`get_members_of_group`	group_id: str	list[dict]: members	returns the members of the given group or raises a `GroupDoesNotExistException` if the group does not exist.
`get_more_than_actions`	num: int, start_time: datetime = None, end_time: datetime = None	list[dict]: more_than	returns a map of user and number of actions taken within the given interval for all user that have taken more than num actions

Signature	Parameters	Return-value	Description
`get_nodes_by_relationship`	gov_id: str, relationship: str	list[dict]: nodes	returns all objects associated with the given object via the given relationship or raises a `ObjectDoesNotExistException` if the gov_id is invalid or raises a `RelationshipDoesNotExistException` if the relationship is invalid
`get_num_of_actions`	start_time: datetime = None, end_time: datetime = None	list[dict]: num_of_actions	returns a map of user and number of actions taken within the given interval.
`get_user`	governance_id: str version: int = None	list[dict]: user	returns the current version of a user or the version that was specified or raises a `UserDoesNotExistException` if user_id is not an existing user or raises a `VersionDoesNotExistException` if version of user does not exist
`get_qr_for_strategy`	strategy_id: str	list[dict]: qrs	Returns all qrs for the strategy as a lst

`Neo4JConnection` implements the following public functions to manipulate the database:

Signature	Parameters	Return-value	Description
`add_configuration`	config_id: str strategy_id: str group_id: str model_id:str dataset_id: str user_id: str	bool: success	creates a database entry for a configuration
`add_quality_requirement`	qr_id: str strategy_id: str group_id: str user_id: str	bool: success	adds a quality requirement to the given strategy
`add_user_to_group`	group_id: str user_res: str user_add: str	bool: success	adds the user with the id user_add to he group with the id user_add
`create_dataset`	dataset_id: str user_id: str	bool: success	creates a database entry for a dataset
`create_group`	group_id: str user_id: str	bool: success	creates a group
`create_model`	model_id: str user_id: str	bool: success	creates a database entry for a model
`create_organisation`	governance_id: str	bool: success	creates an entry for an organisation
`create_proposal`	proposal_id: str strategy_id: str user_id: str	bool: success	creates a new proposal
`create_strategy`	strategy_id: str group_id: str user_id: str	bool: success	creates a database entry for a strategy
`create_user`	governance_id: str	bool: success	creates an entry for a user
`delete_entry`	gov_id: str, user_res: str, type: str	bool: success	deletes an entry

Signature	Parameters	Return-value	Description
`delete_vote`	prop_id: str, voter_id, user_res: str	bool: success	deletes a vote
`end_voting`	proposal_id: str strategy_id: str user_res: str, type: str	bool: success	ends voting
`update_config`	type: str, config_id: str, model_id: str, dataset_id: str, user_res: str	bool: success	updates either the model or the dataset of a config
`vote`	proposal_id: str vote: str user_id: str	bool: success	logs a vote

Please note that when an entry is created more than just the object itself is added to the database. The entries required by PROV-O will also be created.
They include, but are not limited to:

- *An activity that describes the action*
- *An association from the activity to user_res*
- *An attribution from the created entity to user_res*

A full explanation of the organisation of the data can be found in the introduction.

Return to ToC

Expansion

The functionality of `Neo4JConnection` can be expanded via new functions. In these functions the `self._driver` or `self._session` object can be used to interact with the database. Neo4j uses the query-language 'Cypher', the documentation of which can be found here.
Queries can be executed in one of the following ways:

```
self._driver.execute_query('CREATE (n: Node {name:'name'})')
self._session.run('CREATE (n: Node {name:'name'})')
```

Queries can include placeholders which can be taken from the parameters passed to `execute_query` or `run`. In which case the call may look like this:

```
QUERY: str = 'CREATE (n: Node {name:$name})'  # placeholders are denoted with a $
self._driver.execute_query(QUERY, name='test_node')
self._session.run(QUERY, name='test_node')
```

If the executed query contains a `RETURN` the result of that query can be accessed like this:

```
QUERY: str = 'MATCH (n) RETURN n'  # returns all database entries
rec: list[Record] = self._driver.execute_query(QUERY).records
rec: list[dict] = self._session.run(QUERY).data()
```

In general `execute_query` and `run` return objects of the type `Result`. Besides the record it also contains other information that may be useful e.g. counters for nodes created or the time elapsed while processing the query.
The full documentation for the execution of queries can be found here.

In short this is how to expand Neo4JConnection:

1. Create a new function that takes all the arguments needed
2. Write Cypher queries that executes the database operations that are to be performed
3. Add the query as constant to the appropriate file in `metadata/dbmanager/queries`
4. Implement the functions including the calls to the database
5. If the added functionality returns something deal with the result

Return to ToC

Metadata-API

Description

The Metadata-API provides means to query the metadata database. It uses a `FastAPI` to provide a REST-Interface.

Return to ToC

Requirements

The API uses the following packages:

- `fastapi`
- `httpx`
- `neo4j`
- `uvicorn`

Which can be installed like this:

```
pip install fastapi
pip install httpx
pip install neo4j
pip install uvicorn
```

Return to ToC

Usage

Run the `run_metadata_api.py` file, which can be done like this. Remember to change all the URL to their local variants in `const.py`:

```
py run_metadata_api.py
```

Return to ToC

Operations

The Metadata-API supports the following operations:

Method	Path	Description
GET	/actions	returns a JSONResponse containing all actions If a start_time is given then it will return all actions taken since If an end_time is given it will return all actions taken until If both a start and end_time are given all actions in that interval will be returned :param start_time: start_time of interval :param end_time: end_time of interval
GET	/actions/grouped_by_user	Same as /actions but groups result by user
GET	/actions/num	returns a JSONResponse containing a map of user and number of actions taken within the given interval
GET	/actions/more	returns a JSONResponse containing a map of user and number of actions taken within the given interval for all user that have taken more than num actions

Method	Path	Description
GET	/actions/related_to/gov_id	returns a JSONResponse containing all actions related to the object
GET	/configurations	returns a JSONResponse containing all configurations
GET	/datasets	returns a JSONResponse containing all datasets
GET	/groups	returns a JSONResponse containing all groups
GET	/groups/{group_id}/members	returns a JSONResponse containing all members of the given group
GET	/ml-models	returns a JSONResponse containing all ml-models
GET	/organisations	returns a JSONResponse containing all organisations
GET	/proposals	returns a JSONResponse containing all proposals
GET	/strategies	returns a JSONResponse containing all strategies
GET	/strategies/{id}/qr	returns a JSONResponse containing all quality requirements for the given strategy
GET	/users	returns a JSONResponse containing all users
GET	/users/{relationship}/to/{user_id}	returns a JSONResponse containing all agents connected to the object with the given user_id by the given relationship
GET	/users/{user_id}/actions?start_time&end_time	returns a JSONResponse containing all actions taken by the given user if no start_time and no end_time is specified returns a JSONResponse containing all actions taken by the given user since the start_time if a start_time is given returns a JSONResponse containing all actions taken by the given users before the end_time if an end_time is specified returns a JSONResponse containing all actions
GET	/users/{user_id}?version	returns a JSONResponse containing the given version of a user or the current version if version is not specified

Return to ToC

Expansion

The API can be expanded by adding more endpoints.
The operations of the API are divided between `APIRouters` for each path. For example there is a router that handles request for a path beginning with `/users`. If the operation that is to be added fits into an existing router add the required code into that router. To do this create a new function that contains the functionality that is annotated with the method, path, response_model and responses of the request. This could look like this:

```
@router.get('/{gov_id}', response_model=AgentModel,
            responses={
                status.HTTP_200_OK: {
                    'description': 'Returns a user as a JSON.'
                },
                status.HTTP_404_NOT_FOUND: {
                    'description': 'The requested user or version of the user does not exist.'
                }
            })
async def get_user(gov_id: str, version: int | None = None) -> AgentModel:
    """
    Returns a users
    """
    return AgentModel(**db.get_user(gov_id, version)[0])
```

Should the new functionality not fit into an existing router then a new router needs to be created in `metadata/api/routers`. The following template can be used for this:

```python
from fastapi import APIRouter
from starlette import status

from ..database_connection import db_connection as db

# TODO add prefix and tags
router: APIRouter = APIRouter(prefix='', tags=[''])
```

Then the new endpoint can be added to the new router. Finally, the router needs to be added to the API. This can be done like this:

```python
from fastapi import FastAPI
from .routers import action_router as actions

app = FastAPI()
app.include_router(users.router)
```

The full documentation for `APIRouter` can be found here.

Please remember that this API is only for querying the database not for manipulating it. Therefore, all existing endpoint receive GET-requests. If at all possible any new functionality should also be accessed via GET-requests.

Return to ToC