aio-openapi

Release 3.0.1

Quantmind

Jul 27, 2022

CONTENTS

Asynchronous web middleware for aiohttp and serving Rest APIs with OpenAPI v 3 specification and with optional PostgreSql database bindings.

Current version is 3.0.1.

CHAPTER

ONE

INSTALLATION

It requires python 3.8 or above.

```
pip install aio-openapi
```

CHAPTER

TWO

DEVELOPMENT

Clone the repository and install dependencies (via poetry):

```
make install
```

To run tests

```
poetry run pytest --cov
```

CHAPTER

THREE

FEATURES

- Asynchronous web routes with aiohttp
- Data validation, serialization and unserialization with python *dataclasses*
- OpenApi v 3 auto documentation
- SqlAlchemy expression language
- Asynchronous DB interaction with asyncpg
- Migrations with alembic
- SqlAlchemy tables as python dataclasses
- Support click command line interface
- Redoc document rendering (like https://api.metablock.io/v1/docs)
- Optional sentry middleware

CHAPTER

FOUR

CONTENTS

4.1 Tutorial

In this tutorial we guide you through the implementation of a minimal Rest application with a database persistence and open api documentation.

The application has the following modules

```
- main.py
- db.py
- endpoints.py
- models.py
```

The *main.py* is the entrypoint of the application and has the following functions:

```
from aiohttp import web
from openapi.rest import rest


def create_app():
    return rest(setup_app=setup_app, ...)


def setup_app(app: web.Application) -> None:
    ...


if __name__ == "__main__":
    """Run the app"""
    create_app().main()
```

The *setup_app* function setup the aiohttp application with endpoints and middleware. We'll fill the *setup_app* function later on in the tutorial.

4.1.1 Endpoints

Lets add some endpoint in the *endpoints.py* module:

```
from sqlalchemy.sql.expression import null

from openapi.db.path import SqlApiPath
from openapi.spec import op

from .models import (
    Task,
    TaskAdd,
    TaskOrderableQuery,
    TaskPathSchema,
    TaskQuery,
    TaskUpdate,
)

routes = web.RouteTableDef()


@routes.view("/tasks")
class TasksPath(SqlApiPath):
    """
    ---
    summary: Create and query Tasks
    tags:
        - Task
    """

    table = "tasks"

    def filter_done(self, op, value):
        done = self.db_table.c.done
        return done != null() if value else done == null()

    @op(query_schema=TaskOrderableQuery, response_schema=List[Task])
    async def get(self):
        """
        ---
        summary: Retrieve Tasks
        description: Retrieve a list of Tasks
        responses:
            200:
                description: Authenticated tasks
        """
        paginated = await self.get_list()
        return paginated.json_response()

    @op(response_schema=Task, body_schema=TaskAdd)
    async def post(self):
        """
        ---
```

(continues on next page)

(continued from previous page)

```
        summary: Create a Task
        description: Create a new Task
        responses:
            201:
                description: the task was successfully added
            422:
                description: Failed validation
        """
        data = await self.create_one()
        return self.json_response(data, status=201)

    @op(query_schema=TaskQuery)
    async def delete(self):
        """
        ---
        summary: Delete Tasks
        description: Delete a group of Tasks
        responses:
            204:
                description: Tasks successfully deleted
        """
        await self.delete_list(query=dict(self.request.query))
        return web.Response(status=204)


@routes.view("/tasks/{id}")
class TaskPath(SqlApiPath):
    """
    ---
    summary: Create and query tasks
    tags:
        - name: Task
        description: Simple description
        - name: Random
        description: Random description
    """

    table = "tasks"
    path_schema = TaskPathSchema

    @op(response_schema=Task)
    async def get(self):
        """
        ---
        summary: Retrieve a Task
        description: Retrieve a Task by ID
        responses:
            200:
                description: the task
        """
        data = await self.get_one()
        return self.json_response(data)
```

(continues on next page)

(continued from previous page)

```
    @op(response_schema=Task, body_schema=TaskUpdate)
    async def patch(self):
        """
        ---
        summary: Update a Task
        description: Update an existing Task by ID
        responses:
            200:
                description: the updated task
        """
        data = await self.update_one()
        return self.json_response(data)

    @op()
    async def delete(self):
        """
        ---
        summary: Delete a Task
        description: Delete an existing task
        responses:
            204:
                description: Task successfully deleted
        """
        await self.delete_one()
        return web.Response(status=204)
```

4.1.2 Models

The models are dataclasses which implement the validation and documentation logic, these are implemented in the *models.py* module:

```
from dataclasses import dataclass
from datetime import datetime
from decimal import Decimal
from typing import Dict, List, Union

from openapi.data import fields
from openapi.data.db import dataclass_from_table
from openapi.pagination import offsetPagination, searchable

from .db import DB
from .db.tables1 import TaskType


@dataclass
class TaskAdd(
    dataclass_from_table(
        "_TaskAdd", DB.tasks, required=True, default=True, exclude=("id", "done")
    )
):
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def validate(cls, data, errors):
        """here just for coverage"""


Task = dataclass_from_table("Task", DB.tasks)


@dataclass
class TaskQuery(offsetPagination("title", "-title", "severity", "-severity")):
    title: str = fields.str_field(description="Task title")
    done: bool = fields.bool_field(description="done flag")
    type: TaskType = fields.enum_field(TaskType, description="Task type")
    severity: int = fields.integer_field(
        ops=("lt", "le", "gt", "ge", "ne"), description="Task severity"
    )
    story_points: Decimal = fields.decimal_field(description="Story points")


@dataclass
class TaskOrderableQuery(
    TaskQuery,
    searchable("title", "unique_title"),
):
    pass


@dataclass
class TaskUpdate(TaskAdd):
    done: datetime = fields.date_time_field(description="Done timestamp")


@dataclass
class TaskPathSchema:
    id: str = fields.uuid_field(required=True, description="Task ID")
```

4.1.3 Database

The *db.py* module setup the database schema, in this tutorial, a simple table where we store Tasks.

```
import enum
import os

from aiohttp.web import Application

from openapi.db import CrudDB, get_db
import sqlalchemy as sa

from openapi.data import fields
from openapi.db.columns import UUIDColumn
```

(continues on next page)

(continued from previous page)

```
DATASTORE = os.getenv(
    "DATASTORE", "postgresql+asyncpg://postgres:postgres@localhost:5432/openapi"
)


def setup(app: Application) -> CrudDB:
    return setup_tables(get_db(app, DATASTORE))


def setup_tables(db: CrudDB) -> CrudDB:
    sa.Table(
        "tasks",
        db.metadata,
        UUIDColumn("id", make_default=True, doc="Unique ID"),
        sa.Column(
            "title",
            sa.String(64),
            nullable=False,
            info=dict(min_length=3, data_field=title_field),
        ),
        sa.Column("done", sa.DateTime(timezone=True)),
        sa.Column("severity", sa.Integer),
        sa.Column("created_by", sa.String, default="", nullable=False),
        sa.Column("type", sa.Enum(TaskType)),
        sa.Column("unique_title", sa.String, unique=True),
        sa.Column("story_points", sa.Numeric),
        sa.Column("random", sa.String(64)),
        sa.Column(
            "subtitle",
            sa.String(64),
            nullable=False,
            default="",
        ),
    )
    return db


# this global definition is used by the dataclass_from_table function only
DB = setup_tables(CrudDB(DATASTORE))
```

4.1.4 Open API

By default, no openapi tooling is used when creating a rest application. To enable openapi auto-documenation pass the `openapi` entry:

```
from openapi.rest import rest
from openapi.spec import Redoc

 def create_app():
    return rest(
```

(continues on next page)

(continued from previous page)

```
        openapi=dict(
            title="My API",
            description="My Api ...",
            version="1.0.0",
        ),
        redoc=Redoc(),
        setup_app=setup_app
    )
```

The *Redoc* adds a path for serving the HTML version of the openapi specification.

4.1.5 The main module

Finally, we can put things together

```
from aiohttp import web
from openapi.rest import rest
from openapi.middleware import json_error

from . import endpoints, db


def create_app():
    return rest(
        openapi=dict(
            title="My API",
            description="My Api ...",
            version="1.0.0",
        ),
        redoc=Redoc(),
        setup_app=setup_app
    )


def setup_app(app: web.Application) -> None:
    db.setup(app)
    app.middlewares.append(json_error())
    app.router.add_routes(endpoints.routes)


if __name__ == "__main__":
    """Run the app"""
    create_app().main()
```

4.2 Reference

4.2.1 Data

DataView

class openapi.data.view.**DataView**
Utility class for data with a valid *Supported Schema*

cleaned(*schema*, *data*, *, *multiple=False*, *strict=True*, *Error=None*)
Clean data using a given schema

Parameters

- **schema** (Any) – a valid *Supported Schema* or an the name of an attribute in Operation
- **data** (Union[Dict[str, Any], MultiDict]) – data to validate and clean
- **multiple** (bool) – multiple values for a given key are acceptable
- **strict** (bool) – all required attributes in schema must be available
- **Error** (Optional[type]) – optional Exception class

Return type Any

dump(*schema*, *data*)
Dump data using a given a valid *Supported Schema*, if the schema is *None* it returns the same *data* as the input.

Parameters

- **schema** (Any) – a schema or an the name of an attribute in Operation
- **data** (Any) – data to clean and dump

Return type Any

get_schema(*schema=None*)
Get the *Supported Schema*. If not found it raises an exception

Parameters **schema** (Optional[Any]) – a schema or an the name of an attribute in Operation

Return type *TypingInfo*

raise_validation_error(*message=''*, *errors=None*)
Raise an aiohttp.web.HTTPUnprocessableEntity

Return type NoReturn

validation_error(*message=''*, *errors=None*)
Create the validation exception used by *raise_validation_error()*

Return type Exception

TypeInfo

class openapi.utils.**TypingInfo**(*element: Any*, *container: Optional[type] = None*)
Information about a type annotation

container: Optional[type]
Alias for field number 1

element: Any
Alias for field number 0

classmethod get(*value*)
Create a *TypingInfo* from a typing annotation or another typing info

Parameters **value** (Any) – typing annotation

Return type Optional[*TypingInfo*]

property is_complex: bool
True if *element* is either a dataclass or a union

Return type bool

property is_dataclass: bool
True if *element* is a dataclass

Return type bool

property is_none: bool
True if *element* is either a dataclass or a union

Return type bool

property is_union: bool
True if *element* is a union of typing info

Return type bool

4.2.2 Data Fields

openapi.data.fields.**data_field**(*required=False*, *validator=None*, *dump=None*, *format=None*, *description=None*, *items=None*, *post_process=None*, *ops=()*, *hidden=False*, *meta=None*, ***kwargs*)
Extend a dataclass field with the following metadata

Parameters

- **validator** (Optional[Callable[[Field, Any], Any]]) – optional callable which accept field and raw value as inputs and return the validated value
- **required** (bool) – boolean specifying if field is required
- **dump** (Optional[Callable[[Any], Any]]) – optional callable which receive the field value and convert to the desired value to serve in requests
- **format** (Optional[str]) – optional string which represents the JSON schema format
- **description** (Optional[str]) – optional field description
- **items** (Optional[Field]) – field for items of the current field (only used for *List* and *Dict* fields)
- **post_process** (Optional[Callable[[Any], Any]]) – post processor function executed after validation

- **ops** (Tuple) – optional tuple of strings specifying available operations
- **hidden** (bool) – when *True* the field is not added to the Openapi documentation
- **meta** (Optional[Dict[str, Any]]) – optional dictionary with additional metadata

Return type Field

String field

openapi.data.fields.**str_field**(*min_length=0*, *max_length=0*, ***kw*)
A specialized *data_field()* for strings

Parameters

- **min_length** (int) – minim length of string
- **max_length** (int) – maximum length of string

Return type Field

Bool field

openapi.data.fields.**bool_field**(***kw*)
Specialized *data_field()* for bool types

Return type Field

UUID field

openapi.data.fields.**uuid_field**(*format='uuid'*, ***kw*)
A UUID field with validation

Return type Field

Numeric field

openapi.data.fields.**number_field**(*min_value=None*, *max_value=None*, *precision=None*, ***kw*)
A specialized *data_field()* for numeric values

Parameters

- **min_value** (Optional[Number]) – minimum value
- **max_value** (Optional[Number]) – maximum value
- **precision** (Optional[int]) – decimal precision

Return type Field

Integer field

openapi.data.fields.**integer_field**(*min_value=None*, *max_value=None*, ***kw*)
A specialized *data_field()* for integer values

Parameters

- **min_value** (Optional[Number]) – minimum value
- **max_value** (Optional[Number]) – maximum value

Return type Field

Email field

openapi.data.fields.**email_field**(*min_length=0*, *max_length=0*, ***kw*)
A specialized *data_field()* for emails, validation via the *email_validator* third party library

Parameters

- **min_length** (int) – minimum length of email
- **max_length** (int) – maximum length of email

Return type Field

Enum field

openapi.data.fields.**enum_field**(*EnumClass*, ***kw*)
A specialized *data_field()* for enums

Parameters **EnumClass** – enum for validation

Return type Field

Date field

openapi.data.fields.**date_field**(***kw*)
A specialized *data_field()* for dates

Return type Field

Datetime field

openapi.data.fields.**date_time_field**(*timezone=False*, ***kw*)
A specialized *data_field()* for datetimes

Parameters **timezone** – timezone for validation

Return type Field

JSON field

openapi.data.fields.**json_field**(**kw)
A specialized *data_field()* for JSON data

Return type Field

4.2.3 Data Validation

Validate

The entry function to validate input data and return a python representation. The function accept as input a valid type annotation or a *TypingInfo* object.

openapi.data.validate.**validate**(*schema*, *data*, *, *strict=True*, *multiple=False*, *raise_on_errors=False*, *items=None*, *as_schema=False*)
Validate data with a given schema

Parameters

- **schema** (Any) – a typing annotation or a *TypingInfo* object
- **data** (Any) – a data object to validate against the schema
- **strict** (bool) – if *True* validation is strict, i.e. missing required parameters will cause validation to fails
- **multiple** (bool) – allow parameters to have multiple values
- **raise_on_errors** (bool) – when *True* failure of validation will result in a *ValidationErrors* error, otherwise a ValidatedData object is returned.
- **items** (Optional[Field]) – an optional Field for items in a composite type (*List* or *Dict*)
- **as_schema** (bool) – return the schema object rather than simple data type (dataclass rather than dict for example)

Return type Any

Validate Schema

Same as the *validate()* but returns the validation schema object rather than simple data types (this is mainly different for dataclasses)

openapi.data.validate.**validated_schema**(*schema*, *data*, *, *multiple=False*, *strict=True*)
Validate data with a given schema and return a valid representation of the data as a schema instance

Parameters

- **schema** (Any) – a valid *Supported Schema* or a *TypingInfo* object
- **data** (Any) – a data object to validate against the schema
- **strict** (bool) – if *True* validation is strict, i.e. missing required parameters will cause validation to fails

Return type Any

Dataclass from db table

openapi.data.db.**dataclass_from_table**(*name*, *table*, *, *exclude=None*, *include=None*, *default=False*, *required=False*, *ops=None*)

Create a dataclass from an `sqlalchemy.schema.Table`

Parameters

- **name** (`str`) – dataclass name
- **table** (`Table`) – sqlalchemy table
- **exclude** (`Optional[Sequence[str]]`) – fields to exclude from the dataclass
- **include** (`Optional[Sequence[str]]`) – fields to include in the dataclass
- **default** (`Union[bool, Sequence[str]]`) – use columns defaults in the dataclass
- **required** (`Union[bool, Sequence[str]]`) – set non nullable columns without a default as required fields in the dataclass
- **ops** (`Optional[Dict[str, Sequence[str]]]`) – additional operation for fields

Return type `type`

Dump data

openapi.data.dump.**dump**(*schema*, *data*)

Dump data with a given schema.

Parameters

- **schema** (`Any`) – a valid *Supported Schema*
- **data** (`Any`) – data to dump, if dataclasses are part of the schema, the *dump* metadata function will be used if available (see *data_field()*)

Return type `Any`

4.2.4 Openapi Specification

OpenApiInfo

class openapi.spec.**OpenApiInfo**(*title='Open API'*, *description=''*, *version='0.1.0'*, *termsOfService=''*, *contact=Contact(name='API Support', url='http://www.example.com/support', email='support@example.com')*, *license=License(name='Apache 2.0', url='https://www.apache.org/licenses/LICENSE-2.0.html')*)

Open API Info object

OpenApiSpec

class openapi.spec.**OpenApiSpec**(*info=<factory>*, *default_content_type='application/json'*, *default_responses=<factory>*, *security=<factory>*, *servers=<factory>*, *validate_docs=False*, *allowed_tags=<factory>*, *spec_url='/spec'*, *redoc=None*)

Open API Specification

info: *openapi.spec.spec.OpenApiInfo*
openapi info object, it provides metadata about the API

redoc: Optional[*openapi.spec.redoc.Redoc*] = None
Optional object for rendering the specification as an HTML page via redoc

routes(*request*)
Routes to include in the spec

Return type Iterable

spec_route(*request*)
Return the OpenApi spec

Return type Response

spec_url: str = '/spec'
the path serving the JSON openpi specification

op decorator

Decorator for specifying schemas at route/method level. It is used by both the business logic as well the auto-documentation.

class openapi.spec.**op**(*body_schema=None*, *query_schema=None*, *response_schema=None*, *response=200*)
Decorator for a *ApiPath* view which specifies an operation object in an OpenAPI Path. Parameters are data-classes used for validation and OpenAPI auto documentation.

Redoc

Allow to add redoc redering to your api.

class openapi.spec.**Redoc**(*path='/docs'*, *favicon_url='https://raw.githubusercontent.com/Redocly/redoc/master/demo/favicon.png'*, *redoc_js_url='https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js'*, *font='family=Montserrat:300,400,700|Roboto:300,400,700'*)
A dataclass for redoc rendering

4.2.5 DB

This module provides integration with SqlAlchemy asynchronous engine for postgresql. The connection string supported is of this type only:

```
postgresql+asyncpg://<db_user>:<db_password>@<db_host>:<db_port>/<db_name>
```

Database

class openapi.db.container.**Database**(*dsn=''*, *metadata=None*)

A container for tables in a database and a manager of asynchronous connections to a postgresql database

Parameters

- **dsn** (str) – Data source name used for database connections
- **metadata** (Optional[MetaData]) – sqlalchemy.schema.MetaData containing tables

property dsn: str

Data source name used for database connections

Return type str

property metadata: sqlalchemy.sql.schema.MetaData

The sqlalchemy.schema.MetaData containing tables

Return type MetaData

property engine: sqlalchemy.ext.asyncio.engine.AsyncEngine

The sqlalchemy.ext.asyncio.AsyncEngine creating connection and transactions

Return type AsyncEngine

property sync_engine: sqlalchemy.engine.base.Engine

The sqlalchemy.engine.Engine for synchrouns operations

Return type Engine

__getattr__(*name*)

Retrive a sqlalchemy.schema.Table from metadata tables

Parameters **name** (str) – if this is a valid table name in the tables of *metadata* it returns the table, otherwise it defaults to superclass method

Return type Any

connection()

Context manager for obtaining an asynchronous connection

Return type AsyncConnection

transaction()

Context manager for initializing an asynchronous database transaction

Return type AsyncConnection

ensure_connection(*conn=None*)

Context manager for ensuring we a connection has initialized a database transaction

Return type AsyncConnection

async close()

Close the asynchronous db engine if opened

Return type None

create_all()
Create all tables defined in *metadata*

Return type None

drop_all()
Drop all tables from *metadata* in database

Return type None

drop_all_schemas()
Drop all schema in database

Return type None

CrudDB

Database container with CRUD operations. Used extensively by the *SqlApiPath* routing class.

class openapi.db.dbmodel.**CrudDB**(*dsn=''*, *metadata=None*)
A *Database* with additional methods for CRUD operations

async db_count(*table*, *filters*, *, *conn=None*, *consumer=None*)
Count rows in a table

Parameters

- **table** (Table) – sqlalchemy Table
- **filters** (Dict) – key-value pairs for filtering rows
- **conn** (Optional[AsyncConnection]) – optional db connection
- **consumer** (Optional[Any]) – optional consumer (see *get_query()*)

Return type int

async db_delete(*table*, *filters*, *, *conn=None*, *consumer=None*)
Delete rows from a given table

Parameters

- **table** (Table) – sqlalchemy Table
- **filters** (Dict) – key-value pairs for filtering rows
- **conn** (Optional[AsyncConnection]) – optional db connection
- **consumer** (Optional[Any]) – optional consumer (see *get_query()*)

Return type CursorResult

async db_insert(*table*, *data*, *, *conn=None*)
Perform an insert into a table

Parameters

- **table** (Table) – sqlalchemy Table
- **data** (Union[List[Dict], Dict]) – key-value pairs for columns values
- **conn** (Optional[AsyncConnection]) – optional db connection

Return type CursorResult

async db_select(*table*, *filters*, *, *conn=None*, *consumer=None*)
Select rows from a given table

Parameters

- **table** (Table) – sqlalchemy Table
- **filters** (Dict) – key-value pairs for filtering rows
- **conn** (Optional[AsyncConnection]) – optional db connection
- **consumer** (Optional[Any]) – optional consumer (see *get_query()*)

Return type CursorResult

async db_update(*table*, *filters*, *data*, *, *conn=None*, *consumer=None*)
Perform an update of rows

Parameters

- **table** (Table) – sqlalchemy Table
- **filters** (Dict) – key-value pairs for filtering rows to update
- **data** (Dict) – key-value pairs for updating columns values of selected rows
- **conn** (Optional[AsyncConnection]) – optional db connection
- **consumer** (Optional[Any]) – optional consumer (see *get_query()*)

Return type CursorResult

async db_upsert(*table*, *filters*, *data=None*, *, *conn=None*, *consumer=None*)
Perform an upsert for a single record

Parameters

- **table** (Table) – sqlalchemy Table
- **filters** (Dict) – key-value pairs for filtering rows to update
- **data** (Optional[Dict]) – key-value pairs for updating columns values of selected rows
- **conn** (Optional[AsyncConnection]) – optional db connection
- **consumer** (Optional[Any]) – optional consumer (see *get_query()*)

Return type Row

default_filter_field(*field*, *op*, *value*)
Applies a filter on a field.

Notes on 'ne' op:

Example data: [None, 'john', 'roger'] ne:john would return only roger (i.e. nulls excluded) ne: would return john and roger

Notes on 'search' op:

For some reason, SQLAlchemy uses to_tsquery rather than plainto_tsquery for the match operator

to_tsquery uses operators (&, |, ! etc.) while plainto_tsquery tokenises the input string and uses AND between tokens, hence plainto_tsquery is what we want here

For other database back ends, the behaviour of the match operator is completely different - see: http://docs.sqlalchemy.org/en/rel_1_0/core/sqlelement.html

Parameters

- **field** (Column) – field name
- **op** (str) – ‘eq’, ‘ne’, ‘gt’, ‘lt’, ‘ge’, ‘le’ or ‘search’
- **value** (Any) – comparison value, string or list/tuple

Returns

get_query(*table*, *sql_query*, *, *params=None*, *consumer=None*)
Build an SqlAlchemy query

Parameters

- **table** (Table) – sqlalchemy Table
- **sql_query** (Union[Delete, Select, Update]) – sqlalchemy query type
- **params** (Optional[Dict]) – key-value pairs for the query
- **consumer** (Optional[Any]) – optional consumer for manipulating parameters

Return type Union[Delete, Select, Update]

order_by(*table*, *sql_query*, *order_by*)
Apply ordering to a sql_query

Return type Select

order_by_query(*table*, *sql_query*, *order_by*)
Apply ordering to a sql_query

Return type Select

search_query(*table*, *sql_query*, *search*)
Build an SqlAlchemy query for a search

Parameters

- **table** (Table) – sqlalchemy Table
- **sql_query** (Union[Select, Update]) – sqlalchemy query type
- **search** (Search) – the search dataclass

Return type Union[Select, Update]

get_db

openapi.db.**get_db**(*app*, *store_url=None*)
Create an Open API db handler and set it for use in an aiohttp application

Parameters

- **app** (Application) – aiohttp Application
- **store_url** (Optional[str]) – datastore connection string, if not provided the env variable *DATASTORE* is used instead. If the env variable is not available either the method logs a warning and return *None*

This function 1) adds the database to the aiohttp application at key “db”, 2) add the db command to the command line client (if command is True) and 3) add the close handler on application shutdown

Return type Optional[*CrudDB*]

SingleConnDatabase

A *CrudDB* container for testing database driven Rest APIs.

class openapi.testing.**SingleConnDatabase**(**args*, ***kwargs*)
Useful for speedup testing

connection()
Context manager for obtaining an asynchronous connection

Return type AsyncConnection

transaction()
Context manager for initializing an asynchronous database transaction

Return type AsyncConnection

4.2.6 Routes

ApiPath

class openapi.spec.path.**ApiPath**(*request*)
A *DataView* class for OpenAPI path

path_schema: Optional[type] = None
Optional dataclass for validating path variables

insert_data(*data*, ***, *multiple=False*, *strict=True*, *body_schema='body_schema'*)
Validate data for insertion

if a *path_schema* is given, it validate the request *match_info* against it and add it to the validated data.

Parameters

- **data** (Any) – object to be validated against the body_schema, usually obtained from the request body (JSON)
- **multiple** (bool) – multiple values for a given key are acceptable
- **strict** (bool) – all required attributes in schema must be available
- **body_schema** (Union[str, List[Type], Type]) – the schema to validate against

Return type Dict[str, Any]

get_filters(***, *query=None*, *query_schema='query_schema'*)
Collect a dictionary of filters from the request query string. If *path_schema* is defined, it collects filter data from path variables as well.

Parameters

- **query** (Union[Dict[str, Any], MultiDict, None]) – optional query dictionary (will be overwritten by the request.query)
- **query_schema** (Union[str, List[Type], Type]) – a dataclass or an the name of an attribute in Operation for collecting query filters

Return type Dict[str, Any]

async json_data()
Load JSON data from the request.

Raises **HTTPBadRequest** – when body data is not valid JSON

Return type Any

validation_error(*message=''*, *errors=None*)
Create an aiohttp.web.HTTPUnprocessableEntity

Return type Exception

SqlApiPath

class openapi.db.path.**SqlApiPath**(*request*)
An *ApiPath* backed by an SQL model.

This class provides utility methods for all CRUD operations.

table: str = ''
sql table name

property db: *openapi.db.dbmodel.CrudDB*
Database connection pool

Return type *CrudDB*

property db_table: sqlalchemy.sql.schema.Table
Default database table for this route.

Obtained from the *table* attribute.

Return type Table

async get_list(**, filters=None, query=None, table=None, query_schema='query_schema', dump_schema='response_schema', conn=None*)
Get a list of models

Parameters

- **filters** (Optional[Dict[str, Any]]) – dictionary of filters, if not provided it will be created from the query_schema
- **query** (Optional[Dict[str, Any]]) – additional query parameters, only used when filters is not provided
- **table** (Optional[Table]) – sqlalchemy table, if not provided the default *db_table* is used instead
- **conn** (Optional[AsyncConnection]) – optional db connection

Return type *PaginatedData*

async create_one(**, data=None, table=None, body_schema='body_schema', dump_schema='response_schema', conn=None*)
Create a new database model

Parameters

- **data** (Optional[Dict[str, Any]]) – input data, if not given it loads it via *json_data()*
- **table** (Optional[Table]) – sqlalchemy table, if not given it uses the default *db_table*
- **conn** (Optional[AsyncConnection]) – optional db connection

Return type Dict[str, Any]

async create_list(**, data=None, table=None, body_schema='body_schema', dump_schema='response_schema', conn=None*)
Create multiple models

async get_one(*, *filters=None*, *query=None*, *table=None*, *query_schema='query_schema'*, *dump_schema='response_schema'*, *conn=None*)

Get a single model

Parameters

- **filters** (Optional[Dict[str, Any]]) – dictionary of filters, if not provided it will be created from the query_schema
- **query** (Optional[Dict[str, Any]]) – additional query parameters, only used when filters is not provided
- **table** (Optional[Table]) – sqlalchemy table, if not provided the default *db_table* is used instead
- **conn** (Optional[AsyncConnection]) – optional db connection

async update_one(*, *data=None*, *filters=None*, *query=None*, *table=None*, *body_schema='body_schema'*, *query_schema='query_schema'*, *dump_schema='response_schema'*, *conn=None*)

Update a single model

async delete_one(*, *filters=None*, *query=None*, *table=None*, *query_schema='query_schema'*, *conn=None*)

Delete a single model

Return type Row

async delete_list(*, *filters=None*, *query=None*, *table=None*, *query_schema='query_schema'*, *conn=None*)

delete multiple models

Return type CursorResult

4.2.7 Websocket

Websocket RPC

class openapi.ws.manager.**Websocket**

A websocket connection

socket_id: str = ''

websocket ID

SocketsManager

class openapi.ws.manager.**SocketsManager**

A base class for websocket managers

property sockets: Set[*openapi.ws.manager.Websocket*]

Set of connected *Websocket*

property channels: *openapi.ws.channels.Channels*

Pub/sub *Channels* currently active on the running pod

add(*ws*)

Add a new websocket to the connected set

Return type None

remove(*ws*)
Remove a websocket from the connected set

Return type None

server_info()
Server information

Return type Dict

async close_sockets()
Close and remove all websockets from the connected set

Return type None

async publish(*channel*, *event*, *body*)
Publish an event to a channel

Property channel the channel to publish to

Property event the event in the channel

Property body the body of the event to broadcast in the channel

This method should raise CannotPublish if not possible to publish

Return type None

async subscribe(*channel*)
Subscribe to a channel

This method should raise CannotSubscribe if not possible to publish

Return type None

async subscribe_to_event(*channel*, *event*)
Callback when a subscription to an event is done

Property channel the channel to publish to

Property event the event in the channel

You can use this callback to perform any backend subscriptions to third-party streaming services if required.

By default it does nothing.

Return type None

async unsubscribe(*channel*)
Unsubscribe from a channel

Return type None

Channels

class openapi.ws.channels.**Channels**(*sockets*)
Manage channels for publish/subscribe

property registered: Tuple[str, ...]
Registered channels

Return type Tuple[str, ...]

async register(*channel_name*, *event_name*, *callback*)
Register a callback

Parameters

- **channel_name** (str) – name of the channel
- **event_name** (str) – name of the event in the channel or a pattern
- **callback** (Callable[[], None]) – the callback to invoke when the *event* on *channel* occurs

Return type *Channel*

async unregister(*channel_name*, *event*, *callback*)
Safely unregister a callback from the list of event callbacks for channel_name

Return type Optional[*Channel*]

Channel

class openapi.ws.channel.**Channel**(*name*, *_events*=<*factory*>)
A websocket channel

property events
List of event names this channel is registered with

register(*event_name*, *callback*)
Register a `callback` for `event_name`

event_pattern(*event*)
Channel pattern for an event name

WsPathMixin

class openapi.ws.path.**WsPathMixin**
Api Path mixin for Websocket RPC protocol

SOCKETS_KEY = 'web_sockets'
Key in the app where the Web Sockets manager is located

property sockets: *openapi.ws.manager.SocketsManager*
Connected websockets

Return type *SocketsManager*

property channels: *openapi.ws.channels.Channels*
Channels for pub/sub

Return type *Channels*

decode_message(*msg*)
Decode JSON string message, override for different protocol

Return type Any

encode_message(*msg*)
Encode as JSON string message, override for different protocol

Return type str

Subscribe

class openapi.ws.pubsub.**Subscribe**
Mixin which implements the subscribe and unsubscribe RPC methods

Must be used as mixin of *WsPathMixin*

property channel_callback: openapi.ws.pubsub.ChannelCallback
The callback for *Channels*

async ws_rpc_subscribe(*payload*)
Subscribe to an event on a channel

async ws_rpc_unsubscribe(*payload*)
Unsubscribe to an event on a channel

Publish

class openapi.ws.pubsub.**Publish**
Mixin which implements the publish RPC method

Must be used as mixin of *WsPathMixin*

get_publish_message(*data*)
Create the publish message from the data payload

Return type Any

async ws_rpc_publish(*payload*)
Publish an event on a channel

4.3 Validation

Validation is an important component of the library and it is designed to validate data to and from JSON serializable objects.

To validate a simple list of integers

```
from typing import List

from openapi.data.validate import validate

validate(List[int], [5,2,4,8])
# ValidatedData(data=[5, 2, 4, 8], errors={})

validate(List[int], [5,2,"5",8])
# ValidatedData(data=None, errors='not valid type')
```

The main object for validation are python dataclasses:

```
from dataclasses import dataclass
from typing import Union

@dataclass
class Foo:
    text: str
```

(continues on next page)

(continued from previous page)

```
    param: Union[str, int]
    done: bool = False


validate(Foo, {})
# ValidatedData(data=None, errors={'text': 'required', 'param': 'required'})

validate(Foo, dict(text=1))
# ValidatedData(data=None, errors={'text': 'not valid type', 'param': 'required'})

validate(Foo, dict(text="ciao", param=3))
# ValidatedData(data={'text': 'ciao', 'param': 3, 'done': False}, errors={})
```

4.3.1 Validated Schema

Use the *validated_schema()* to validate input data and return an instance of the validation schema. This differs from *validate()* only when dataclasses are involved

```
from openapi.data.validate import validated_schema

validated_schema(Foo, dict(text="ciao", param=3))
# Foo(text='ciao', param=3, done=False)
```

4.3.2 Supported Schema

The library support the following schemas

- Primitive types: `str`, `bytes`, `int`, `float`, `bool`, `date`, `datetime` and `Decimal`
- Python `dataclasses` with fields from this supported schema
- `List` from `typing` annotation with items from this supported schema
- `Dict` from `typing` with keys as string and items from this supported schema
- `Union` from `typing` with items from this supported schema
- `Any` to skip validation and allow for any value

Additional, and more powerful, validation can be achieved via the use of custom `dataclasses.field()` constructors (see *Data Fields* reference).

```
from dataclasses import dataclass
from typing import Union
from openapi.data import fields

@dataclass
class Foo:
    text: str = fields.str_field(min_length=3, description="Just some text")
    param: Union[str, int] = fields.integer_field(description="String accepted but␣
↪convert to int")
    done: bool = False = fields.bool_field(description="Is Foo done?")
```

(continues on next page)

(continued from previous page)

```
validated_schema(Foo, dict(text="ciao", param="2", done="no"))
# Foo(text='ciao', param=2, done=False)
```

4.3.3 Dump

Validated schema can be dump into valid JSON via the *dump()* function

4.4 Queries

The library provide some useful tooling for creating dataclasses for validating schema when querying paginated endpoints.

4.4.1 Pagination

Base class

class openapi.pagination.**Pagination**

Base class for Pagination

apply(*visitor*)

Apply pagination to the visitor

Return type None

links(*url*, *data*, *total=None*)

Return links for paginated data

Return type Dict[str, str]

paginated(*url*, *data*, *total=None*)

Return paginated data

Return type *PaginatedData*

Paginated Data

class openapi.pagination.**PaginatedData**(*url: yarl.URL*, *data: list*, *pagination:* openapi.pagination.pagination.Pagination, *total: Optional[int] = None*)

Named tuple containing paginated data and methods for retrieving links to previous or next data in the pagination

data: list

Paginated list of data

header_links()

Header links

Return type str

json_response(*headers=None*, ***kwargs*)

Create a JSON response with link header

pagination: *openapi.pagination.pagination.Pagination*
Pagination dataclass which created the data

total: Optional[int]
Total number of records (supported by limit/offset pagination only)

url: yarl.URL
Base url

Visitor

class openapi.pagination.**PaginationVisitor**
Visitor for pagination

apply_cursor_pagination(*cursor*, *limit*, *order_by*, *previous*)
Apply cursor pagination

apply_offset_pagination(*limit*, *offset*, *order_by*)
Apply limit/offset pagination

4.4.2 Limit/Offset Pagination

openapi.pagination.**offsetPagination**(**order_by_fields*, *default_limit=50*, *max_limit=100*)
Crate a limit/offset *Pagination* dataclass

Return type Type[*Pagination*]

4.4.3 Cursor Pagination

openapi.pagination.**cursorPagination**(**order_by_fields*, *default_limit=50*, *max_limit=100*)

Return type Type[*Pagination*]

4.4.4 searchable

openapi.pagination.**searchable**(**searchable_fields*)
Create a dataclass with *search_fields* class attribute and *search* field. The search field is a set of field which can be used for searching and it is used internally by the library, while the *search* field is the query string passed in the url.

Parameters **searchable_fields** – fields which can be used for searching

Return type type

4.5 Websocket RPC

The library includes a minimal API for Websocket JSON-RPC (remote procedure calls).

To add websockets RPC you need to create a websocket route:

```
from aiohttp import web
from openapi.spec import ApiPath
from openapi.ws import WsPathMixin
from openapi.ws.pubsub import Publish, Subscribe

ws_routes = web.RouteTableDef()

@ws_routes.view("/stream")
class Websocket(ApiPath, WsPathMixin, Subscribe, Publish):

    async def ws_rpc_info(self, payload):
        """Server information"""
        return self.sockets.server_info()
```

the *WsPathMixin* adds the get method for accepting websocket requests with the RPC protocol. *Subscribe* and *Publish* are optional mixins for adding Pub/Sub RPC methods to the endpoint.

The endpoint can be added to an application in the setup function:

```
from aiohttp.web import Application

from openapi.ws import SocketsManager

def setup_app(app: Application) -> None:
    app['web_sockets'] = SocketsManager()
    app.router.add_routes(ws_routes)
```

4.5.1 RPC protocol

The RPC protocol has the following structure for incoming messages

```
{
    "id": "abc",
    "method": "rpc_method_name",
    "payload": {
        ...
    }
}
```

The *id* is used by clients to link the request with the corresponding response. The response for an RPC call is either a success

```
{
    "id": "abc",
    "method": "rpc_method_name",
    "response": {
        ...
```

(continues on next page)

(continued from previous page)

```
    }
}
```

or an error

```
{
    "id": "abc",
    "method": "rpc_method_name":
    "error": {
        ...
    }
}
```

4.5.2 Publish/Subscribe

To subscribe to messages, one need to use the *Subscribe* mixin with the websocket route (like we have done in this example). Messages take the form:

```
{
    "channel": "channel_name",
    "event": "event_name",
    "data": {
        ...
    }
}
```

4.5.3 Backend

The websocket backend is implemented by subclassing the *SocketsManager* and implement the methods required by your application. This example implements a very simple backend for testing the websocket module in unittests.

```python
import asyncio

from aiohttp import web
from openapi.ws.manager import SocketsManager

class LocalBroker(SocketsManager):
    """A local broker for testing"""

    def __init__(self):
        self.binds = set()
        self.messages: asyncio.Queue = asyncio.Queue()
        self.worker = None
        self._stop = False

    @classmethod
    def for_app(cls, app: web.Application) -> "LocalBroker":
        broker = cls()
        app.on_startup.append(broker.start)
        app.on_shutdown.append(broker.close)
```

(continues on next page)

(continued from previous page)

```
        return broker

    async def start(self, *arg):
        if not self.worker:
            self.worker = asyncio.ensure_future(self._work())

    async def publish(self, channel: str, event: str, body: Any):
        """simulate network latency"""
        if channel.lower() != channel:
            raise CannotPublish
        payload = dict(event=event, data=self.get_data(body))
        asyncio.get_event_loop().call_later(
            0.01, self.messages.put_nowait, (channel, payload)
        )

    async def subscribe(self, channel: str) -> None:
        """ force channel names to be lowercase"""
        if channel.lower() != channel:
            raise CannotSubscribe

    async def close(self, *arg):
        self._stop = True
        await self.close_sockets()
        if self.worker:
            self.messages.put_nowait((None, None))
            await self.worker
            self.worker = None

    async def _work(self):
        while True:
            channel, body = await self.messages.get()
            if self._stop:
                break
            await self.channels(channel, body)

    def get_data(self, data: Any) -> Any:
        if data == "error":
            return self.raise_error
        elif data == "runtime_error":
            return self.raise_runtime
        return data

    def raise_error(self):
        raise ValueError

    def raise_runtime(self):
        raise RuntimeError
```

4.6 Environment Variables

Several environment variables can be configured at application level

- **DATASTORE** Connection string for postgresql database
- **BAD_DATA_MESSAGE** (Invalid data format), message displayed when data is not in valid format (not JSON for example)
- **ERROR_500_MESSSAGE** (Internal Server Error), message displayed when things go wrong
- **DBPOOL_MAX_SIZE** (10), maximum number of connections in postgres connection pool
- **DBECHO**, if set to *true* or *yes* it will use *echo=True* when setting up sqlalchemy engine
- **MICRO_SERVICE_PORT** (8080), default port when running the *serve* command
- **MICRO_SERVICE_HOST** (0.0.0.0), default host when running the *serve* command
- **MAX_PAGINATION_LIMIT** (100), maximum number of objects displayed at once
- **DEF_PAGINATION_LIMIT** (50), default value of pagination
- **SPEC_ROUTE** (/spec), path of OpenAPI spec doc (JSON)

4.7 Glossary

asyncio Python `asyncio` module for asynchronous IO programming

dataclasses Python `dataclasses` module

openapi The OpenAPI specification: a broadly adopted industry standard for describing modern APIs. The most up-to-date versions are available on github OpenAPI-Specification

schema Types and syntax for a valid *Supported Schema*

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W