aioli Documentation

Robert Wikman <rbw@vault13.org>

Nov 03, 2019

Setup

Aioli was created with usability, extensibility and performance in mind, and is used for building modular, fast and highly concurrent web backend applications of any size.

It works with modern versions of Python (3.6+), is *event loop driven* and makes use of asyncio with uvloop.

Check out The Guestbook Repository for a comprehensive RESTful HTTP example.

CHAPTER 1

Install

The Aioli Framework can be installed using Python pip.

```
$ pip3 install aioli
```

CHAPTER 2

Configure

The *Application* and associated *Units* can be configured using either environment variables, or by a dictionary provided to the *config* parameter when creating the `Application`.

Note: *Environment* takes precedence over *Application Constructor* config.

2.1 Mappings

Environment and Dictionary configs uses different naming conventions, for obvious reasons, but follows the same logic.

2.1.1 Application

Mappings used for configuring core parts of an Aioli Application.

Locations

- Dictionary key: "aioli_core"
- Environment prefix: "AIOLI_CORE"
- Run-time access: `aioli.Application.config`

Mappings

Dictionary	Environment	DEFAULT
dev_host	AIOLI_CORE_DEV_HOST	127.0.0.1
dev_port	AIOLI_CORE_DEV_PORT	5000
api_base	AIOLI_CORE_API_BASE	/api
pretty_json	AIOLI_CORE_PRETTY_JSON	False
allow_origins	AIOLI_CORE_ALLOW_ORIGINS	["*"]
debug	AIOLI_CORE_DEBUG	False

2.1.2 Unit

A custom *Unit* configuration schema can be defined using the `UnitConfigSchema` class, which comes with a set of common parameters listed below.

Locations

- Dictionary key: [unit_name]
- Environment prefix: [PACKAGE_NAME]
- Run-time access: `aioli.Unit.config`

Mappings

Dictionary	Environment	DEFAULT
debug	[PACKAGE_NAME]_DEBUG	None
controllers_enable	[PACKAGE_NAME]_CONTROLLERS_ENABLE	True
services_enable	[PACKAGE_NAME]_SERVICES_ENABLE	True

Check out the *Unit Config Schema docs* for info on how to extend the base schema with custom parameters.

2.2 Environment

Configuring Aioli using *Environment Variables* can be useful in some environments.

Example

```
$ export AIOLI_CORE_DEV_HOST="0.0.0.0"
$ export AIOLI_CORE_DEV_PORT="5555"
$ export AIOLI_RDBMS_TYPE="mysql"
$ export AIOLI_RDBMS_HOST="127.0.0.1"
$ export AIOLI_RDBMS_DATABASE="aioli"
$ export AIOLI_RDBMS_USERNAME="aioli"
$ export AIOLI_RDBMS_PASSWORD="super_secret"
$ export AIOLI_GUESTBOOK_VISITS_MAX="10"
```

2.3 Constructor

The configuration can be provided as a dictionary to the *config* parameter when creating the `Application`.

Check out an *Application configuration example*.

2.4 Access

Both `Application` and `Unit` configurations can be easily accessed from both *Service* and *Controller* instances, using the *config* property.

CHAPTER 3

Deploy

Note: Containerization deployment guidelines will be added shortly

CHAPTER 4

Application

The *Application* constructor expects one or more *Unit* modules to be registered with the instance.

class aioli.**Application**(*units*, ***kwargs*)

Aioli application core

Parameters

- **config** – Configuration dictionary
- **units** – List of units

Variables

- **log** – Aioli Application logger
- **registry** – ImportRegistry instance
- ***config*** – Application config

Example – Guestbook Web API making use of the aioli_rdbms extension

```
import aioli_guestbook
import aioli_rdbms

import toml

from aioli import Application

app = Application(
    config=toml.load("aioli.cfg"),
    units=[
        aioli_guestbook,
        aioli_rdbms,
    ]
)
```

CHAPTER 5

Unit

For code to be allowed registration with an Application–be it a local package, an extension or something else–it must adhere to the Aioli *Unit* format, which mainly serves the purpose of providing modularity, consistency and flexibility.

Units, in its simplest form, are tagged groups of one or more *Components* in the form of:

- Services: Implements application logic and exposes an API for internal consumption
- Controllers: Handles HTTP requests and typically interacts with *Service* APIs

class aioli.**Unit**(*meta=None*, *auto_meta=False*, *controllers=None*, *services=None*, *config=None*)

Associates components and meta with a unit, for registration with a Aioli Application.

Parameters

- **meta** – Unit metadata, cannot be used with auto_meta
- **auto_meta** – Attempt to automatically resolve meta for Unit, cannot be used with meta
- **controllers** – List of Controller classes to register with the Unit
- **services** – List of Services classes to register with the Unit
- **config** – Unit Configuration Schema

Variables

- **app** – Application instance
- **meta** – Unit meta dictionary
- **log** – Unit logger
- ***config*** – Unit config
- **controllers** – List of Controllers registered with the Unit
- **services** – List of Services registered with the Unit

call_startup_handlers()

Call startup handlers in the order they were registered (integrated services last)

Example – Creating a Unit with Controller and Service layers

```
from aioli import Unit

from .service import VisitService, VisitorService
from .controller import HttpController
from .config import ConfigSchema


export = Unit(
    auto_meta=True,
    controllers=[HttpController],
    services=[VisitService, VisitorService],
    config=ConfigSchema,
)
```

5.1 Extend

Units can be connected using `integrate()` or `connect()`, and those with the sole purpose of serving others, are known as *Extensions*.

Example – Leverage the aioli-rdbms Unit to gain database access

```
from aioli import BaseService
from aioli_rdbms import DatabaseService

from .database import UserModel

class UsersService(BaseService):
    db = None

    async def on_startup(self):
        self.db = (
            self.integrate(DatabaseService)
            .use_model(UserModel)
        )

    async def get_one(user_id):
        return await self.db.get_one(pk=user_id)

    ...
```

CHAPTER 6

Config

Unit config schemas make use of the Marshmallow library and offers a clean and safe way of customizing Units.

Read more in *Setup/Configure*, or check out an *Example*.

class aioli.config.**UnitConfigSchema**(**args*, ***kwargs*)

Unit configuration schema

Variables

- **debug** – Set debug level for unit, effectively overriding Application's debug level
- **path** – Unit path, uses Unit name if empty
- **should_import_services** – Setting to False skips Service registration for this Unit
- **should_import_controllers** – Setting to False skips Controller registration for this Unit

CHAPTER 7

Controller

The *Controller Component* takes care of routing, request handling, transformation, validation and more.

Multiple Controllers of different type may coexist in a Unit's Controller layer.

7.1 HTTP

The *BaseHttpController* is mainly for building RESTful HTTP APIs, and may leverage the Aioli toolkit to perform transformation, validation, request-object plucking and more.

API

class aioli.controller.**BaseHttpController**(*unit*, *config_override=None*)

HTTP API Controller

Parameters **unit** – Attach to this unit

Variables

- **unit** – Parent Unit
- ***config*** – Unit configuration
- **log** – Controller logger

on_request(**args*)

Called on request arrival for this Controller

on_shutdown()

Called when the Application is shutting down gracefully

on_startup()

Called after the Unit has been successfully attached to the Application and the Loop is available

Example – Controller without route handlers

```
from aioli.controller import BaseHttpController

from .service import VisitService


class HttpController(BaseHttpController):
    def __init__(self):
        super(HttpController, self).__init__(unit)

        self.log.debug("Guestbook opening")
        self.visit = VisitService(unit)

    async def on_startup(self):
        self.log.debug(f"Guestbook opened")

    async def on_request(self, request):
        self.log.debug(f"Request received: {request}")
```

7.1.1 Route

Route handlers are standard Python methods decorated with *@route*.

API

aioli.controller.decorators.**route**(*path*, *method*, *description=None*)

Prepares route registration, and performs handler injection.

Parameters

- **path** – Handler path, relative to application and unit paths
- **method** – HTTP Method
- **description** – Endpoint description

Returns Route handler

Example – Route handler without transformation helpers

```
from aioli.controller import BaseHttpController, Method, route

from .service import VisitService


class Controller(BaseController):
    def __init__(self):
        self.visit = VisitService()

    @route("/", Method.GET, "List of entries")
    async def visits_get(self, request):
        # Pass along the query params as-is.
        # Then..
        # Return whatever get_many() returned.
        return await self.visit.get_many(**query)
```

7.1.2 Transform

Transformation is implemented on route handlers using *@takes* and *@returns*. These decorators offer a simple yet powerful way of shaping and validating request data, while also ensuring API endpoints returns according to their schemas.

Takes

The *@takes* decorator is used to instruct Aioli how to deserialize and validate parts of a request, and injects the validated data as arguments into the decorated function.

API

`aioli.controller.decorators.`**`takes`**(*props=None*, ***schemas*)

Takes a list of schemas used to validate and transform parts of a request object. The selected parts are injected into the route handler as arguments.

Parameters

- **`props`** – List of *Pluck* targets
- **`schemas`** – list of schemas (kwargs)

Returns Route handler

Example – Route handler making use of @takes

```
from aioli.controller import (
    BaseHttpController, ParamsSchema, RequestProp,
    Method, route, takes
)

from .service import VisitService


class Controller(BaseController):
    def __init__(self):
        self.visit = VisitService()

    @route("/", Method.GET, "List of entries")
    @takes(query=ParamsSchema)
    async def visits_get(self, query):
        # Transform and validate query params using
        # ParamsSchema and pass along to get_many().
        # Then..
        # Return whatever get_many() returned.
        return await self.visit.get_many(**query)
```

Returns

The *@returns* decorator takes care of serializing data returned by its route handler, into JSON.

API

`aioli.controller.decorators.`**`returns`**(*schema_cls=None*, *status=200*, *many=False*)

Returns a transformed and serialized Response

Parameters

- **schema_cls** – Marshmallow.Schema class
- **status** – Return status (on success)
- **many** – Whether to return a list or single object

Returns Response

Example – Route handler making use of @takes and @returns

```
from aioli.controller import (
    BaseHttpController, ParamsSchema, RequestProp,
    Method, route, takes, returns
)

from .service import VisitService


class Controller(BaseController):
    def __init__(self):
        self.visit = VisitService()

    @route("/", Method.GET, "List of entries")
    @takes(query=ParamsSchema)
    @returns(Visit, many=True)
    async def visits_get(self, query):
        # Transform and validate query params using
        # ParamsSchema and pass along to VisitService.get_many()
        # Then..
        # Transform and dump the object returned by get_many()
        # using the Visit schema, as a JSON encoded response.
        return await self.visit.get_many(**query)
```

7.2 GraphQL

Note: GraphQL support will be added shortly.

7.3 WebSocket

Note: WebSocket support will be added shortly.

CHAPTER 8

Service

The typical *Service Component* takes care of interacting with external systems: Databases, Remote Web APIs, Messaging systems, etc. and provides an API for internal consumption.

Check out the *Connecting Services* example to see how a service can integrate and interact with other services.

class aioli.service.**BaseService**(*unit*, *config_override=None*)

Base Service class

Parameters **unit** – Attach to this unit

Variables

- **app** – Application instance
- **registry** – Application ImportRegistry
- **unit** – Parent Unit
- ***config*** – Unit configuration
- **log** – Unit logger

connect(*cls*)

Reuses existing instance of the given Service class, in the context of the Unit it was first registered with.

Parameters **cls** – Service class

Returns Existing Service instance

integrate(*cls*)

Creates a new instance of the given Service class in the context of the current Unit.

Parameters **cls** – Service class

Returns Integrated Service

on_shutdown()

Called when the Application is shutting down gracefully

on_startup()

Called after the Unit has been successfully attached to the Application and the Loop is available

CHAPTER 9

CLI

The *Aioli CLI* provides a set of commands for managing an Aioli project.

9.1 attach

Attaches the given application instance.

Example – attach app at my_app:export

```
$ aioli --app_path my_app:export attach
```

9.2 info

Dump details about the current application.

Example – dump info about my_app:export

```
$ aioli info
```

9.3 create

Create a new app with the given name.

Command input

Name	Default	Description
–dst_path	Current directory	Directory in which to create the project
–profile	minimal	One of: minimal, guesthouse, whoami
–confirm	Not set	Answer yes to confirmations

Example – create a new app using the guesthouse profile

```
$ aioli create beachhouse --profile guesthouse
```

9.4 start

Starts a development server.

Name	Default	Description
–host	127.0.0.1	Bind socket to this host
–port	5000	Bind socket to this port
–no_reload	Not set	Disable the reloader
–no_debug	Not set	Disable debug mode
–workers	1	Number of workers

Example – Start the attached application on port 127.0.0.1:1234

```
$ aioli start --port 1234
```

9.5 units

Show a info about attached or remotely available (PyPI) Aioli units.

9.5.1 local

Work with local units.

Example – show a list of local Units

```
$ aioli units local list
```

9.5.2 pypi

Work with Units on PyPI.

Example – show details about the aioli-openapi Unit on PyPI

```
$ aioli units pypi show aioli-openapi
```

CHAPTER 10

Shell

The *Aioli Shell* is an interactive layer on top of the *Aioli CLI* for managing *Applications* and working with local or remote *Units*.

Invoking the shell:

```
$ aioli --app_path my_app:export shell
```

CHAPTER 11

Service

Read more in the *Service* documentation.

11.1 Connecting Services

Service making use of other services with *integrate* and *connect*.

```
from aioli.service import BaseService
from aioli.exceptions import AioliException, NoMatchFound

from aioli_rdbms import DatabaseService

from .visitor import VisitorService

from .. import database


class VisitService(BaseService):
    visitor: VisitorService
    db = None

    async def on_startup(self):
        self.db = self.integrate(DatabaseService).use_model(database.VisitModel)
        self.visitor = self.connect(VisitorService)

    async def get_authored(self, visit_id, remote_addr):
        visit = await self.db.get_one(pk=visit_id)
        if visit.visitor.ip_addr != remote_addr:
            raise AioliException(status=403, message="Not allowed from your IP")

        return visit

    async def delete(self, visit_id, remote_addr):
```

(continues on next page)

(continued from previous page)

```
        visit = await self.get_authored(visit_id, remote_addr)
        await visit.delete()

    async def update(self, visit_id, payload, remote_addr):
        visit = await self.get_authored(visit_id, remote_addr)
        return await self.db.update(visit, payload)

    async def create(self, remote_addr, visit):
        visit_count = await self.db.count(visitor__ip_addr__iexact=remote_addr)
        visits_max = self.config["visits_max"]

        if visit_count >= visits_max:
            raise AioliException(
                status=400,
                message=f"Max {visits_max} entries per IP. Try deleting some old ones.
↪",
            )

        async with self.db.manager.database.transaction():
            city, country = await self.visitor.ipaddr_location(remote_addr)

            visitor = dict(
                name=visit.pop("visitor_name"),
                ip_addr=remote_addr,
                location=f"{city}, {country}",
            )

            try:
                visit["visitor"] = await self.visitor.db.get_one(**visitor)
            except NoMatchFound:
                visit["visitor"] = await self.visitor.db.create(**visitor)
                self.unit.log.info(f"New visitor: {visit['visitor'].name}")

            visit_new = await self.db.create(**visit)
            self.log.info(f"New visit: {visit_new.id}")

        return await self.db.get_one(pk=visit_new.id)
```

CHAPTER 12

Config

Read more about the Aioli Configuration System in the *Configuration* documentation.

12.1 Unit config schema

This example uses code from the aioli_rdbms extension Unit.

12.1.1 Create

Define a custom Unit configuration schema.

File: aioli_rdbms/config.py

```
from aioli.config import UnitConfigSchema, fields, validate


class ConfigSchema(UnitConfigSchema):
    type = fields.String(
        validate=validate.OneOf(["mysql", "postgres"]),
        required=True
    )
    username = fields.String(required=True)
    password = fields.String(required=True)
    host = fields.String(missing="127.0.0.1")
    port = fields.Integer(missing=3306)
    database = fields.String(missing="aioli")
```

12.1.2 Associate

Associate the configuration schema with a Unit.

File: aioli_rdbms/__init__.py

```
from aioli import Unit

from .service import DatabaseService
from .config import ConfigSchema


export = Unit(
    auto_meta=True,
    controllers=[],
    services=[DatabaseService],
    config=ConfigSchema,
)
```

12.1.3 Configure

Create the configuration using the format of choice.

File: aioli.cfg

```
[aioli_core]
dev_port = 5555
path = "/api"
pretty_json = false
allow_origins = ["*"]
debug = true

[aioli_guestbook]
path = "/guestbook"
# Maximum number of visits per IP
visits_max = 14

[aioli_rdbms]
type = "(mysql|postgres)"
username = "user"
password = "pass"
host = "127.0.0.1"
port = 3306
database = "aioli"
```

12.1.4 Register

Parse the configuration file and pass it as a Dictionary to the *`Application`* constructor.

File: my_application/main.py

```
import aioli_guestbook
import aioli_rdbms

import toml

from aioli import Application

app = Application(
    config=toml.load("aioli.cfg"),
    units=[
```

(continues on next page)

(continued from previous page)

```
        aioli_guestbook,
        aioli_rdbms,
    ]
)
```

Python Module Index

a

Index