aioli Documentation

Robert Wikman <rbw@vault13.org>

Jun 21, 2019

SETUP

Aioli is a Framework for building RESTful HTTP and WebSocket API packages, with a sensible separation between request/response handling (transformation, validation, etc), application logic and data access layers.

Furthermore, it makes use of asyncio, is lightweight and provides good performance and concurrency.

Not in mood for reading docs? Check out the The Guestbook example: a Comprehensive RESTful HTTP API package.

CHAPTER

ONE

INSTALL

The Aioli Framework can be installed using Python pip.

```
$ pip3 install aioli
```

CHAPTER

TWO

CONFIGURE

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

Note: 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
debug	AIOLI_CORE_DEBUG	False

2.1.2 Package

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

Locations

- Dictionary key: [package_name]
- Environment prefix: [PACKAGE_NAME]
- Run-time access: `aioli.Package.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 *Package 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 the *Package config schema* for an example.

2.4 Access

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

CHAPTER

THREE

DEPLOY

3.1 Development

Given an Application like:

```
# File: main.py

import aioli_guestbook
import aioli_rdbms

import toml

from aioli import Application

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

. . . the Aioli CLI can be used to start the Application (using a built-in Uvicorn server).

```
$ python3 -m aioli dev-server main:app
```

3.2 Production

Work in progress

CHAPTER

FOUR

APPLICATION

To run the Aioli application, an *Application* instance must be created. Its constructor expects a *list* of at least one *Package* to be registered with the application.

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

Creates an Aioli application

Parameters

- **config** – Configuration dictionary
- **packages** – List of package tuples [(<mount path>, <module>), . . .]
- **kwargs** – Keyword arguments to pass along to Starlette

Variables

- **log** – Aioli Application logger
- **packages** – Packages registered with the Application

add_exception_handler(*exception*, *handler*)

Add a new exception handler

Parameters

- **exception** – Exception class
- **handler** – Exception handler

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("config.toml"),
    packages=[
        aioli_guestbook,
        aioli_rdbms,
    ]
)
```

CHAPTER

FIVE

PACKAGE

A *Package* is a namespaced and labelled group of components that can be imported into an *Application*.

There are two main components for building Web API Packages: *Controller* & *Service*

When developing an Aioli application, local *Packages* typically contain code that makes sense to modularize in the Application at hand.

class aioli.**Package**(*name*, *description*, *version*, *controllers=None*, *services=None*, *config=None*)

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

Parameters

- **name** – Package name ([a-z, A-Z, 0-9, -])
- **description** – Package description
- **version** – Package semver version
- **controllers** – List of Controller classes to register with the Package
- **services** – List of Services classes to register with the Package
- **config** – Package Configuration Schema

Variables

- **app** – Application instance
- **log** – Package logger
- **state** – Package state
- **path** – Package Path
- **name** – Package Name
- **version** – Package Version
- ***config*** – Package config
- **controllers** – List of Controllers registered with the Package
- **services** – List of Services registered with the Package

Example – Creating a Package with Controller and Service layers

```
from aioli import Package

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

(continues on next page)

(continued from previous page)

```
export = Package(
    name="aioli_guestbook",
    version="0.1.0",
    description="Example guestbook Package",
    controllers=[HttpController],
    services=[VisitService, VisitorService],
    config=ConfigSchema,
)
```

CHAPTER

SIX

CONFIG

Package config schemas makes use of the Marshmallow library and offers a simple, clean and safe way of customizing Packages.

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

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

Package configuration schema

Variables

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

CHAPTER

SEVEN

CONTROLLER

The Controller layer takes care of routing & request/response handling (transformation, validation, etc). Multiple Controllers of different type may coexisting in a Package's Controller layer.

7.1 HTTP

Creating an HTTP Interface – be it RESTful or otherwise – is done using the *BaseHttpController* class.

API

class aioli.controller.**BaseHttpController**(*pkg*)
HTTP API Controller

Parameters **pkg** – Attach to this package

Variables

- **pkg** – Parent Package
- ***config*** – Package configuration
- **log** – Controller logger

async on_request(**args*)
Called on request arrival for this Controller

async on_shutdown()
Called when the Application is shutting down gracefully

async on_startup()
Called after the Package has been successfully attached to the Application

Example – Controller without route handlers

```
from aioli.controller import BaseHttpController

from .service import VisitService


class HttpController(BaseHttpController):
    def __init__(self):
        self.visit = VisitService()
        self.log.debug("Guestbook opening")

    async def on_startup(self):
        self.log.debug(f"Guestbook opened at {self.package.path}")
```

(continues on next page)

(continued from previous page)

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

7.1.1 Routing

Route handlers are standard Python methods decorated with the *@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 package 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):
        # Just pass along the query params as-is.
        #
        # Serialize and return whatever get_many() returns.
        return await self.visit.get_many(**request.query_params)
```

7.1.2 Transformation

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 making sure API endpoints only returns expected data.

This makes the API more secure and consistent.

Takes

The *@takes* decorator is used to instruct Aioli how to deserialize and validate parts of a request, and injects the resulting dictionaries as arguments to 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 against ParamsSchema,
        # then pass it along to get_many().
        #
        # Serialize and return whatever get_many() returns.
        return serialize(await self.visit.get_many(**query))
```

Returns

The *@returns* decorator takes care of serializing the data returned by the route handler.

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
)
```

(continues on next page)

(continued from previous page)

```
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 against ParamsSchema,
        # then pass it along to get_many().
        #
        # Transform and dump the object returned from get_many() -
        # according to the Visit schema, as a JSON encoded response
        return await self.visit.get_many(**query)
```

7.2 WebSocket

Note: *Work in progress*

WebSocket support is not fully integrated with the Framework yet, hence not documented.

CHAPTER

EIGHT

SERVICE

The Service layer typically takes care of interacting with external applications: Databases, Remote Web APIs, Message Queues, etc.

Services can be connected and–to provide a good level of flexibility–supports both Inheritance and two types of Composition.

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

class aioli.service.**BaseService**(*pkg*)

connect(*svc*)
Reuses existing instance of the given Service class, in the context of the Package it was first registered with.

Parameters **svc** – Service class

Returns Existing Service instance

integrate(*svc*)
Creates a new instance of the given Service class in the context of the current Package.

Parameters **svc** – Service class

Returns Service instance

async on_shutdown()
Called when the Application is shutting down gracefully

async on_startup()
Called after the Package has been successfully attached to the Application

CHAPTER

NINE

ABOUT

The typical Aioli extension-type *Package* manages one or more *Service* objects, provides an API of its own, and may contain *Controller* code as well.

CHAPTER

TEN

CREATE

Extensions make use of the *BaseService* class and usually implements the Factory pattern teamed by the *integrate()* method.

Check out the aioli-rdbms extension for an example.

CHAPTER

ELEVEN

USE

Extensions are registered with the *Application*, just like a regular *Package*– and usually have their Services incorporated into other *Packages*.

Example

Register the local *users* Package and its dependency, *aioli_rdbms*.

```
import aioli_rdbms

from .packages import users

app = Application(
    packages=[
        ("/users", users),
        (None, aioli_rdbms),
        ...
    ]
)
```

The *aioli_rdbms.Service* can now be attached to *users.UsersService*:

```
from aioli import BaseService
from aioli_rdbms import DatabaseService

from .database import UserModel

class UsersService(BaseService):
    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

TWELVE

PUBLISH

Shortly, a Package Management CLI will be added, along with the https://pkgs.aioli.dev website for showing useful info about extension-type *Packages*; their trust status, install instructions, author and license data, as well as links to source code and more.

CHAPTER

THIRTEEN

ABOUT

The idea with these examples, or snippets if you will, is to show how components *can* be built using the Aioli Framework.

If you're looking for a fully functional example; check out The Guestbook example – a Comprehensive RESTful HTTP API package.

CHAPTER

FOURTEEN

SERVICE

Read more in the *Service* documentation.

14.1 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):
        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"]
```

(continues on next page)

(continued from previous page)

```
        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.pkg.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

FIFTEEN

CONFIG

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

15.1 Package config schema

This example uses code from the aioli_rdbms extension Package.

15.1.1 Create

Define a custom Package configuration schema.

File: aioli_rdbms/config.py

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


class ConfigSchema(PackageConfigSchema):
    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")
```

15.1.2 Associate

Associate the configuration schema with a Package.

File: aioli_rdbms/__init__.py

```
from aioli import Package

from .service import DatabaseService
from .config import ConfigSchema


export = Package(
```

(continues on next page)

(continued from previous page)

```
    name="aioli_rdbms",
    version="0.1.0",
    description="ORM and CRUD Service for Aioli with support for MySQL and PostgreSQL
↪",
    controllers=[],
    services=[DatabaseService],
    config=ConfigSchema,
)
```

15.1.3 Configure

Create the configuration using the format of choice.

File: config.toml

```
[aioli_core]
dev_port = 5555
path = "/api"
debug = true

[aioli_rdbms]
type = "mysql"
host = "127.0.0.1"
port = 3306
database = "aioli"
username = "aioli01"
password = "super_secret"

[aioli_guestbook]
visits_max = 10
```

15.1.4 Load

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

File: my_application/main.py

```
import toml
import aioli

import aioli_rdbms
import aioli_guestbook

app = aioli.Application(
    config=toml.loads(["/path/to/config.toml"]),
    packages=[
        (None, aioli.rdbms),
        ("/guestbook", aioli_guestbook)
    ]
)
```

PYTHON MODULE INDEX

a

INDEX