Aurora Documentation

Release 0.9.0

Yeiniel Suárez Sosa

January 20, 2013

CONTENTS

Contents:

CHAPTER

ONE

WHY I NEED TO DEVELOP A NEW PYTHON WEB FRAMEWORK

I begin developing Web applications on 2004 for the library at UCLV university, back at that time I use PHP as programming language. But 5 years latter it became clear for me that the same design decisions that make PHP a easy to use programming language for Web development purposes where the same that degrade exponentially the performance of my products as they became more complex.

On 2009 I met Python and **PEP 333** for the first time and the first impression was: "Oh wow! This is what I was looking for!". The simple concept of application presented at **PEP 333** and the realization of its capabilities make me switch to Python as programming language for my future Web experiments.

After toying with most of the open source Python based Web frameworks I realize of some issues that arise on them:

- The use of thread local (global in the worst cases) objects for sharing resources/services. This issue make the testing of components complicated.
- The use of the Web request object as a registry for sharing resources/services. This create a dependency problem and corrupt the meaning of Web request.
- The use of the WSGI protocol for connecting required application components as middleware.
- The use of the WSGI protocol for writing actual Web request handlers. This force the developer to handle the complexities of this low level protocol.

The first two issues share a common consequence: component requirements gets replaced by the framework (because you need the framework to provide the component requirements trough the use of a global object or the request object) and the real component requirement expression is at the component implementation and not at the component interface. This break the rule that explicit is better than implicit.

This are the conditions that motivate me to develop a new Python based Web framework with **PEP 8** as coding style and the Zen of Python as design philosophy.

CHAPTER

TWO

WRITING YOUR FIRST AURORA BASED WEB APPLICATION, PART 1

In this tutorial you will learn by example how to create a Web based Blogging application using the Aurora Web application framework.

For cut and paste purposes, the source code for all stages of this tutorial can be browsed at https://github.com/yeiniel/aurora/tree/master/documentation/intro/webapp/src/tutorial-1.

2.1 Basic Layout

Scaffolding

Currently the Aurora Web application framework doesn't provide a tool to automate the task of creating scaffolds for a new Web application, it must be done by hand.

Before we start coding our Web application we need to setup the basic layout for it. First you need to create a folder that host all application's specific files, lets call this folder *tutorial-1*. Inside that folder create a Python module named `application`, we will use this module to host the Web application definition. Open the module file once created in your preferred text editor and write the following:

```
#! /usr/bin/env python3
from aurora import webapp
__all__ = ['Application']


class Application(webapp.Application):
    """ Web Blogging application
    """

if __name__ == '__main__':
    from wsgiref import simple_server
    from aurora.webapp import foundation

    wsgi_app = foundation.wsgi(Application())
    httpd = simple_server.make_server('', 8008, wsgi_app)

    print("Serving on port 8008...")
    httpd.serve_forever()
```

This Python module define a class (the `Application` class) that inherit from a base class provided by the Web application framework, this is the Web application definition. Additional code is provided to allow the module to be executed directly as a console application, in this case the WSGI Web server shipped with the Python standard library is used to serve a Web application based on that definition on port `8008`.

If you execute this module at the console and open the http://localhost:8008/ address in your preferred Web browser you will see a default and simple `Not Found` message. This is correct and it means you did not setup a Web request path mapping that associate a Web request handler to the base Web application path. As you can see the Web application framework doesn't do any magic for you and this is one of its design principles.

2.2 Defining the Blogging services

There are three services that all Blogging's platforms provide:

- present summary of recently published Posts.
- present a published Post.
- form for composing a new Post.

We are going to implement this three services into our Web application. As a first step we are going to add the corresponding three service definitions stubs (methods) to the Web application definition (class) as follows:

```
    def list_posts(self, request: foundation.Request) -> foundation.Response:
        """ List summaries for posts added more recently. """
        return request.response_factory(text="""
        <html>
            <body>
                <h1>List of posts</h1>
                <div>
                    <h2><a href="/post/1">Post title</a></h2>
                    <p>
                        Post summary (or the initial segment of post
                        content).
                    </p>
                </div>
                <div>
                    <h2><a href="/post/1">Post title</a></h2>
                    <p>
                        Post summary (or the initial segment of post
                        content).
                    </p>
                </div>
                <div>
                    <h2><a href="/post/1">Post title</a></h2>
                    <p>
                        Post summary (or the initial segment of post
                        content).
                    </p>
                </div>
            </body>
        </html>
        """)

    def show_post(self, request: foundation.Request) -> foundation.Response:
        """ Show a post. """
        return request.response_factory(text="""
        <html>
```

```
            <body>
                <h1><a href="/post/1">Post title</a></h1>
                <p>
                    Post content (full).
                </p>
            </body>
        </html>
        """)

    def add_post(self, request: foundation.Request) -> foundation.Response:
        """ Add a new Blog post. """
        return request.response_factory(text="""
        <html>
            <body>
                <h1>Add new post</h1>
                <form action="/add" method="post">
                    <fieldset>
                        <legend>Add Post</legend>
                        <div class="clarfix">
                            <label for="title">Title</label>
                            <div class="input">
                                <input type="text" id="title" name="title"></input>
                            </div>
                        </div>
                        <div class="clarfix">
                            <label for="content">Content</label>
                            <div class="input">
                                <textarea id="content" name="content" class="xlarge" rows="3"></texta
                            </div>
                        </div>
                        <div class="actions">
                            <input class="btn primary" type="submit" name="submit" value="Add" />
                        </div>
                    </fieldset>
                </form>
            </body>
        </html>
        """)
```

As you can see, an additional Python module has been imported, and used to annotate the three service definition stubs (the `aurora.webapp.foundation` module). This has been done to make clear to any user that read the source code, that this three services implement the Web request handling protocol. By making this three services implement this protocol (interface), we make them able to handle Web requests (read the `Handler` documentation for more information). But if you restart your Web application at the console once you make the changes to your copy of the Python module, you will not going to be able to see this services at action because the Web application don't know which Web requests map to the different services that implement the Web request handling protocol (remember that the Web application framework don't do magic for you).

2.3 Mapping Web request paths to Web request handlers

Now we are going to add code that map this three services as Web request path characteristic (specifically the `_handler` characteristic) with the Web application `mapper`, this way the Web application will know which Web requests sent to the three different Web request handlers. The code looks as follows:

```
    def __init__(self):
        self.mapper.add_rule(mapping.Route('/'), _handler=self.list_posts)
        self.mapper.add_rule(mapping.Route('/post/(?P<id>\d+)'),
            _handler=self.show_post)
        self.mapper.add_rule(mapping.Route('/compose'),
            _handler=self.add_post)
```

As you can see, an additional Python module has been imported (the `aurora.webapp.mapping` module), and used to create the Web request path rules used to map the tree Web request handlers. Once that you update your Web application definition code, restart your Web application running at the console and refresh the http://localhost:8008/ address in your browser. You will see the list of Posts summaries produced by the stub, from there you can browse to the pages of the independent Posts.

2.4 Conclusion

Well, this is all for now. In this tutorial you learn how to create a Web application using the Aurora library and how to write services that act as Web request handlers. In the *next* part of this tutorial you will learn how to integrate components shipped with the Aurora library to address common needs and how to to create components to integrate third party libraries.

CHAPTER

THREE

WRITING YOUR FIRST AURORA BASED WEB APPLICATION, PART 2

In this second part of the Web application tutorial you will learn how to integrate components shipped with the Aurora library to address common needs and how to create components to integrate third party libraries.

Before you start coding we recommend you to copy the Python module produced in the first part of this tutorial named `application` into a new folder named `tutorial-2`. For cut and paste purposes, the source code for all stages of this tutorial can be browsed at https://github.com/yeiniel/aurora/tree/master/documentation/intro/webapp/src/tutorial-2.

3.1 Using views

The MVC pattern is widely used in Web application development. Products created using the Web application framework from the Aurora library are not enforced to use this pattern or any other related to the concept of separation of concerns (MVP, etc). Even though the library provide a component very useful to construct the `View` part for an MVC based product (The `Views` component). Next we are going to refactor our Web application to implement Web request handlers using the MVC pattern. As a first step we are going to make accessible the `Views` component from the Web application services by adding it as a Web application attribute (property).

```
1     @property
2     def views(self) -> views.Views:
3         try:
4             return self.__views
5         except AttributeError:
6             self.__views = views.Views()
```

This step require that you first import the `views` module. Now we are ready to modify the Web request handlers so they implement the MVC pattern. The code will look as follows:

```
def list_posts(self, request: foundation.Request) -> foundation.Response:
    """ List summaries for posts added more recently. """
    return self.views.render2response(request, 'list.html')

def show_post(self, request: foundation.Request) -> foundation.Response:
    """ Show a post. """
    return self.views.render2response(request, 'show.html')

def add_post(self, request: foundation.Request) -> foundation.Response:
    """ Add a new Blog post. """
    if request.method == 'POST':
```

```
        # process form submission
        pass
    else:
        return self.views.render2response(request, 'form.html')
```

The Web request handler services has been modified to use the `render2response()` service. This service takes a Web request object, the path of the template file (without the last extension, read the API documentation for that service for more information), an arbitrary number of arguments used as `view` context and return the corresponding Web response object.

Once the Web application has been modified the only missing step is adding the templates used to render the views, but we are going to do that latter once we write the data persistence logic. Create a folder inside the Web application folder named `templates`, this is the one that we are going to register as template source using the following snippet of code added to the Web application `__init__()` method:

```
        # add the 'templates' folder as template source for the 'views'
        # component
        self.views.add_path(os.path.join(os.path.dirname(__file__),
            'templates'))
        self.views.add_default('url_for', self.url_for)
```

This snippet additionally add as default `views` context element the `url_for()` Web application service used to create links on the Web responses.

At this point is recommended that you review the *Web application components* section of the *API Documentation* and learn about other Web application components shipped with the Aurora library.

3.2 Integrating SQLAlchemy

SQLAlchemy is a powerful Database abstraction library writen in Python that provide a ORM pattern implementation. We re going to use the ORM to implement the data layer of the Web application (the M part of the MVC design pattern). In order to integrate the library into the application we are going to add a new component and will name it `engine_provider`. Add a Python package named `components` to the Web application folder and there add a Python module with that name with the following content:

```
import sqlalchemy

__all__ = ['EngineProvider']


class EngineProvider:
    """ 'SQLAlchemy'_ support provider.

    This component provide support for use the ''SQLAlchemy'' library to
    connect to one database. The 'get_engine' method is the only exposed
    service.

    The source database is configured using the 'dsn' component attribute.

    If you need different database connections in the same application you
    can create multiple instances of this component and distribute them as
    needed.

    .. _SQLAlchemy: http://www.sqlalchemy.org/
    """

```

```
    dsn = 'sqlite:///application.db'

    def __init__(self, dsn=None):
        self._engine = sqlalchemy.create_engine(dsn or self.dsn)

    def get_engine(self) -> sqlalchemy.engine.Engine:
        """ Return an :class:`sqlalchemy.engine.Engine` object.
        :return: a ready to use :class:`sqlalchemy.engine.Engine` object.
        """
        return self._engine
```

Once we have that component in place add Web application models into a separated Python module named `models` in the Web application folder as follows:

```
import sqlalchemy
from sqlalchemy.ext import declarative

__all__ = ['Post']


Model = declarative.declarative_base()


class Post(Model):
    __tablename__ = 'blog_post'

    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    title = sqlalchemy.Column(sqlalchemy.String, nullable=False)
    content = sqlalchemy.Column(sqlalchemy.Text, nullable=False)
    author = sqlalchemy.Column(sqlalchemy.String, nullable=False)
    date = sqlalchemy.Column(sqlalchemy.DateTime, nullable=False)
```

Now to make the models available to the Web application definition we need to add the following import line:

```
from components import engine_provider
```

As we do with the `views` component we need to make accessible the component from the Web application services by importing the component Python module into the Web application definition Python module and adding it as a Web application attribute (property):

```
    def db(self) -> engine_provider.EngineProvider:
        try:
            return self.__db
        except AttributeError:
            self.__db = engine_provider.EngineProvider()
            return self.__db
```

Now to make things easy lets ensure that tables on database needed to store the model data are available by adding the following line to the Web application `__init__()` method:

```
        # try to create the database tables if needed
        models.Post.metadata.create_all(self.db.get_engine())
```

And finally lets show you the Web request handlers services once modified to use the `get_engine()` service from the `EngineProvider` component:

```
        """ List summaries for posts added more recently. """
        orm_session = orm.sessionmaker(bind=self.db.get_engine())()

```

```
        return self.views.render2response(request, 'list.html',
            posts=orm_session.query(models.Post).order_by(
                sqlalchemy.desc(models.Post.date))[:10],
            blog=self)

    def show_post(self, request: foundation.Request) -> foundation.Response:
        """ Show a post. """
        post_id = request.params['id']

        orm_session = orm.sessionmaker(bind=self.db.get_engine())()

        return self.views.render2response(request, 'show.html',
            post=orm_session.query(models.Post).filter_by(id=post_id).one(),
            blog=self)

    def add_post(self, request: foundation.Request) -> foundation.Response:
        """ Add a new Blog post. """
        if request.method == 'POST':
            # process form submission
            # TODO: need to implement form validation here.
            post = models.Post(
                title=request.POST['title'],
                content=request.POST['content'],
                author='',
                date=datetime.datetime.utcnow(),
            )

            orm_session = orm.sessionmaker(bind=self.db.get_engine())()
            orm_session.add(post)
            orm_session.commit()

            # redirect to the post page
            resp = request.response_factory()
            resp.status_int = 302
            resp.location = request.application_url

            return resp
        else:
            return self.views.render2response(request, 'form.html', blog=self)
```

For this code to work you need first to import the following modules: `datetime`, `sqlalchemy` and `sqlalchemy.orm`.

The used architecture allow us to share a common database connections across a set of components and provide different database engine for different components if needed (consider the case you are using one component that use specific features from one database engine). Once we have the Blogging application passing real data objects into the `views` we only need to show you the templates used.

`templates/list.html.suba`

```
<section id="index">
    %(for post in posts:)
        %(render('summary', post=post, blog=blog))
    %/
</section>
```

`templates/summary.suba`

```
<section class="summary" id="summary-%(str(post.id))">
    <h1><a href="%(url_for(_handler=blog.show_post, id=str(post.id)))">
        %(post.title)</a></h1>
    <div class="content">%(post.content[:60]) ...</div>
</section>
```

templates/show.html.suba

```
<section class="show" id="show-%(str(post.id))">
    <h1><a href="%(url_for(_handler=blog.show_post, id=str(post.id)))">
        %(post.title)</a></h1>
    <div class="date">%(str(post.date))</div>
    <div class="content">%(post.content)</div>
</section>
```

templates/form.html.suba

```
<section id="form">
    <form action="%(url_for(_handler=blog.add_post))" method="post">
        <fieldset>
            <legend>Add Post</legend>
            <div class="clarfix">
                <label for="title">Title</label>
                <div class="input">
                    <input type="text" id="title" name="title"></input>
                </div>
            </div>
            <div class="clarfix">
                <label for="content">Content</label>
                <div class="input">
                    <textarea id="content" name="content" class="xlarge" rows="3"></textarea>
                </div>
            </div>
            <div class="actions">
                <input class="btn primary" type="submit" name="submit" value="Add" />
            </div>
        </fieldset>
    </form>
</section>
```

As you can guess by the template file extension this templates are using the suba template engine, a lightweight template engine based on the mod (%) operator.

3.3 Conclusion

Well, this is all for now. In this tutorial you learn how to integrate components provided by the Aurora library into your application and how to create new ones to integrate third party libraries to provide your Web application with features not provided by the Aurora library and you have learn that the Aurora library provide a generic template based `views` framework with support by default for the suba template engine.

CHAPTER

FOUR

WRITING YOUR FIRST AURORA BASED WEB APPLICATION, PART 3

CHAPTER

FIVE

API DOCUMENTATION

5.1 View rendering framework

class aurora.views.**Engine**
Provide syntax specific template based view rendering support.

A Views engine is any callable object that takes a string containing the absolute filesystem path of the View template as first positional argument and any named argument used as context and return a string containing the rendered content.

class aurora.views.**Views**
Provide generic template based view rendering support.

The template file extension is very important, is used to map template file to the engine capable of handling correctly the template syntax.

Template composition support is provided (the `render()` method is mapped into the default context). This allow composing templates with different syntax as long as the Template `Engine` of the composed view support using a context element as callable.

The only template engine available by default is one based on suba and created on the fly the first time the template engine cache is hit (either because a new template engine is added using `add_engine()` or because `render()` is called). A copy of the suba module is shipped with the Aurora library (the *aurora.webcomponents._suba* Python source module.) because it can't be added as a library dependency because this library is not on PYPI Python package index.

add_default(*key*, *value*)
Add a default context item.

Parameters

- **key** – Item key string.
- **value** – Item value

add_engine(*engine*, **extensions*)
Register an `Engine`-like object for rendering files.

Parameters

- **engine** – An `Engine`-like object.
- **extensions** – List of template file extensions.

add_path(*path*)
Add an absolute path as template file source.

Parameters **path** – The absolute path (string) to a folder that store templates files.

render(*template_name*, ***context*)
Render a template into content with context.

Parameters

- **template_name** – The relative template name string without the last extension.
- **context** – The context mapping.

Returns The rendered output string.

5.2 Web application framework

class aurora.webapp.mapping.**Rule**
Map a Web request path and its characteristics back and forth.

A rule implement the logic needed to perform the mapping between a specific Web request path and its associated characteristics. This mapping can be addressed direct or reverse.

This class is not meant to be inherited it is here just for interface documentation purposes.

assemble(***characteristics*)
Map characteristics into its associated Web request path.

If the characteristics mapping can be mapped by this rule then a string containing the `Web request` path unique respect to the Rule is returned otherwise return *False*.

Parameters **characteristics** – The characteristics mapping.

Returns The Web request path or *False*.

match(*path*)
Map the Web request path into its associated characteristics.

If the `Web request` path can be mapped by this rule then a mapping of the characteristics that make this `Web request` path unique respect to the Rule is returned otherwise return *False*.

Parameters **path** – The Web request path.

Returns The characteristic mapping or *False*

class aurora.webapp.mapping.**Route**(*pattern*, ***defaults*)
A Web request path mapping `Rule` that use pattern matching.

The pattern matching algorithm used is plain regex. The expressions allowed in the pattern in order to be reassembled into a valid `Web request` path are described by the `dialect` attribute of this class. Any deviation may result in an incorrectly assembled `Web request` path.

At initialization the pattern is passed as first positional argument, other named arguments are used as default values for any optional pattern group.

class aurora.webapp.mapping.**DefaultRule**
A Web request path mapping `Rule` that map all Web requests.

This is a simple Web request path mapping `Rule` very useful for mapping the Web request handler for not mapped Web requests. It match all Web request paths and return always *False* on `assemble()` method calls.

class aurora.webapp.mapping.**Mapper**
Map a Web request path and its characteristics using multiple rules.

The Mapper implement the same interface provided by `Rule` and add the possibility of tagging rules using an additional mapping of elements known as metadata. Additionally metadata can be used to override `Web request` path characteristics with default values.

Every time the `match()` or `assemble()` methods are called all added rules are evaluated in the reverse order of addition. This implementation detail imply that in order to function correctly generic rules must be added first and specific ones latter.

add_rule(*rule*, ***metadata*)
Add a `Rule` and its associated metadata to the mapping.

Parameters

- **rule** – A mapping `Rule`.
- **metadata** – The `Rule` associated metadata.

assemble(****characteristics*)
Map characteristics into its associated Web request path.

A `Rule` is mapped only in the case that all `Rule` associated metadata are present in the characteristics mapping and have the same value.

Parameters **characteristics** – The characteristics mapping.

Returns The Web request path or *False*.

match(*path*)
Map the Web request path into its associated characteristics.

Once a `Rule` is mapped successfully its associated metadata is used to update the `Web request` path characteristics mapping.

Parameters **path** – The Web request path.

Returns The characteristic mapping or *False*

5.3 Web application components

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

INDEX