Table Of Contents

Introduction

Aurora is an MVC web framework for creating CRUD applications quickly and simply.

It based on REST architecture. In another word it is a RESTFUL web framework.

Aurora is written in Python, and built on Flask.

Why Python

As a programmer who have had over a decade experience in programming on different platforms and in different programming languages, I believe python undoubtedly is one of the best programming languages of all time. It is cross-platform and OS-independent.

I have already made a bunch of closed-source web apps in php and Node.js. They both are great, and I don't want to bring down any programming languages in particular. But with confidence I can say that python is something else.

Python is really fast, easy to use, and easy to deploy. Its syntax is so clean and makes writing code really fun. YouTube, Instagram, and much more famous companies rely on python.

It has a great community, tons of packages and everything you need as a developer.

Why Flask

Flask is a micro web framework purely written in python. Flask is lightweight, simple and flexible. With flask you can create simple up to complex apps. Working with flask is totally up to the developer. It won't decide for you, what extensions or databases to use.

Flask is really great, and if you feel comfortable to use it alone, it's perfectly fine.

Why Aurora

I have made a closed-source framework purely written in php, for making web apps, something similar to CodeIgniter.

With this previous experience, it was about a year ago when I decided to make a web framework for python. First, I decided to write it purely in python, and don't depend on any other packages.

What I understood at the time was, that unlike php, python isn't specifically made for the web.

Python is a cross-platform programming language. It takes considerably more time and effort to handle routes, requests and so on.

I had two choices, one was to spend more time and energy to write everything from zero, and another choice was to rely on some packages that other developers have written.

First, I searched for available python web frameworks. I found Django, Flask, and Bottle. I spent enough time in all of them to fully understand the philosophy behind python web frameworks.

I have zero interest of comparing these frameworks, and I believe each of them has its own advantages. I totally admire all the developers behind these frameworks. They have done something really precious.

But, to be honest because of the custom MVT structure in these frameworks, it becomes more complicated to make and maintain medium size up to large scale applications. For small applications and somehow medium size application they are fantastic. But whenever your code becomes larger, they become really messy. However, you can create generic views and create more organized codes. But that makes your code even more messy.

With Flask I fill more comfortable. It doesn't decide for me what to do, and I feel more freedom. It's a micro web framework that you can depend on.

Aurora is a framework with more standards and a more organized architecture, partially built on flask and mostly written in python. It uses Flask for routing, request handling and some other stuff.

As I said before if you fill comfortable to use Flask alone, it's perfectly fine. But with Aurora, you can create maintainable web apps more easily, and you can handle large scale web apps much simply.

Aurora has a clean MVC architecture with a great built-in support for popular SQLite, PostgreSQL, and MySQL databases, to create CRUD web apps simply and quickly.

Aurora is a RESTFUL web framework, which makes handling CRUD requests much simpler.

What is MVC

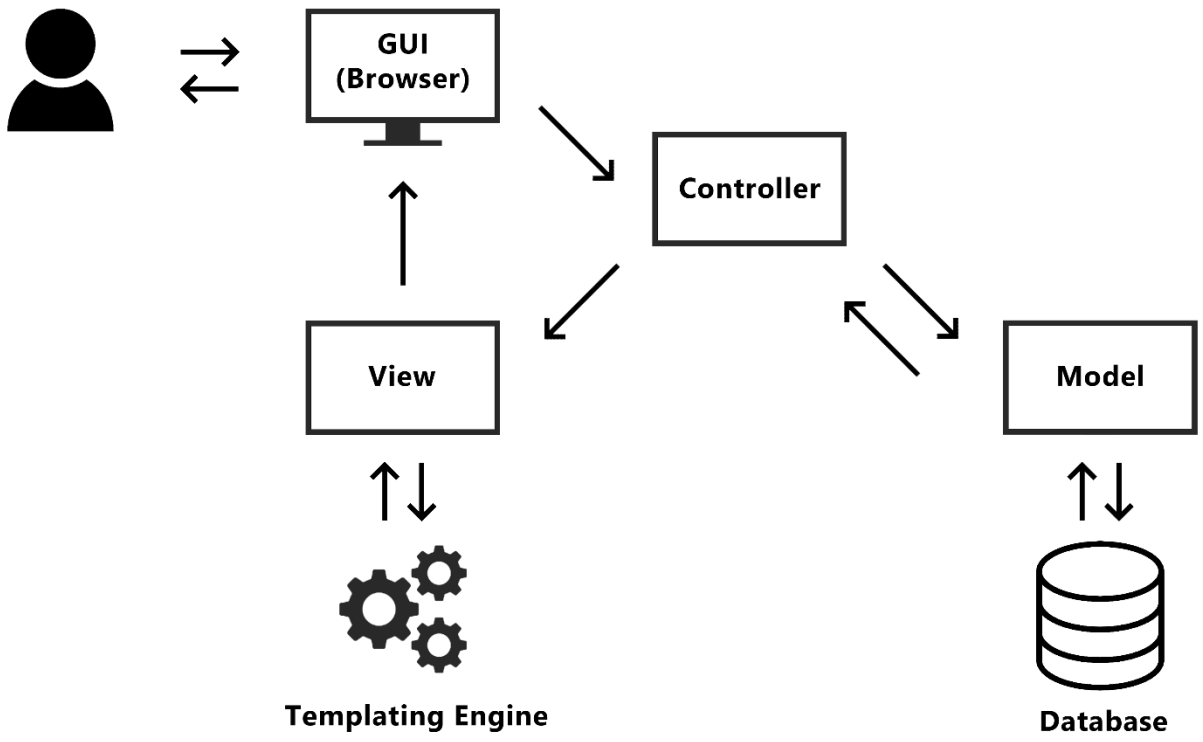
MVC is a software design pattern, which separates an application into three components: **Model**, **View** and **Controller**. It also describes the interactions between them.

> Each component can be divided into smaller counterparts. In this case, they may be referred as models, views and controllers.

MVC traditionally used for desktop graphical user interfaces (desktop GUIs). However, it is become pretty popular for developing web applications.

> Most of the modern back-end web frameworks are built in MVC pattern.

Here's an illustration of an MVC web application:



1. User uses a browser to send a request to the Controller.
2. Controller receives and processes the request.
    a. If the response doesn't need data from the database, it prepares the View to show a proper response based on the request.

b. Else if the request needs data from the database, it sends a CRUD request to the Model.

   i. Model receives the request, and queries the database and sends back the result to the Controller.

   ii. Controller processes the query result, and prepares the View to show a proper response based on the query result.

3. View receives the Controller's command, and sends an updated/rendered HTML file to the browser as the response.

4. Browser shows the response to the user.

MVT Structure

Some frameworks (such as Django or Flask), use a built-in Templating Engine to render the HTML view files. In such cases sometimes the MVC pattern may be referred as **MVT**.

The MVT (Model View Template) pattern is slightly different from MVC.

In MVT architecture, the framework itself takes care of the Controller component, the logic that controls the Model and the View and the interaction between them.

The Template which consists of HTML syntax plus a special templating syntax, describes how dynamic content would be inserted and pure HTML will be outputted.

Using Aurora framework, you are able to use the MVC structure plus a templating engine, which will be discussed at the following.

Model

Model is the lowest part of the pattern, which is responsible for managing the data of the application. It receives the user request as a command (a CRUD request) via the controller.

It is the application's dynamic data structure, independent from the user interface.

Model can store data to the database.

Model can retrieve data from the database.

Model can modify the database.

View

View is the data which is meant to be shown to the end user.

It can be, any type of information such as texts, graphics, charts, diagrams, tables etc.

The view renders presentation of the model in a particular format.

Controller

Controller is the core component of the pattern. It is the application logic which accepts the user inputs and converts them into commands for the model and view.

The controller responds to the user input and performs interactions on the data model objects.

The controller optionally validates the user input, before passing it to the model.

Get Started

Perquisites

Before the installation of Aurora, make sure you have python installed on your machine.

You can check it simply by running the following command in your CLI program:

```
$ python --version
```

Important Note: Aurora framework needs python version 3 or higher to run.

For windows users use **`py`** instead of **`python`** for all CLI commands in this documentation.

If you don't have python, first download it from its official website:

https://www.python.org/

Installation

You can install Aurora by running:

```
$ pip install aurora-mvc
```

If you are new to python and don't know how to install a package, visit the following link:

https://packaging.python.org/en/latest/tutorials/installing-packages/

Aurora needs flask to be installed on your environment. Using the above command, you should have it already. To check if it is installed, run the following command:

```
$ flask --version
```

If you don't have it installed on your machine, you can simply install it by running the following command:

```
$ pip install flask
```

Usage

To get started with Aurora simply do the following steps:

1. Create the project (root app) directory:

```
$ mkdir my_app
```

Here **my_app** is a variable name. Change it to anything of your choice at any time you want.

2. Install a python virtual environment in the same path that the project directory exists:

```
$ python -m venv venv
```

3. Activate the virtual environment:

Linux / Mac

```
$ . venv/bin/activate
```

Windows

```
$ venv\scripts\activate
```

4. Navigate to the project directory:

```
(venv) cd my_app
```

Notice that the project directory must be empty, otherwise you will get an error on the next step.

5. Initialize the root app with Aurora via python shell:

```
(venv) python
>>> from aurora import init_app
>>> init_app.start()
```

It will create all the required directories, modules and packages in the project directory.

Congratulations!

You successfully initialized the root app. Now you are ready to get started with Aurora.

6. To start the root app run the following command:

```
(venv) python app.py
```

Now you can visit the following address in your browser to make sure everything is fine:

http://127.1.1.1:5000/

This is the default configuration for the URL of your app.

You will learn how to modify it at the Aurora Configuration section.

Project Skeleton

The basic skeleton of a simple Aurora project with a single child app called **`app_name`** is something like this:

```
/controllers
    /app_name
        _controllers.py
        ControllerName.py
/models
    _models.py
    Users.py
/statics
    /app_name
        main.css
        main.js
/views
    /app_name
        layout.html
        index.html
_apps.py
app.py
```

```
config.py
manage.py
```

- The `controllers` directory contains all the controller packages for child apps of the root app. Here we have a package called `app_name` for a child app with the same name. The `_controllers.py` is a system module created with Aurora, which contains the controller classes and their URLs and methods for the current package. Here `ControllerName.py` is a controller class for the current app.

Every child app has its own controller packages and classes inside the `controllers` directory.

All the modules start with a single `_` are system modules created with Aurora. Please do not change them until you fully understand the framework structure and how it works.

In general, you don't need to change them, because you can simply manage them with the framework itself.

- The `models` directory, contains all the models of your project. In another word it is the database CRUD package for the root app and all the child apps. The `_models.py` is a system module, that contains a list of all available model names. Here it contains the name of a single model class called `Users.py`, which is created by the framework for you.
- The `statics` directory, contains all the sub directories for the static files of your child apps, that you want to show to the public.

Danger: Do not put any sensitive data in the `statics` directory, because they meant to be available for the public.

- The `views` directory, contains all the sub directories for the view files of your child apps, that you want to be rendered and sent back to the browser by a controller class. They may contain static files from the `statics` directory.

Aurora uses Jinja2 templating engine for rendering the view files.

- The `_apps.py` module, is a system module that contains a collection of all available apps for your project. It also shows the base URL for each app.

- The `app.py` module, is the primary module of your project. In general, you don't need to do anything with this file. It just instantiates the Aurora class of the Aurora framework to serve the root app of your project.

The root app is an instance of the `Flask` class.

The root app is responsible to serve the controllers of child apps. To put it simply, it connects each route with a specific controller class. It also is responsible to run the server for your apps.

- The `config.py` module is the configuration module of your Aurora project. You will learn about this module in a separate section.
- The `manage.py` module is the CLI application for managing your project. You will learn about this module in a separate section.

Important Notes:

Please do not touch lines with the flag `#do-not-change-me` on the system modules of Aurora framework, otherwise it stops working as expected.

Please do not change the `__init__.py` modules of Aurora framework, otherwise it stops working as expected.

Aurora Apps

The Root App

Almost everything in Aurora framework is an application. Even the HTTP errors. Even the root app itself. All child apps are instances of the root app.

When you run an Aurora project using the following command, it will serve the root app:

```
(venv) python app.py
```

All you need to do is to create your own apps. The root app serves them for you.

Child Apps

Beside the root app you can make unlimited number of child apps for your project.

> You will learn how to create and manage your child apps in the Manage Apps section.

The project skeleton from the past section, was simplified in order to clarifying the basic structure of an Aurora project.

The one you have got after initializing the root app, using the given command is a little larger:

```
...
>>> init_app.start()
...
```

It has two child apps: `errors`, and `aurora`.

The `errors` app is responsible for handling the HTTP errors you can use with all other apps. You will learn about `errors` app in the HTTP Errors section.

> Do not delete this app because you will need it for all the other apps. You will learn how to simply modify the views and styles of this app at the continuation of this documentation.

The `aurora` app is created only for making sure that your project runs correctly for the first time, and you can use it as a reference, to know what version you have, and links to documentations and so on. If you are a minimalist and don't want to have it in your project you can simply delete it using the following command:

```
(venv) python manage.py delete-app
```

Then it will prompt you to enter the app name, in this case it is `aurora`, and after confirmation it will be gone.

> You will learn more about Aurora CLI commands in the later section.

App Components

Every child app has its own components. Some components are common between apps.

Generally saying Aurora apps are divided into the following components:

- Primary Components:
  - **Models** – Every model is a class. They are common for all apps.
  - **Views** – Every view is an HTML like file with Jinja2 syntax. Each app has its own views.
  - **Controllers** – Every controller is a class. Each app has its own controllers.
- Secondary Components:
  - **Forms** – Every form is a class. Each app can have its own forms.
  - **Statics** – Each app can have its own static files. Files like images, CSS files, JS file, etc.

Aurora CLI

Using Aurora, you don't need to manually create or delete your app components.

You can simply use the Aurora CLI app.

To see a list of all available CLI commands run:

```
(venv) python manage.py --help
```

The general syntax is:

```
(venv) python manage.py COMMAND
```

Here are a list of all available command and a description about what they can do:

Command	Description
create-app	Creates a new app with some default components if not exist.
delete-app	Deletes an existing app and all its components.
create-controller	Creates a controller blueprint if not exists for an existing app.
delete-controller	Deletes an existing controller for an existing app.

create-view	Creates a view blueprint if not exists for an existing app.
delete-view	Deletes an existing view for an existing app.
create-form	Creates a form blueprint if not exists for an existing app.
delete-form	Deletes an existing form for an existing app.
create-model	Creates a model blueprint if not exists.
delete-model	Delete an existing model.
check-db	Checks the models and database for changes and migrations.
init-db	Initialize the database for the first time if not initialized already.
migrate-db*	Migrate the model changes into an already initialized database.

*The **migrate-db** command is currently unavailable, for this version.

For now, you can create all your models first, then use the **init-db** command to initialize the database.

Manage Apps

As I mention before, you can find a list of all your apps in the **_apps.py** module.

Create App

You can simply create a new app by running the following command:

```
(venv) python manage.py create-app
```

It will prompt you for some required arguments.

As an example, here we will create an app called **notes**:

```
(venv) python manage.py create-app
App Name: notes
Base URL: my-notes
```

For simplicity you can use the same name for your App Name and its Base URL.

You must provide unique names for both parameters or else you will get an error message.

Valid characters for App Name are: `a-z`, `_` (Notice that the `-` character itself is not valid.)

Valid characters for Base URL are: `a-z`, `-` (Notice that the `_` character is not valid.)

All characters must be in lowercase English alphabetical letters.

It will create the following components for the newly created app:

- Controller blueprints.
- View blueprints.
- Static blueprints.
- Form blueprints.

I will also update the `_apps.py` module for you and you don't need to update it manually.

Note:

Keep that in mind that Aurora framework makes it easier for you to write and maintain your code. It doesn't write your code for you. There is no framework which does. It's the developer's job to deal with. Aurora will be your assistant, and tries to lift some weight off your shoulders.

Only you know what you need, and how to do it. You need to fill out these blueprints with some useful code that serves your project.

Delete App

You can simply delete an existing app by running the following command:

```
(venv) python manage.py delete-app
```

It will prompt you for some required arguments.

As an example, here we will delete an existing app called **`notes`**:

```
(venv) python manage.py delete-app
App Name: notes
```

After that it will alerts you with the data loss, if you confirm, it will delete all the data for the selected app. Here we enter no, because we need this app for the remaining sections of this documentation.

If the requested app doesn't exist, it will alert you.

Very Important Note!

Be super careful with all the delete commands. If you delete something accidentally you won't be able to reverse the process. The data loss will be permanent.

Manage Controllers

Create Controller

You can simply create a new controller for an existing app by:

```
(venv) python manage.py create-controller
```

It will prompt you for some required arguments.

As an example, here we will create a controller called **`Index`** for an existing app called **`notes`**:

```
(venv) python manage.py create-controller
App Name: notes
Controller Name: Index
Controller URL (optional):
```

```
Methods (optional): get, post
```

It will create the new controller for you, and also updates the **`_controllers.py`** module of the selected app.

Even though the Controller URL is optional, if you leave it empty it will become required for the next controller.

The Controller Name must be in "CamelCase" form with at least two "a-z" and "A-Z" characters.

Valid characters for Controller URL are: **`a-z`**, **`-`**, **`/`**, **`<`**, **`:`**, **`>`**

Valid Methods are: post, get, put, delete

If you leave the methods empty, the framework considers the `GET` method for the selected controller.

You will learn more about these methods at RESTFUL CRUD Methods section.

Here are some examples for valid Controller URLs:

Do

```
Controller URL: index
Controller URL: my-notes
```

You shouldn't add the base URL off the current app to the Controller URL. It will be handled by the framework automatically.

Don't

```
Controller URL: notes/index
```

The framework considers this URL as: **`/notes/notes/index/`**

Generic URLs

You can also create generic URLs using the following data types:

Type	Description
int	Accepts integers.
float	Accepts floating point values.
str	Accepts text without slashes (the default).
path	Accepts text with slashes.

```
Controller URL: greet/<name>
Controller URL: greet/<str:name>
Controller URL: edit/<int:id>
```

Note:

Please do not use `/` at the beginning and ending of your Controller URLs. It will be handled automatically by the framework itself.

Using Controllers

Here's the controller class we created in the previous chapter of this section:

```
# Dependencies
from aurora import Controller, View

# The controller class
class Index(Controller):

    # POST Method
    def post(self):
        pass

    # GET Method
    def get(self):
        return 'Page content...'
```

It is the blueprint of your controller created by the Aurora framework.

On the first line it imports the **`Controller`**, **`View`** classes of the Aurora framework.

The current controller inherits the **`Controller`** class, with the given methods.

If you need an **`__init__`** method in the controller class please do not inherit the parent class:

Don't

```
...
class Index(Controller):
    def __init__(self):
        super().__init__()
...
```

Using the code above allows the users access to all methods of the parent class in the current controller. For this specific example even to **`PUT`** and **`DELETE`**. This is something that you don't want.

You can still use all magic methods for this class, that's perfectly fine. Only remember not to inherit the parent class in the constructor method of the current class:

Do

```
...
class Index(Controller):
    def __init__(self):
        # TODO
...
```

However, it is possible to import the Flask request method and handle the methods manually, but this is useless and doesn't make any sense:

Don't

```
from flask import request
...
class Index(Controller):
    ...
    def post(self):
```

```
        if request.method == 'POST':
            # TODO
        else:
            # TODO
...
```

Do

```
...
class Index(Controller):
    ...
    def post(self):
        # TODO
...
```

If a user tries to access a Controller class with a method that is not defined, the **`errors`** app will handle it with a **`405 forbidden method`**, and you don't need to do anything.

Here we returned a simple text in the **`GET`** method of the current controller. You use the **`View`** class to return rendered HTML to the browser:

```
...
class Index(Controller):
    ...
    def get(self):
        return View("index")
...
```

Note that you don't need to provide the **`.html`** for your view, only the view name.

It will try to render a view template called **`index.html`** in the **`/views/notes/`** directory (which doesn't exist yet).

Modifying Controllers

If you already created a controller for an existing app and want to modify it, you should do it manually in the **`_controllers.py`** module of the desired app.

For the notes app here's how it looks like:

```
...
controllers = [
   ('Index', '', ['GET', 'POST']),
]#do-not-change-me
```

For example, we want to remove the post method:

```
...
controllers = [
   ('Index', '', ['GET']),
]#do-not-change-me
```

> **Important Note:**
>
> Please do not touch lines with the flag **`#do-not-change-me`** on this framework, otherwise it stops working as expected.

Now you need to also remove the post method from the **`Index`** controller of the **`notes`** app:

```
# Dependencies
from aurora import Controller, View

# The controller class
class Index(Controller):

    # GET Method
    def get(self):
        return 'Page content...'
```

You can do the same for 'GET', 'PUT', and 'DELETE' methods.

You can also add a method. Only remember to update both **`_controllers.py`** and Controller modules for the current app.

If you want to use a controller more than once, you cannot use the **`create-controller`**, because it considers the requested controller as an existing one and gives you an error message.

However, it's possible to add it manually somehow:

```
controllers = [
...
    ('Greet', 'greet'),
    ('Greet', 'greet/<str:name>'),
...
```

Important Note:

However, the URLs must be unique, otherwise your application stops working and raise an `AssertionError`.

Delete Controller

You can simply delete an existing controller for an existing app by:

```
(venv) python manage.py delete-controller
```

It prompts you for App Name and Controller Name. After confirmation of the data loss, it will delete your controller permanently.

Manage Views

Create View

You can simply create a new view for an existing app by:

```
(venv) python manage.py create-view
```

It will prompt you for some required arguments.

As an example, here we will create a view called `index.html` an existing app called `notes`:

```
(venv) python manage.py create-view
App Name: notes
View Name: index
```

Note that you shouldn't provide the `.html` for your view, only the view name.

Valid characters for the view name are: `a-z`, `-`, `_`

It will create the view for you in the `/views/notes/` directory.

As you may notice the `layout.html` already created for you. Indeed, this is a blueprint created by the `create-app` command.

If you open the `index.html` file you will see the framework have done the templating inheritance and some other stuffs for you.

You can now add your HTML, CSS, and JavaScript codes and make the view what you want.

Using Views

You can use a view inside a controller using the `View` class of the Aurora framework:

```
...
class Index(Controller):
    ...
    def get(self):
        return View("index")
...
```

You don't have to hard code your view to add the app name, the framework recognizes the app name for your view, although you can do it explicitly:

```
...
class Index(Controller):
    ...
    def get(self):
        return View(view="index", app="notes")
```

```
...
```

You can also explicitly set the status code:

```
...
class Index(Controller):
    ...
    def get(self):
        return View(view="index", app="notes", code=302)
...
```

`302` is the default status code for the view.

If you wanted to return other app views, for whatever reason you can do it like:

```
...
class Index(Controller):
    ...
    def get(self):
        return View(view="another_view", app="another_app")
...
```

Delete View

You can simply delete an existing view for an existing app by:

```
(venv) python manage.py delete-view
```

It prompts you for App Name and View Name. After confirmation of the data loss, it will delete your view permanently.

Manage Statics

When you create a new app via **`create-app`** command, it will create empty **`main.css`** and **`main.js`** files in the statics directory of your app, and include them in **`layout.html`** of the selected app.

You can add any type of static files you want to the **`statics`** directory of your app.

For clarification let's take a look at the **`layout.html`** view of our **`notes`** app:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Page Title{% endblock %}</title>
    <link rel="stylesheet" href="/{{STATICS}}/notes/main.css">
    <script src="/{{STATICS}}/notes/main.js"></script>
</head>
<body>
     <!-- Page Content  -->
    {% block body %}{% endblock %}
</body>
</html>
```

`STATICS` here is a global variable you can use in your templates. If you change your statics configuration and statics directory, you don't need to worry about your templates.

The framework considers the new name for you.

So, whenever you need to add a static file to your template, you don't have to hard code it.

You can change the statics directory name via **`config.py`** module.

You will learn more about Global Variables in the Aurora Configuration section.

Manage Forms

Create Form

Aurora v.0.6.0 only supports WTForms for handling forms.

The WTForms is a third-party package, and you should have it installed on your environment.

If you don't have it yet, to install it run the following command:

```
(venv) pip install WTForms
```

To see the WTForms documentation please visit the following address:

> https://wtforms.readthedocs.io/

If you are using WTForms for handling your forms, use the following command to create a new Form Class for an existing app:

```
(venv) python manage.py create-form
```

It will prompt you for some required arguments.

As an example, here we will create a Form Class called **`AddNoteForm`** for our **`notes`** app:

```
(venv) python manage.py create-form
App Name: notes
Form Name: AddNoteForm
```

It will create the mentioned class and updates the **`_forms.py`** system module for you.

> The Form Name must be in "CamelCase" form with at least two "a-z" and "A-Z" characters.

> Even though it's not necessary, but it is a good habit to end your Form Class names with the **`Form`** word. Because later you want to import it in your Controllers, and it ensures that name conflictions not happen.

However, there is a second way to handle this name conflictions using python `import...as...`

Let's take a look at the form class created for us:

```
# Dependencies
from aurora import Forms
from wtforms import *

# The form class
class AddNoteForm(Forms):
    ...
```

Now, you need to add some inputs to your form blueprint:

```
# Dependencies
from aurora import Forms
from wtforms import *

# The form class
class AddNoteForm(Forms):
    title = StringField(label='Title', validators=[validators.DataRequired()])
    content = TextAreaField(label='Content', validators=[validators.DataRequired()])
    submit = SubmitField(label='Submit')
```

Very Important Note:

In Aurora framework by default, the `csrf` is activated for all your Forms.

But, for whatever reason if you want to inactive it, simply set the `csrf` property of the `Meta` class to `False`. However, I strongly recommend you not to do it.

The `csrf` token is super important for your application security.

```
...
# Insecure form class
class InsecureForm(Forms):
    ...
```

```
    class Meta:
        csrf = False
```

Using Forms

The first step to using Form classes is to import them into your controllers.

As an example, we want to import the created form in our **`Index`** controller of our **`notes`** app:

```
# Dependencies
from aurora import Controller, View
from forms.notes import AddNoteForm

# The controller class
class Index(Controller):

    # POST Method
    def post(self):
        pass

    # GET Method
    def get(self):
        form = AddNoteForm()
        return View("index", form=form)
```

Now we need to view it in the **`index.html`** view of the **`notes`** app:

```
{% extends 'notes/layout.html' %}

{% block title %}{% endblock %}

{% block body %}
<h1>Notes App</h1>
<form action="/{{NOTES}}/" method="post" id="add-note-form">
    {{ form.csrf_token }}
    <div>
```

```
        {{ form.title.label }}: {{ form.title(class_="input-class") }}
    </div>
    <div>
        {{ form.content.label }}: {{ form.content }}
    </div>
    <div>
        {{ form.submit }}
    </div>
</form>
<div id="result" style="display: none;"></div>
{% endblock %}
```

- The `NOTES` here is a global variable for your app base url.
- The `form.csrf_token`, will creates a hidden input with the name of `csrf_token` for you.
- Using `form.title(class_="input-class")` syntax, you can add css classes to your inputs.

The above code will generate the following form for you:

```
<form action="/my-notes/" method="post" id="add-note-form">
    <input id="csrf_token" name="csrf_token" type="hidden" value="...">
    <div>
        <label for="title">Title</label>:
        <input class="input-class" id="title" name="title" required type="text" value="">
    </div>
    <div>
        <label for="content">Content</label>:
        <textarea id="content" name="content" required></textarea>
    </div>
    <div>
        <input id="submit" name="submit" type="submit" value="Submit">
    </div>
</form>
```

You can still design custom forms of your choice, without using this syntax, but remember to add the `{{ form.csrf_token }}` at the beginning of your form tag.

Now, it's time to validate your form using WTForms validator:

```
# Dependencies
```

```
from aurora import Controller, View
from forms.notes import AddNoteForm
from flask import request

# The controller class
class Index(Controller):

    # POST Method
    def post(self):
        # Form data
        data = request.form

        # Collect form inputs (for later purposes)
        title = data.get('title')
        content = data.get('content')

        # Pass the requested form data to the form class
        form = AddNoteForm(data)

        # Validate the form
        if form.validate():
            return 'Validation passed!'
        else:
            return 'Validation failed!'

    # GET Method
    def get(self):
        form = AddNoteForm()
        return View('index.html', form=form)
```

Normally you return a rendered template, or a JSON object as the result. Let's send the form with JavaScript and return a JSON response:

Add the following JavaScript Code to the `main.js` of the `notes` app statics:

```
// Document loaded
document.addEventListener('DOMContentLoaded', function() {

    // New Task form
```

```
const add_note_form = document.querySelector('#add-note-form');
if (add_note_form) {
    // Handle form submit
    add_note_form.onsubmit = function () {

        // Form date
        const action = this.getAttribute('action');
        const method = this.getAttribute('method');
        const data = new FormData(this);

        // Submit form data via fetch API
        fetch(action, {
            method: method,
            body: data,
        })
        .then(response => response.json())
        .then(result => {
            // Result container
            let result_div = document.querySelector('#result');

            // Display the result container
            result_div.style.display = 'block';

            // An error occurred
            if (result.error) {
                result_div.innerHTML = `${result.error}`;
            }

            // Everything is fine
            else if (result.success) {
                // Show message
                result_div.innerHTML = `${result.success}`;

                // Reset the form
                this.reset();
            }
        });

        // Prevent the form default behavior
        return false
    };
```

```
    }
});
```

Now update the controller:

```
...
# The controller class
class Index(Controller):

    # POST Method
    def post(self):
        # Form data
        data = request.form

        # Collect form inputs (for later purposes)
        title = data.get('title')
        content = data.get('content')

        # Pass the requested form data to the form class
        form = AddNoteForm(data)

        # Validate the form
        if form.validate():
            return {
                'success': 'Validation passed!',
            }, 200

        else:
            return {
                'error': 'Validation failed!',
            }, 400
...
```

Now try to submit the form, this time using the JavaScript!

Delete Form

You can simply delete an existing form class for an existing app by:

```
(venv) python manage.py delete-form
```

It prompts you for App Name and Form Name. After confirmation of the data loss, it will delete your form class permanently.

Manage Models

Create Model

You can simply create a new model using:

```
(venv) python manage.py create-model
```

It will prompt you for the model's name and if not exists it creates the model blueprint for you.

Here's an example:

```
(venv) python manage.py create-model
Model Name: Notes
```

It will create the model for you and updates the **`_models.py`** module.

> The model's name must be in "CamelCase" form with at least two "a-z" and "A-Z" characters.

Let's take a look at the created model:

```
# Dependencies
from aurora import Model

# The model class
class Notes(Model):

    # Model columns
```

```
    id = Model.column(datatype='integer', not_null=True)

    # Model meta data
    def __init__(self):
        # Inherit the parent class
        super().__init__()

        # Override the parent class default properties
        self.table = 'notes'
        self.primary_key = 'id'
```

As you can see it created a column called **`id`** of type **`integer`** for you. Here we don't want to dive into the details. You will learn more about Model class methods in the Aurora SQL section.

You can change the default table name for the model, and set the primary key.

Note: AuroraSQL only supports a single primary key for each model.

The **`id`** column is the default primary key, even if you delete it the framework create this column in the database for you. If you want to change the name. change the name for both column name and **`primary_key`**.

Modifying Models

The **`Model`** class of Aurora framework, provides several methods to interact with the database. Here we only want to work with the **`column`** method.

Unlike other ORM database engines, AuroraSQL only provides a single method for creating database columns. Which makes it much cleaner.

The only thing, you should consider is to use correct datatype and constraint arguments for each column parameters.

The column method provides the following parameters:

- **`datatype`** – Which is the only required argument.

AuroraSQL follows the same standards for datatypes as SQL standard provides.

You can visit the following links to see what datatypes are available for SQL databases:

https://www.w3schools.com/sql/sql_datatypes.asp

https://www.sqlite.org/datatype3.html

AuroraSQL datatypes are not case sensitive, you can use lowercase, or uppercase. They will be converted to uppercase automatically. Only be careful to type the datatypes correctly.

Notice:

As you may know, SQLite only provides, `NULL`, `INTEGER`, `REAL`, `TEXT`, and `BLOB` datatypes.

But don't worry you can still use all standard SQL datatypes. Hopefully the SQLite database engine usually tries to automatically convert values to the appropriate datatypes.

Auto Increment:

SQLite database provided a cool feature for primary key of the tables. If you provide an integer primary key, it will become an auto increment field automatically.

Using MySQL or PostgreSQL databases, you have to explicitly do it yourself. Each one provides a different syntax.

Using AuroraSQL you don't need to do anything, your primary keys of type integer will become auto increment fields automatically, for all three databases.

- `unique` – It is an optional argument representing the SQL `UNIQUE` constraint.
- `not_null` – It is an optional argument representing the SQL `NUT NULL` constraint.
- `default` – It is an optional argument representing the SQL `DEFAULT` constraint.
- `check` – It is an optional argument representing the SQL `CHECK` constraint.
- `related_to` – It is an optional argument representing the `FOREIGN KEY` constraint.
- `on_update`* – It is an optional argument representing the `ON UPDATE` statement for the `FOREIGN KEY` constraint.
- `on_delete`* – It is an optional argument representing the `ON DELETE` statement for the `FOREIGN KEY` constraint.

* The available options for both `on_delete` and `on_update` args are:

`RESTRICT` | `CASCADE` | `SET NULL` | `NO ACTION` | `SET DEFAULT`

The `CASCADE` is the default value.

As an example, let's add some columns to `Notes` model:

```
...
    # Model columns
    id = Model.column(datatype='integer', not_null=True)
    user_id = Model.column(datatype='integer', related_to='Users')
    title = Model.column(datatype='text', not_null=True)
    content = Model.column(datatype='text', not_null=True)
    date = Model.column(datatype='date', not_null=True)
...
```

The `related_to` argument, creates a `one-to-one` or `one-to-many` relationship, automatically, on the primary key of the selected model.

Here it creates a `one-to-many` relationship with the `Users` model, that created for us by default.

Important Note:

This is the only type of relationship that AuroraSQL **directly** support.

There are ORM frameworks out there, that provide a **direct** `many-to-many` relationship also.

They taking a different approach than the SQL itself provide. Maybe you are a fan of these ORM databases, which is perfectly fine.

However, I personally believe, it makes your code so complicated and I'm not a fan of those kind of approaches.

For making a `many-to-many` relationship with AuroraSQL, the easiest and most efficient way is to create a new model class, with two foreign keys (using `related_to` parameter).

In AuroraSQL each model is responsible to manipulate only a single table in your database.

Using Models

Before import and use a model in our controllers we need to initialize our database for our controllers. For now, we only import our model inside the `Index` controller of our `notes` app:

```
# Dependencies
...
from models import Users, Notes
...
    # GET Method
    def get(self):
        users = Users()
        notes = Notes()
...
```

It doesn't do anything yet. But soon it will be responsible for CRUD requests for the `notes` table of our database.

Delete Models

For delete an existing model, use the following command:

```
(venv) python manage.py delete-model
```

It prompts you for Model Name. After confirmation of the data loss, it will delete your model class permanently.

> It doesn't affect your database directly, but only the model itself. However, if you modify or delete a model, on database initialization or migration it will affect your database.

SQL Reference

If you want to learn more about SQL language, here is a great resource, that you can use as a reference:

https://www.w3schools.com/sql/default.asp

Aurora SQL

Aurora framework uses a custom Database API Engine, called AuroraSQL.

Database is an important part of your app. It is responsible for storing your precious data.

If you are somehow familiar with SQL statements and get tired of normal ORM Database Adapters, AuroraSQL is perfect for you. With AuroraSQL you can use both ORM like and also pure SQL syntax.

AuroraSQL uses something known as DB-API to interact with the database.

The Python Database API (DB-API) defines a standard for Python applications to connect and interact with major database systems.

As the name says, it is an API standard provided by the Database Systems that python libraries can use to interact with those Database Systems.

There are dozens of different DB-API libraries for python, that follow these standards.

Here are some of them:

Database System	Database API Library
SQLite	sqlite3
PostgreSQL	psycopg2
MySQL	mysql.connector

Currently AuroraSQL has a built-in support for major SQL Database Systems, SQLite, PostgreSQL, and MySQL, and uses these DB-API libraries.

Database Systems are programs installed on your machine, whether it is local or on the clouds.

SQLite is a C library that provides a lightweight file-based SQL database that doesn't require a separate server process.

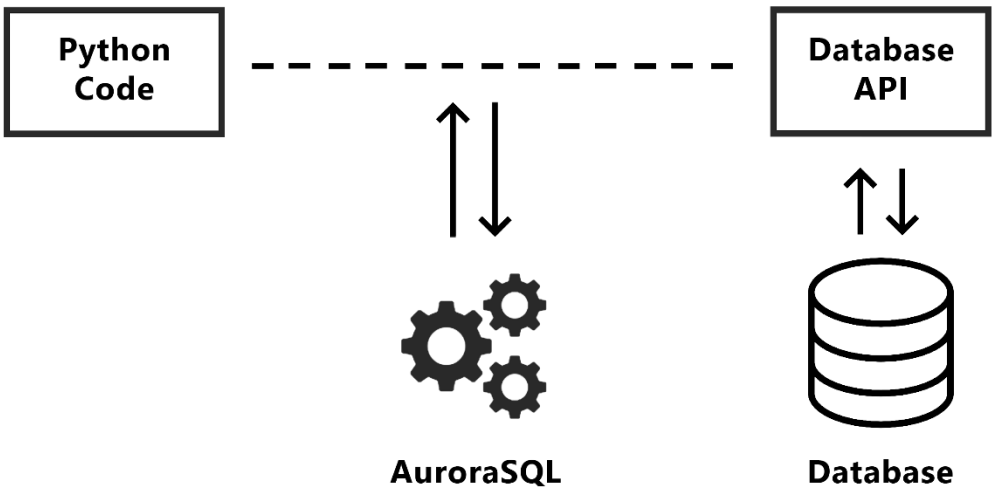
You don't need to install `sqlite3` module. It is included in the standard library since python 2.5.

You may use SQLite for internal data storage. It's also possible to prototype your application using SQLite and then port your code to a larger database such as PostgreSQL or MySQL.

Using DB-API libraries needs a lot of experience and configuration to work with SQL databases safely. Here's where the AuroraSQL comes into place.

Aurora SQL is a middleware in your application that acts like an engine to process your queries in a safe and standard way to make it much easier to work with Database API libraries.

Here's an illustration of how AuroraSQL works:



Every DB-API has its own standards and configuration. You can rely on AuroraSQL and focus on your business logic.

Using AuroraSQL, it is also possible to interact with the DB-API libraries directly which needs extra care and knowledge. Because it can damage your data, if you don't have enough experience with SQL injection and data binding concepts.

Setup the Database

AuroraSQL has a built-in support for major database systems: SQLite, PostgreSQL, and MySQL.

Before you start to initialize or migrate your modules into the database there are a few steps you should take:

1. Select a database system to work with. You can simply do it in your `config.py` module:

```
...
# Database System
DB_SYSTEM = 'SQLite'
...
```

Choose either `'SQLite'`, `'PostgreSQL'`, `'MySQL'`

However, it is possible to prototype your application with SQLite, but it's a good habit to select a database system like PostgreSQL or MySQL if you are making a real-world project.

2. Check if your system has the database system installed.

If you are using SQLite, you don't need to do anything further. It's all set for you.

If you're using PostgreSQL on your local machine, download and install it from:

https://www.postgresql.org/download/

If you are planning to deploy your application to a server, check if your server supports the database system you want to work with.

If you're using MySQL on your local machine, download and install it from:

https://www.mysql.com/downloads/

3. Do the configuration for your database system:

```
...
    # PostgreSQL Database
    DB_CONFIG = {
        'host':     'localhost',
        'user':     'postgres',
        'password': 'db_password',
        'database': 'app_db',
        'port': '5432',
    }
```

```
...
```

When you install the Postgres Database on your system you will get the user and password for connecting to your database.

Usually by default, the host is `'localhost'`, and the port is `'5432'`, and the user is `'postgres'`.

You only need to set the password.

> You can do very similarly for MySQL database.

4. Check if the database API library is installed on your machine.

For PostgreSQL check: psycopg2

For MySQL check: mysql-connector-python

> That's it! It was all the configuration you need to do to set up your database.

Check Database

Now that you setup your database, run the following command to check the database:

```
(venv) python manage.py check-db
Database connection established successfully!
Database not initialized!
To initialize the database run the following command:
python manage.py init-db
```

> If you already initialized your database with Aurora framework, you will get a different message.

Initialize Database

To initialize your database, first check if you created all models you need for the first run.

After checking your models, to initialize the database use the following command:

```
(venv) python manage.py init-db
Initializing the database...
Creating the initial migration...
Database initialized successfully!
```

Now you are ready to use your models.

Migrate Database

AuroraSQL v0.6.0 is a beta version, and some features are in progress.

> The **`migrate-db`** command is currently unavailable, for this version.

> For now, you can create all your models first, then use the **`init-db`** command to initialize the database.

Database CRUD Methods

Aurora framework is designed to make CRUD applications easily and quickly.

The CRUD stands for **C**reate, **R**ead, **U**pdate, and **D**elete.

SQL uses CREATE, and INSERT keywords for Creating data.

SQL uses SELECT keyword for Reading data.

SQL uses ALTER, and UPDATE keywords for Updating data.

SQL uses DROP, and DELETE keywords for Deleting data.

> For simplicity, AuroraSQL uses **`create`**, **`read`**, **`update`**, and **`delete`** all in lowercase, in order to interact with the database.

AuroraSQL also uses a special kind of read methods called `exist` to check the data existence.

AuroraSQL provides the following methods:

- Exist Methods:
  - `_exist_fk` – Checks if a column is a foreign key.
  - `_exist_column` – Checks if a column exists.
  - `_exist_table` – Checks if a table exists.
  - `_exist_database` – Checks if a database (file – for SQLite) exists.

- Create Methods:
  - `create` – Inserts a single row into a table if exists.
  - `create_multi` – Inserts multi rows by looping the `create` method for a given list of data.
  - `_create_fk` – Adds a foreign key to an existing column for MySQL and PostgreSQL.
  - `_create_column` – Adds a new column to an existing table.
  - `_create_table` – Creates a table if not exists.
  - `_create_database` – Creates a new database (file – for SQLite).

- Read Methods:
  - `read` – For reading table rows, and sending them as SQL queries to the `Select` class of AuroraSQL based on a given where clause. For fetching data, you need to use its own methods:
    i. `first` – Fetches the first row as a dictionary matching the given SQL query.
    ii. `last` – Fetches the last row as a dictionary matching the given SQL query.
    iii. `all` – Fetches all the rows matching the given SQL query.
    iv. `count` – Counts the number of rows matching the given SQL query.
    v. `min` – Fetches the minimum of the first given column (must be of type int or float).
    vi. `max` – Fetches the maximum of the first given column (must be of type int or float).
    vii. `avg` – Fetches the average of the first given column (must be of type int or float).

viii. `sum` – Fetches the summary of the first given column (must be of type int or float).

- Update Methods:
  - `update` – Updates row(s) of a table based on a given where clause.
  - `_update_column` – Updates a column with new name, datatype and constraints.
  - `_update_table` – Renames a table if exists.

- Delete Methods:
  - `delete` – Deletes row(s) of a table based on a given where clause.
  - `_delete_fk` – Drops a foreign key from an existing table for MySQL and PostgreSQL.
  - `_delete_column` – Drops a column from a table.
  - `_delete_table` – Drops a table if exists.
  - `_delete_database` - Drops a database (Removes a database file for SQLite) if exists.

You cannot use `_update_column` and `_delete_column` methods with foreign key columns.

Important Note:

For security reasons, only `create`, `create_multi`, `read`, `update`, `delete` methods are available for your models. All `read` methods are also available. (`first`, `last`, etc.)

Danger!

If you are a pro user of SQL, and you know exactly what you are doing, you can still access to all methods somehow.

Somewhere in your controllers import and instantiate the Database class of AuroraSQL:

```
from aurora.SQL import Database
...
db = Database()
```

Now, you have access to all AuroraSQL methods.

> Please be super careful with methods starting with `_`, because they are low-level methods that can harm your database if you don't know how to use them.
>
> As an extra note, please never use these low-level methods in production code, only for development purposes.

As an example, let's add a user, and some notes in the `Index` controller of our `notes` app:

```
# Dependencies
...
from models import Users, Notes
...
    # GET Method
    def get(self):
        users = Users()
        user_data = {
            'username': 'admin',
            'email': 'admin@test.com',
            'password': '123456'
        }
        users.create(data=user_data)
...
```

> **Note:**
>
> AuroraSQL is an auto-commit SQL library. You don't need to save or commit a query manually.

Now, restart your application and access the following address:

http://127.1.1.1:5000/my-notes/

It will create, the defined user for you.

> **Security Alert:**
>
> Never store your passwords into database like what we did here. It's only for learning purpose.

Always hash your password before storing them into the database. You will learn how to do it in the Aurora Security section.

Now, let's create several notes for the created user:

```
# Dependencies
...
from datetime import datetime
...
    # GET Method
    def get(self):
        notes = Notes()

        notes_data = [
            {
                'user_id': 1,
                'title': 'First Note',
                'content': 'It is my first note',
                'date': datetime.now()
            },
            {
                'user_id': 1,
                'title': 'Second Note',
                'content': 'It is my second note',
                'date': datetime.now()
            },
            {
                'user_id': 1,
                'title': 'Another Note',
                'content': 'It is another note',
                'date': datetime.now()
            }
        ]

        notes.create_multi(data=notes_data)
...
```

In a real-world application, normally you don't insert data into your database like that.

It's much better to put it in a POST method, and use the form data comes from the user requests as your method data. Here we don't want to dive into that subject.

Now let's read the data we already inserted into our notes table:

```
...
    # GET Method
    def get(self):
        notes = Notes()

        print(notes.read().first())
        print(notes.read(where={'id':2}).first())
        print(notes.read().last())
        print(notes.read().all())
        print(notes.read(order_by={'id':'DESC'}).all())
        print(notes.read(cols=['title', 'content']).all())
        print(notes.read(cols=['id']).count())
        print(notes.read(cols=['id']).min())
        print(notes.read(cols=['id']).max())
        print(notes.read(cols=['id']).avg())
        print(notes.read(cols=['id']).sum())
...
```

Now, let's update the user we already created:

```
...
    # GET Method
    def get(self):
        users = Users()
        print(users.read(where={'id':1}).first())
        users.update(data={'username':'administrator'}, where={'id':1})
        print(users.read(where={'id':1}).first())
...
```

Security Alert:

If you don't provide a where argument to `update` method, you should provide another argument called `confirm` and set it to `True`. Normally the `confirm` is optional, but if you omit the where argument, it becomes required.

However, you should be super careful, because it will update all the rows of your database.

In a real-world application, normally you don't update your database data like that.

It's much better to put it in a PUT method, and use the form data comes from the user requests as your method data. Here we don't want to dive into that subject.

Now let's delete some note of our user:

```
...
    # GET Method
    def get(self):
        notes = Notes()
        print(notes.read().all())
        notes.delete(where={'id':3})
        print(notes.read().all())
...
```

Security Alert:

If you don't provide a where argument to `delete` method, you should provide the `confirm` argument and set it to `True`. Normally the `confirm` is optional, but if you omit the where argument, it becomes required.

However, you should be super careful, because it will delete all the rows of your database.

In a real-world application, normally you don't delete your database data like that.

It's much better to put it in a DELETE method, and use the form data comes from the user requests as your method data. Here we don't want to dive into that subject.

Custom SQL Queries

As I said before, if you are a pro user of SQL, and you know exactly what you are doing, you can still access to all methods somehow.

Sorry for being such a pain, but I have to alert you again:

> Please be super careful with methods starting with `_`, they are low-level methods that can harm your database if you don't know how to use them.
>
> As an extra note, please never use these low-level methods in production code, only for development purposes.

```
...
from aurora.SQL import Database
...
    # GET Method
    def get(self):
        db = Database()
        print(db.read(table='notes').all())
...
```

> Using AuroraSQL this way, you have to provide table name, because you are now accessing to all tables of your database, and you have to explicitly tells the AuroraSQL which table you want to interact with.

There is another method that you can access now for creating custom queries and interact with database API libraries directly. This method is called `query`.

Here is an example:

```
...
    # GET Method
    def get(self):
        db = Database()
        print(db.query('SELECT * FROM notes').fetchone())
        print(db.query('SELECT * FROM notes').fetchmany(2))
        print(db.query('SELECT * FROM notes').fetchall())
...
```

> Very Important Note:

If you are using AuroraSQL this way, be super careful to bind incoming data, because this way your SQL queries can be really insecure against SQL Injection attacks.

Here's an example of how to bind your data in a safe way:

SQLite

```
db.query('SELECT * FROM notes WHERE id=? AND user_id=?', ('1', '1')).fetchone()
```

MySQL / PostgreSQL

```
db.query('SELECT * FROM notes WHERE id=%s AND user_id=%s', ('1', '1')).fetchone()
```

Using AuroraSQL in the normal way, you don't have to worry about SQL Injection attacks and data binding, they will be filtered before querying the database automatically.

AuroraSQL Reference

AuroraSQL has many built-in features, that needs a separate documentation and make this documentation file so long and hard to read.

I will provide a separate documentation for AuroraSQL asap.

RESTFUL CRUD Methods

Representational state transfer (REST) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. (Read More)

In order to diminishing this tutorial, we don't dive into the REST APIs and RESTFUL apps.

Only keep it in mind that Aurora is designed to create RESTFUL web apps easily and efficiently.

Aurora provides the following REST methods that you can use inside of your controllers:

CRUD	HTTP	Aurora	Description (Aurora)
Create	POST	**post**	Used for creating (inserting) new data.
Read	GET	**get**	Used for reading (selecting) the data.
Update	PUT	**put**	Used for updating the data.
Delete	DELETE	**delete**	Used for deleting the data.

So far in this documentation, for simplicity, we did the CRUD methods of the database inside the get method. However, it's not what how Aurora framework is designed for.

I personally don't want to make this documentation long and hard to read. But let's take a quick example to see how a RESTFUL web app works.

First, let's create another controller called **`Notes`**, with all the available methods:

```
(venv) python manage.py create-controller
App Name: notes
Controller Name: Notes
Controller URL: notes-app
Methods (optional): post, get, put, delete
```

Here's the blueprint created for us:

```
# Dependencies
from aurora import Controller, View

# The controller class
class Notes(Controller):

    # POST Method
    def post(self):
        pass

    # GET Method
    def get(self):
        return 'Page content...'
```

```
    # PUT Method
    def put(self):
        pass

    # DELETE Method
    def delete(self):
        pass
```

Now, imagine we have some forms in some view files of our app, that provides the user the ability to create, update, and delete posts from the database. And finally, they can show all the data. And we have used JavaScript to send these forms to the server.

So, we should update our controller to handle the user requests:

```
# Dependencies
from aurora import Controller, View

# The controller class
class Notes(Controller):

    # POST Method
    def post(self):
        # Handle the requested form data
        TODO

        # Create the new data into database
        TODO

        # Send back JSON like result
        if True:
            return {
                'success': 'Success message!',
            }, 200
        else:
            return {
                'error': 'Error message!',
            }, 400
```

```
    # GET Method
    def get(self):
        # Read data from the database
        data = TODO

        # Send back the rendered template with the retrieved data
        return View(view='notes', data=data)

    # PUT Method
    def put(self):
        # Handle the requested form data
        TODO

        # Update the database
        TODO

        # Send back JSON like result
        if True:
            return {
                'success': 'Success message!',
            }, 200
        else:
            return {
                'error': 'Error message!',
            }, 400

    # DELETE Method
    def delete(self):
        # Handle the requested form data
        TODO

        # Delete data from the database
        TODO

        # Send back JSON like result
        if True:
            return {
                'success': 'Success message!',
            }, 200
```

```
        else:
            return {
                'error': 'Error message!',
            }, 400
```

Based on what we worked together so far, you should be able to create such an application simply with a little effort and time.

HTTP Errors

Another important part of RESTFUL frameworks is to show the users sufficient messages for their requests. If the requests come via JavaScript, you can send back the HTTP status code like what we did in the previous section.

However, there are times that the user requests come directly by the browser. Here's where the importance of handling user requests becomes clear.

As we saw already, using these methods, you don't have to explicitly handle user requests, the framework handles it for you automatically. As an example, try to send a post request to a Controller that doesn't support the post request. You will see a 405-error forbidden method message instead.

The question is where did these messages come from? The answer is via **`errors`** app.

You can simply, check the **`errors`** app controllers and views to see what's going on.

As I said previously in this documentation, almost everything in Aurora framework is an application. You can update the styles for these error messages and customize them for your project.

Only consider that these error messages will be rendered dynamically by the framework whenever they needed. If you want to see them in a normal situation, there is a possibility for that. First check if the **`DEVELOPMENT`** attribute of your **`config.py`** module is set to **`True`**.

You can visit the following addresses on your browser to see the error messages:

```
http://127.1.1.1:5000/errors/400/

http://127.1.1.1:5000/errors/403/
```

```
http://127.1.1.1:5000/errors/404/

http://127.1.1.1:5000/errors/405/

http://127.1.1.1:5000/errors/500/
```

If the **`DEVELOPMENT`** is set to False, all these URLs show the 404-error message, which is the goal of the final project for the real world. Because you don't want someone to see the **`Internal Server Error!`** message on your website.

There are more standard HTTP errors out there, and if you want to create new ones, you cannot do it by the framework. You have to do it manually. Look exactly at one of the controllers of this app. It's very similar to normal controllers of your apps.

Only remember to update the **`_controller.py`** of the **`errors`** app.

Here's a reference on these error messages:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses

If you want to explicitly raise an error inside your controllers, you can do it in two ways:

You can use aurora View method:

```
from aurora import View
...
return View(view="404", app="errors", code=404)
```

Or you can use the **`abort`** method:

```
...
from aurora.security import abort
...
abort(404)
```

Aurora Configuration

You can simply configure your Aurora project by modifying the **`config.py`** module of your project. But be careful to provide the valid data.

At the following we will discuss how to do it in a correct way:

1. Your project (root app) path:

```
ROOT_PATH = os.path.dirname(__file__)
```

> **Important Note:**
>
> Do not change or delete this line, because Aurora framework needs this attribute to interact with your project. If you want to use it somewhere in your code that's perfectly fine.

2. The error app:

```
ERROR_APP = 'errors'
```

> This is the error app used by all the other apps, please do not change or remove it.

3. The default app:

```
DEFAULT_APP = 'aurora'
```

> It contains the default app name to serve the '/' url. Change it to any app name you want.

4. Static files directory:

```
STATICS = 'statics'
```

> This attribute holds the name of your static directory. Change it to anything you like.

5. The development mode:

```
DEVELOPMENT = True
```

Security Alert:

Use this attribute with True only for development purposes. Remember to set it to False for the production. Using this attribute, you don't need to set the `DEBUG` attribute manually.

6. Database Configurations:

```
DB_SYSTEM = 'SQLite'
```

For database system attribute you can use either `'SQLite'`, `'PostgreSQL'`, `'MySQL'`.

Only Remember that this attribute is case sensitive.

We covered other database configurations so far and if you take quick look the file it will become pretty clear.

Just remember that Aurora framework only supports AuroraSQL as the DB-API engine.

7. Form Engine:

```
FORM_ENGINE = 'WTForms'
```

Currently Aurora framework only supports `'WTForms'` for handling HTML forms.

8. App Secret Key:

```
SECRET_KEY = random_string(24)
```

Security Alert:

Aurora uses this attribute for the security purposes like producing CSRF tokens. Please do not remove it.

9. Global Variables:

```
GLOBALS = {
    'key': 'Value',
}
```

Use the `GLOBALS` attribute to create global variables accessible inside your view (`.html`) files.

Here the `'key': 'Value'` is a dummy variable only for testing, do whatever you want with it.

Note:

Only remember that you have to use them in uppercase letters. You can create them in either lowercase or uppercase, but always use them uppercase. it is for preventing name confliction in your view files.

Here's an example for the dummy variable we already have:

```
...
{{ KEY }}
...
```

It will output the `Value` for us.

10. Auto Global Variables:

There are a few global variables created automatically by the Aurora framework that you can use inside of your view files.

They are:

```
ERROR_APP – Using this will output the errors app Base URL.
DEFAULT_APP – Using this output the default app Base URL.
STATICS – Using this attribute outputs the statics directory name.
```

In addition to these attributes, all of your app URLs becomes auto global. And you can access them using their name in uppercase. For the notes app here's an example:

```
...
<a href="/{{NOTES}}/">Notes App</a>
...
```

It will output:

```
<a href="/my-notes/">Notes App</a>
```

Do

```
<img src="/{{STATICS}}/notes/my-image.jpg">
```

Don't

```
<img src="/{{STATICS}}/{{NOTES}}/my-image.jpg">
```

It will output: `<img src="/statics/my-notes/my-image.jpg">`.

A file pointing to a directory on your disk that doesn't exists.

Important Note:

In order to avoid complexity, I recommend you to use the same name for your app names and base URLs. As you may notice all the local directories and files called after the app names.

The base URLs only used with HTML links on your website. For static files inside your view files, use the app name as simple strings.

Aurora Security

Introduction

As a professional user of technology for over a decade, I can say that in digital world, nothing is perfectly secured. There is not a single app that is %100 secured.

The reason is that what human make, human can also break.

If someone told you that their application is secured, ask them to what extent? And against what threats?

The security term is a concept that a developer should always consider as a priority.

For Aurora framework I have considered the security as one of my priorities, to make something secure enough to rely on. There are important security concepts that I have considered in the framework development.

There are a few important things you should consider as a user of this framework:

1. The AuroraSQL methods, except for the `query` method, are made to be automatically secured against SQL Injection attacks. You can pass any type of data to these methods and your values will be bound automatically before querying the database.

For the `query` method, it is the job of developer to bind the data before querying the database, which we discussed in this documentation.

2. Aurora framework forms are secured by a random string for every server request stored as a CSRF token, for avoid CSRF attacks. By default, it is enabled. Please never disable it, and never allow any of your forms to be submitted to the database without CSRF check.

3. For the sake of security, AuroraSQL low level methods are hidden from the normal users, and for advanced users I would say that please be super careful with these methods, and only use them in development code and testing.

4. Always hash your passwords before store into the database.

Aurora Security:

Aurora framework provides a library called `security` that you can take benefit from.

At the following we will discuss about its methods and how to use them.

Hashing Passwords

Use `hash_password` method to hash sensitive data before storing it into the database.

Here's an example:

```
from aurora.security import hash_password
...
'password': hash_password('123456')
...
```

This is a single way encryption, and if someone find it, they cannot guess or find the original data by decryption.

The only way to validate a hashed data is by using the `check_password` method:

```
from aurora.security import check_password
...
if check_password(hashed_password, requested_password):
    // Validation passed

else:
    // Validation failed
...
```

`check_password` method returns `true` if the validation is successful, otherwise it returns `false`.

The `hashed_password` argument can be a hashed password already stored in the database, and the `requested_password` can be a password requested by a user (like '123456').

URL Redirecting

Using aurora security module, you can restrict URLs for different type of users. Aurora security provides the following methods for redirecting:

- `abort` – Redirects to an errors app controller via a given status code:

```
from aurora.security import abort
...
abort(code=404)
```

The `404` is the default `code`, you can simply ignore it.

- `redirect` – Redirects to a URL using a given status code:

```
from aurora.security import redirect
...
redirect(url='/notes/my-notes/', code=302)
```

The `302` is the default `code`, you can simply ignore it.

- `redirect-to` – Redirects to a controller using the app name and a controller name:

```
from aurora.security import redirect_to
...
redirect_to(app="notes", controller="MyNotes", code=302)
```

The `controller` is optional, and the index controller of the app would be the default value.

The `302` is the default `code`, you can simply ignore it.

- `login_required` decorator – Redirects not logged-in users.

Imagine we have a controller in our app that we want only registered users have access to:

```
from aurora.security import login_required
...
class NewNote(Controller):

    # GET Method
    @login_required(app='users')
    def get(self):
        ...
```

Normally it can take several arguments:

```
login_required(app:str, controller:str=None, validate:str='user')
```

The `app` is required and takes the app name.

The `controller` is optional, and the index controller of the app would be the default value.

The `validate` is optional, and holds the name of the session and/or the cookie that we want to validate. The default value is `'user'`.

- `login_abort` decorator – Redirects logged-in users.

> It works in reverse vs the `login_required` method.

Validating Users

Aurora security module also provides the following methods to validate the users:

- `check_session` – Checks if a session exists.

```
from aurora.security import check_session
...
if check_session(name='user'):
    ...
```

> It returns `true` if the session exists, otherwise it returns `false`.

- `get_session` - Returns a session value via a given name.

```
from aurora.security import get_session
...
get_session(name='user')
```

- `set_session` - Sets a session via a given name and value.

```
from aurora.security import set_session
...
set_session(name='user', value='admin')
```

- `unset_session` - Unsets a session via a given name.

```
from aurora.security import unset_session
...
```

```
unset_session(name='user')
```

- `check_cookie` – Checks if a cookie exists.

```
from aurora.security import check_cookie
...
if check_cookie (name='user'):
    ...
```

It returns `true` if the cookie exists, otherwise it returns `false`.

- `get_cookie` - Returns a cookie value via a given name.

```
from aurora.security import get_cookie
...
get_cookie(name='user')
```

- `set_cookie` - Sets a cookie via a given name and value.

```
from aurora.security import set_cookie
...
set_cookie(name='user', value='admin')
```

It optionally can take two other arguments:

```
set_cookie(name:str, value:str, data:dict={}, days:int=30)
```

The `days` argument takes the number of days that before the cookie expires.

The `data` argument can take a dictionary of data that specifies the behavior of the method and how to return the response after setting the cookie. Here's an example:

```
data = {'type': 'text', 'response': 'Cookie is set.'}
set_cookie(name='user', value='admin', data=data)
```

It can take the following data:

Type	Description	Example

`'text'`	Returns plain text as response after setting the cookie. (This is the default value)	`data = {`
		`'type': 'text',`
		`'response': 'Cookie is set.'`
		`}`

`'json'`	Returns json data as response after setting the cookie.	`data = {`
		`'type': 'json',`
		`'response': {...}`
		`}`

`'redirect'`	Redirect to a URL after setting the cookie.	`data = {`
		`'type': 'text',`
		`'response': '/notes/my-notes/'`
		`}`

`'render'`	Renders a template as response after setting the cookie. (Takes the template path as value.)	`data = {`
		`'type': 'text',`
		`'response': '/notes/notes.html'`
		`}`

- `unset_cookie` - Unsets a cookie via a given name.

```
from aurora.security import unset_session
...
unset_cookie(name='user')
```

It optionally can take another argument:

```
unset_cookie(name:str, data:dict={})
```

The `data` argument can take a dictionary of data that specifies the behavior of the method and how to return the response after unsetting the cookie.

Its behavior is just like the data of the set_cookie method. The only difference is that it returns the response after **unsetting** the cookie.

Free Notes Project

I personally believe that the best way to learn something is by applying it in action.

You can find almost all we have done in this documentation in a sample application called "Free Notes". I created this application for you and you can access to the full source of it via the following GitHub repository:

```
https://github.com/heminsatya/free_notes
```

It is an open-source project with MIT license. Do whatever you want with it!

Contribution

The term security is much longer than what we discussed here.

If you saw any issues, please let me know. I appreciate any comments and suggestions.

About the Author

I'm Hemin Satya, a freelance programmer, someone who love writing code.

Aurora is my first open-source project, I hope you like it. I have many ideas and codes that I really love to share with you.

There are many things about security and other programming concepts that I still don't know. But I'm really enthusiastic to learning them. If you are the person who knows them, please share them with me.

I really want to make this framework as secure as possible, as simple as possible, as useful as possible. But it's not the job of a single person. I hope someday in a near future there will be a community that work together on this framework to make it something really magnificent for all of us.

I appreciate any comments, observations and suggestions. Use this framework as it's yours and use it for whatever purpose you want. I hope it serves you as it serves me.

I really appreciate the precious time you spent reading this document. I hope I have been able to do something useful for you that is worth your past time.

Sincerely yours.

Special Thanks

It was almost about two years ago, that one of my brothers' friends, told me why you don't take a verified online course in Computer Science.

He said, there are really great stuff online that you can use. And he introduced me the edX courses on computer science. I instantly fell in love with courses of HarvardX, an online institute of Harvard University that represents free courses in programming and computer science.

Since then, I have been taking HarvardX online courses. Honestly, I can't thank enough my brother's friend to introduce me these courses.

I believe HarvardX and CS50 teachers are the best in the world. I feel so lucky to have enrolled to these courses.

Here I want to thank all the staff behind the edX, HarvardX and specially CS50. I really appreciate all of them.

This short time that I've been your student, undoubtedly have been the best learning experience of my life. I have learned so much in a short time all thanks to you.

Special thanks to professor David J. Malan, the greatest teacher in the world.

I'm one of millions of your students. You don't even know me. But you are the hero of my life.

I dedicate this framework to you.