Aurora Framework

v0.9.3 beta

Hemin Satya <github.com/heminsatya>

6/11/2023

Table Of Contents

Introduction

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

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

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

Aurora is written in Python, and partially have used Flask.

Why Python

As a programmer who have had over a decade experience in programming in different platforms and 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 already have made several closed-source web apps in PHP and Node.js; They both are great.

I believe every programming language has its cons and pros. But, I found Python very interesting and decided to create this framework for it, and since I've never done an open-source project, I made it under the MIT license so everyone can use it.

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 other 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, fast, simple and flexible. With flask you can create simple and also 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

For years I had used many web frameworks such as Laravel (PHP) and Express.js (Node.js), and also have made a closed-source framework purely written in PHP, for making web apps.

With that previous experience, I decided to make a web framework for python. First, I decided to write it purely in python, and don't depend on any packages.

What I have learned so far is that unlike PHP, Python is a cross-platform programming language, and isn't specifically made for the web.

It takes considerably more time and effort to handle routing and requests.

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 really great.

> However, to be honest because of the custom MVT structure in some of these frameworks, I found it is so complicated to make and maintain medium size up to large scale applications.
>
> For small applications and somehow medium size applications 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 even that makes your code even more messy.

I felt more comfortable and freedom with Flask. I found that Flask is a great 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 pure python. It uses Flask for routing, request handling and some other stuff.

With Aurora, you can create maintainable web applications more easily, and you can manage large-scale web applications very simply.

Aurora has a clean MVC architecture with a great support for popular 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

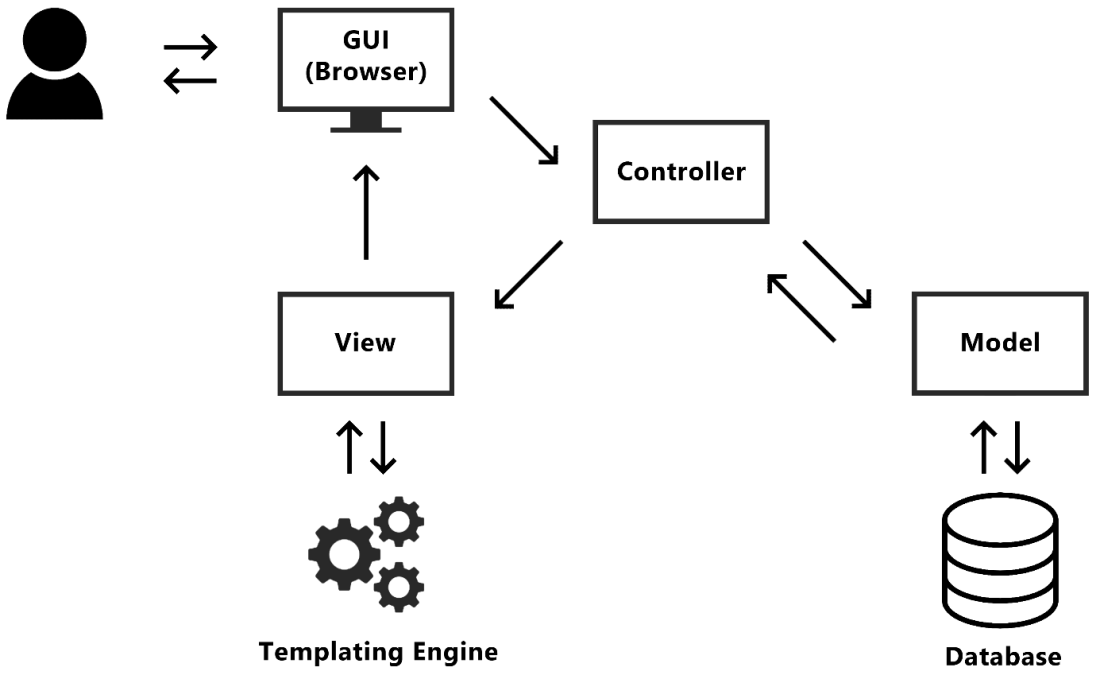
MVC is a software design pattern, which separates an application into three components: **Model**, **View** and **Controller**. It also describes the interactions between them.

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 commands 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 commands 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** (**M**odel **V**iew **T**emplate) 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.

Aurora MVC:

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. (Create)

Model can retrieve data from the database. (Read)

Model can modify the database. (Update and Delete)

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:

```
$ python3 --version
```

Aurora framework needs Python version 3 or higher to run.

For windows users use `py` instead of `python3` for all CLI commands in this documentation.

If you don't have Python, first download & install it from its official website:

https://www.python.org/

Installation

Now that you have Python installed on your machine, you can install Aurora framework using the following command:

```
$ 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/

Dependecies

Aurora uses several dependencies to run and work correctly:

- **Flask**:

Aurora needs flask to be installed on your environment. You can install it by:

```
$ pip install flask
```

- **WTForms**:

Aurora uses WTForms package to produce CSRF tokens. You can install it by:

```
$ pip install WTForms
```

- **flask_compress**:

Aurora uses flask_compress package for compressing the responses. Install it by:

```
$ pip install flask_compress
```

All At Once

You can also install all Aurora dependencies at once by running:

```
$ pip install flask WTForms flask_compress
```

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. Create a python virtual environment in the same path the project directory exists:

Linux / Mac

```
$ python3 -m venv venv
```

Windows

```
$ py -m venv venv
```

3. Activate the virtual environment:

Linux / Mac

```
$ source venv/bin/activate
```

Windows

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

If you are using Git Bash as your CLI program you shoud use:

```
$ . 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:

Linux / Mac

```
(venv) python3
```

Windows

```
(venv) py
```

Then run:

```
>>> from aurora import init
>>> init.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:

Linux / Mac

```
(venv) python3 -m app
```

Windows

```
(venv) py -m app
```

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 like this:

```
/_migrations
/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 **`_migrations`** is a system directory controlled by the framework used for database migrations and will be discussed in the following of this documentation.
- 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 routers. Here **`ControllerName.py`** is a controller class for the current app.

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

Important Note:

All modules start with a single `_` are system modules created by 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 which is used for managing the database of the root app and all its 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.
- The `statics` directory, contains all static files of your child apps, that you can 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 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 app routers for your project.
- 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 child apps controllers. 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 change or remove lines with `#do-not-change-me` flag in 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 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 -m app
```

You can add your own apps, then the root app serves them.

Child Apps

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

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

In order to clarifying the basic structure of an Aurora project, the project skeleton from the previous section, was simplified.

The one you have got after initializing the root app, is a little larger.

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

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

> **Caution!**
>
> 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 run. You can use it as a reference.

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) python3 -m manage delete-app
```

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

> 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 program.

To see a list of all available CLI commands run:

```
(venv) python3 -m manage --help
```

The general syntax is:

```
(venv) python3 -m manage COMMAND
```

> For windows users use `py` instead of `python3` for all CLI commands in this documentation.

Here is a table of all available commands and a description about what they can do:

Command	Description
`create-app`	Creates a new app if not exist with some default components.
`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`	Deletes an existing model.

`check-db`	Checks the models and database for errors and changes.
`init-db`	Initializes the database for the first time if not initialized already.
`migrate-db`	Migrates the model changes to the database.
`repair-db`	Is used for renaming the existing model columns.
`reset-db`	Is used for resetting the database, based on the current models.

Notice:

We will fully discuss all of these commands and how to use them at the following of this documentation.

Important Note:

Aurora CLI commands are only available in development (**`DEVELOPMENT = True`**).

Manage Apps

You can find a list of all your apps routers in the `_apps.py` system module.

Create App

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

```
(venv) python3 -m manage create-app
```

It will prompt you for some required arguments.

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

```
(venv) python3 -m manage create-app
App Name: notes
App URL: notes
```

For simplicity, it is recommended to use the same name for your App Name & 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 App URL are: `a-z`, `-` (Notice that the `_` character is not valid.)
- All characters must be in English lowercase alphabetical letters.

The above command creates the following components for the newly created app:

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

It also updates the `_apps.py` module for you, so 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) python3 -m manage delete-app
```

It will prompt you for some required arguments.

As an example, here we try to delete an existing app called `notes`:

```
(venv) python3 -m manage 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) python3 -m manage 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) python3 -m manage 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`**, **`0-9`**, **`-`**, **`/`**, **`<`**, **`:`**, **`>`**
- Valid Methods are: **post**, **get**, **put**, **delete**

> If you leave the methods empty, the framework by default 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
```

Don't

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

The framework considers this URL as `/notes/notes/index/`, which doesn't exists!

Do not add the App URL to the Controller URL. It will be handled by the framework.

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. Only remember, not 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 can use the `View` class to return rendered HTML to the browser:

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

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

> Note that you don't need to provide the `.html` for the view, only the view name is enough.

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:

```
# Dependencies
from aurora.helpers import controller

# Controllers routes
controllers = [
    controller(name='Index', url='', methods=['GET', 'POST']),
]#do-not-change-me
```

For example, you can remove the post method:

```
...
    controller(name='Index', url='', methods=['GET']),
...
```

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

Then, you also need to 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 in the `_controllers.py` module:

```
...
    controller(name='Greet', url='greet'),
    controller(name='Greet', url='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) python3 -m manage 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) python3 -m manage 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) python3 -m manage 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.

Templating Inheritance

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

This view file is like a class (blueprint) that all other views can inherit.

If you open the `index.html` file you will see that the framework have done the templating inheritance and some other tasks 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. However, 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=200)
...
```

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

For whatever reason, if you want to return other app views, 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) python3 -m manage 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`** 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 %}{% 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.0.7.0 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, install it using 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) python3 -m manage 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) python3 -m manage 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 them 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 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 aurora.security 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) python3 -m manage 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) python3 -m manage create-model
```

It will prompt you for the model name, then if not exists it creates the model blueprint for you.

Here's an example:

```
(venv) python3 -m manage 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.

Safe Typing

Previously in Aurora you have to use exact SQL datatypes in order to run your SQL queries correctly. Using Aurora v.0.7.0 you can type safely with standard python datatypes plus some added datatypes without worrying about SQL syntax errors.

However, it is an option added to Aurora configuration and if you are a pro user of SQL you can still use traditional SQL datatypes.

However, I recommend you to use the Safe Typing (`SAFE_TYPE`) configuration set to True, because it uses standard python datatypes and reduces the probability of SQL syntax errors.

In the following we take a look at the model created for us with both options.

- `SAFE_TYPE = False`:

```
# Dependencies
from aurora import Model
```

```
# The model class
class Notes(Model):

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

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

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

        # Repair the database
        self.repair = {}
```

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 AuroraSQL section.

- **`SAFE_TYPE = True`**:

```
# Dependencies
from aurora import Model

# The model class
class Notes(Model):

    # Model columns
    id = Model.column(datatype='int', size='lg', not_null=True)

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

        # Override the parent class default properties
```

```
        self.table = 'notes'
        self.primary_key = 'id'

        # Repair the database
        self.repair = {}
```

Here we have used the standard python datatype `int`. The size parameter is added by the framework and will be discussed in its section.

You can change the default table name for the model.

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 it in the database. If you want to rename it, rename both column name and `primary_key` value.

The `self.repair` property can be used for repairing (renaming) the existing column after initialization and migration which will be discussed in its section.

Caution!

Before the initialization of your database decide to choose a Safe Typing setup once, and never changes it again, because it can harm your database.

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.

Using AuroraSQL with `SAFE_TYPE = True`, you can use standard python datatypes and the framework converts them to equivalent SQL datatypes based on the database system you used.

Available safe types in AuroraSQL are: `str`, `int`, `float`, `bool`, `date`, `time`, `datetime`

Using `str`, `int`, and `float` you can use the optional `size` parameter.

Using `float` datatype, you can use another optional `scale` parameter which relates to the decimal part.

These parameters will be discussed in its section.

Using AuroraSQL with `SAFE_TYPE = False`, you can use standard SQL datatypes.

The only thing you should consider is to choose correct datatype for every database:

You can learn more about available datatypes for SQL databases at SQL Reference section.

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 tables. If you provide an integer primary key, it will become an auto increment field automatically.

Traditionally with MySQL and Postgres databases, you have to explicitly do it yourself. In which each one provides a different syntax.

Using AuroraSQL, your primary keys of type integer (int) will become auto increment fields automatically, for all three databases.

Independent from which configuration you choose for `SAFE_TYPE`, AuroraSQL datatypes are not case sensitive, you can use lowercase, or uppercase.

They will be converted to uppercase automatically. Only be careful to type them correctly.

Other parameters are the same for both `SAFE_TYPE` configuration:

- `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`

Which the `CASCADE` is the default value.

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

- `SAFE_TYPE = False`:

SQLite:

```
...
    # 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='numeric', not_null=True)
```

```
...
```

MySQL:

```
...
    # Model columns
    id = Model.column(datatype='integer', not_null=True)
    user_id = Model.column(datatype='integer', related_to='Users')
    title = Model.column(datatype='varchar(50)', not_null=True)
    content = Model.column(datatype='varchar(500)', not_null=True)
    date = Model.column(datatype='datetime', not_null=True)
...
```

Postgres:

```
...
    # Model columns
    id = Model.column(datatype='integer', not_null=True)
    user_id = Model.column(datatype='integer', related_to='Users')
    title = Model.column(datatype='varchar(50)', not_null=True)
    content = Model.column(datatype='varchar(500)', not_null=True)
    date = Model.column(datatype='timestamp', not_null=True)
...
```

- `SAFE_TYPE = True`: (SQLite, MySQL, Postgres)

```
...
    # Model columns
    id = Model.column(datatype='int', size='lg', not_null=True)
    user_id = Model.column(datatype='int', size='lg', related_to='Users')
    title = Model.column(datatype='str', size='xs', not_null=True)
    content = Model.column(datatype='str', size='sm', not_null=False)
    date = Model.column(datatype='datetime', not_null=True)
...
```

The `related_to` argument, creates a `one-to-one` or `one-to-many` relationship, automatically, with the primary key of the given model name (Users here).

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 believe, it makes your code so complicated.

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 one table in your database.

Using Models

Before the usage of models in our controllers we need to initialize the database. But for now, we only import our models 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) python3 -m manage 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.

AuroraSQL

Introduction

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
MySQL	mysql.connector
Postgres	psycopg2

AuroraSQL has a built-in support for major SQL Database Systems, SQLite, MySQL, and Postgres, 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.

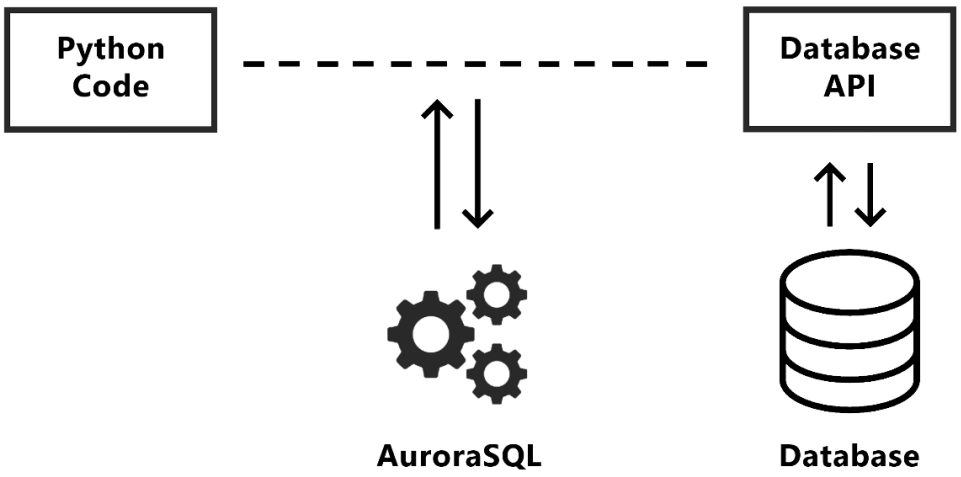
You don't need to install `sqlite3` module. It is included in the standard library since python 2.5.

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 MySQL or Postgres.

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.

AuroraSQL 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, MySQL, and Postgres.

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'`**, **`'MySQL'`**, **`'Postgres'`**

However, it is possible to prototype your application with SQLite, but it's a good habit to select a database system like MySQL or Postgres 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 Postgres 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:

```
...
    # Postgres 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. Set the `SAFE_TYPE` configuration:

If you are a SQL pro set it to False and use standard SQL datatypes based on which database system you work with.

If you want to focus on the business logic and don't get confused with different datatypes for different database systems, set it to True and use standard python datatypes. The framework handles the datatypes for you and converts them to equivalent SQL datatypes.

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

For MySQL check: mysql-connector-python

For Postgres check: psycopg2

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

Check Database

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

```
(venv) python3 -m manage check-db
...
Database connection not found!
To create a connection and initialize the database run the following command:
python3 -m manage init-db
```

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

In general, this command checks your models and database for errors and changes.

Initialize Database

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

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

```
(venv) python3 -m manage init-db
...
Initializing the database...
Creating the initial migration...
Database initialized successfully!
```

Now you are ready to use your models.

Migrate Database

AuroraSQL v0.7.0 now fully supports the **`migrate-db`** command.

It migrates the model changes to the database.

```
(venv) python3 -m manage migrate-db
```

This command:

- Checks for deleted models and drops their corresponding tables from the database.
- Checks for renamed tables for existing models, and renames them in the database.
- Checks for added models and creates their corresponding tables in the database.
- Checks for deleted columns and drops them for corresponding tables from the database.
- Checks for added columns and adds them to the corresponding tables in the database.

- Checks for modified columns and updates them for corresponding tables in the database.
- Checks for modified primary key and updates it for corresponding table in the database.

Very Important Note:

Use `migrate-db` with extra caution. However, I fully tested it for all three database systems, but there is still a probability of corrupting your database if you provide incorrect `default` and `check` constraints.

Repair Database

For renaming the primary key you can simply rename its attribute, then set `self.primary_key` to the new name, and finally run `migrate-db` command. It will be renamed safely without any data loss.

Except for the primary key, you cannot use `migrate-db` to rename columns.

However, you can remove a column and run `migrate-db`, then add the column again and rerun the `migrate-db`. But it will cause the data loss for that column.

A much safer way to rename your columns is to use the `repair-db` command.

Before running this command, you need to do several steps:

- First, you need to make sure that there are no errors in your models and all changes migrated to the database. You can simply check it via `check-db` command.
- Then, modify the `self.repair` attribute for the desired model.

Here's an example: (For Notes model)

```
...
        # Repair the database
        self.repair = {
            'content':'description'
        }
...
```

The `'content'` here is the current column name, and `'description'` is the name that you want to rename the column to. You don't need to rename the column attribute itself.

- Finally, run the following command:

```
(venv) python3 -m manage repair-db
```

If the `'description'` doesn't exist in the model, it will rename the column in the database without data loss, and also updates the model and renames the attribute.

Reset Database

Sometimes you may want to reset your database based on the current models. For examples changing the database system.

You can use the `reset-db` command for this purpose.

DANGER!

By resetting the database, you will lose all migrations and the data inserted into the database permanently.

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.

Primary AuroraSQL CRUD Methods:

- `create` – Inserts a single row into an existing table.
- `create_multi` – Inserts multiple rows by looping the create method for a given list of data.
- `read` – Reads table rows. For fetching data, you need to use its own methods:
  - `first` – Fetches the first row as a dictionary matching the given SQL query.
  - `last` – Fetches the last row as a dictionary matching the given SQL query.
  - `all` – Fetches all the rows matching the given SQL query.
  - `count` – Counts the number of rows matching the given SQL query.
  - `min` – Fetches the minimum of the first given column (must be of type int or float).
  - `max` – Fetches the maximum of the first given column (must be of type int or float).
  - `avg` – Fetches the average of the first given column (must be of type int or float).
  - `sum` – Fetches the summary of the first given column (must be of type int or float).
- `join` – Joins related tables. For fetching data, you need to use its own methods:
  - `first` – Fetches the first row as a dictionary matching the given SQL query.
  - `last` – Fetches the last row as a dictionary matching the given SQL query.
  - `all` – Fetches all the rows matching the given SQL query.
  - `count` – Counts the number of rows matching the given SQL query.
- `update` – Updates row(s) of a table based on a given where clause.
- `delete` – Deletes row(s) of a table based on a given where clause.

Using the `join` method is the easiest and best way to create **many-to-many** relationships.

Using the `join` method indirectly, you need to use the Model name before the column name for all parameters. (ex. `Model1.column`)

You can also use name aliases for columns with the same name for related Models. This way you can prevent name confliction. (ex. `Model1.column as custom_column`)

Besides these primary methods, AuroraSQL has many more built-in methods and features that you can check at AuroraSQL Reference.

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/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.

Safe Typing Reference

Available safe types in AuroraSQL are:

```
str         # Standard python datatype for storing string like values
int         # Standard python datatype for storing integers
float       # Standard python datatype for storing floating point numbers
```

```
bool      # Standard python datatype for storing booleans (True, False)
date      # Non-standard python datatype for storing date (YYYY-MM-DD)
time      # Non-standard python datatype for storing time (hh:mm:ss)
datetime  # Non-standard python datatype for storing date & time (YYYY-MM-DD hh:mm:ss)
```

Using str, **`int`**, and **`float`** you can use the optional **`size`** parameter.

Using **`float`** datatype, you can use another optional **`scale`** parameter which relates to the decimal part.

`str`

size	SQLite	MySQL	Postgres
xs	TEXT	VARCHAR(50)	VARCHAR(50)
sm	TEXT	VARCHAR(500)	VARCHAR(500)
md	TEXT	VARCHAR(5000)	VARCHAR(5000)
lg	TEXT	TEXT	VARCHAR(65535)
xl	TEXT	LONGTEXT	TEXT

`int`

size	SQLite	MySQL	Postgres
xs	INTEGER	TINYINT	SMALLINT
sm	INTEGER	SMALLINT	SMALLINT
md	INTEGER	MEDIUMINT	INTEGER
lg	INTEGER	INTEGER	BIGINT
xl	INTEGER	BIGINT	BIGINT

`float`

size	SQLite	MySQL	Postgres

xs	REAL	DECIMAL(4, *scale*)	DECIMAL(4, *scale*)
sm	REAL	DECIMAL(8, *scale*)	DECIMAL(8, *scale*)
md	REAL	DECIMAL(12, *scale*)	DECIMAL(12, *scale*)
lg	REAL	DECIMAL(16, *scale*)	DECIMAL(16, *scale*)
xl	REAL	DECIMAL(20, *scale*)	DECIMAL(20, *scale*)

`bool`

datatype	SQLite	MySQL	Postgres
bool	NUMERIC	BOOLEAN	BOOLEAN

`date`, `time`, `datetime`

datatype	SQLite	MySQL	Postgres
date	NUMERIC	DATE	DATE
time	NUMERIC	TIME	TIME
datetime	NUMERIC	DATETIME	TIMESTAMP

Database Backups

There are tons of rules and exceptions in Database Systems, I fully tested all AuroraSQL methods for all the Database Systems mentioned above, and have used migrations for any possible situations I knew. However, there is a small possibility for data loss in migrations.

Hence I strongly recommend to use all CLI commands only in development, and take frequent backups before performing them to prevent any data loss.

Here are some useful links you can use to learn how to backup and restore your database:

SQLite:	https://www.sqlite.org/backup.html
MySQL:	https://dev.mysql.com/doc/refman/8.0/en/backup-and-recovery.html

Postgres: https://www.postgresql.org/docs/8.1/backup.html

SQL Reserved Keywords

AuroraSQL has used escape characters for all three database systems. So, you can use any English words as identifiers without worrying about name confliction with SQL reserved words.

Although I fully tested all methods for all three databases but still there is a probability of name confliction with SQL reserve words.

Hence, I recommend you to test all of your models in development before deployment to a real server. Alternatively, you can ignore these reserved keywords.

Here you can find the reserved keywords for all three database systems in AuroraSQL:

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

https://dev.mysql.com/doc/refman/8.0/en/keywords.html

https://www.postgresql.org/docs/8.1/sql-keywords-appendix.html

---

SQL Reference

Visiting the following links, you can read more about available datatypes for SQL databases:

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

https://dev.mysql.com/doc/refman/8.0/en/data-types.html

https://www.postgresql.org/docs/9.5/datatype.html

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

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

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

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) python3 -m manage 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 users appropriate messages for their requests. If the requests come via JavaScript, you can send back the HTTP status code like what we did in a 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 them 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 as a page on your website.

There are more standard HTTP errors out there, and if you want to create new ones, you can

simply do it via the `create-controller` command. And for App Name type `errors`, and for App URL type the status code, and leave the methods empty. Only remember to set the status code in the `View` class in the controller and connect it to a proper view file.

Here's a reference on HTTP 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 the `View` class:

```
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(code=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'`, `'MySQL'`, `'Postgres'`.

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 Darabase API engine.

7. Safe Typing:

```
SAFE_TYPE = True|False
```

Using `AuroraSQL` with `SAFE_TYPE = True`, you can use standard python datatypes and the framework converts them to equivalent SQL datatypes based on the database system you used.

Using `AuroraSQL` with `SAFE_TYPE = False`, you can use standard SQL datatypes. The only thing you should consider is to choose correct datatype for every database:

8. Form Engine:

```
FORM_ENGINE = 'WTForms'
```

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

9. 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.

10. 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.

11. 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 a top priority, to make it 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 have discussed in this documentation.

2. Aurora framework forms are secured by a random string for every server request stored as a CSRF token, to 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 a CSRF check.

3. For the sake of security, AuroraSQL low level methods are hidden for normal users, and for advanced users, they need to 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.

Password Security

Password Hashing

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 `validate_password` method:

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

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

`validate_password` method returns `true` if the validation is successful, otherwise 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').

Password Strength

You can use **`password_strength`** method to check the strength of a given password.

```
password_strength(password:str, length:int=8, digit:bool=True, uppercase:bool=True,
lowercase:bool=True, symbol:bool=True)
```

With all parameters provided:

> **The password should:**
>
> - Be greater than or equal to `length`
> - Contain one digit or more
> - Contain one uppercase letter or more
> - Contain one lowercase letter or more
> - Contain one symbol or more

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, check:Union[str, list]='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 `check` is optional, and holds the name of the session and/or the cookie that we want to validate. The default value is `'user'`. (It can also validates a list of values)

- `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='user', 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.

IP Methods

- `server_ip` - Returns the server IP

```
from aurora.security import server_ip
...
print(server_ip())
```

- `client_ip` - Returns the client IP

It takes an optional parameter called `strict`, which If set to `True`, returns the client's public IP, not a private IP behind a proxy.

```
from aurora.security import client_ip
...
print(client_ip(True))
```

AuroraSQL Reference

AuroraSQL uses **`create`**, **`read`**, **`update`**, and **`delete`** keywords 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
- Create Methods
- Read Methods
- Update Methods
- Delete Methods

For all examples in this section, we have used AuroraSQL class directly:

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

Exist Methods

Exist methods are a special kind of read methods that check data existence.

`_exist_fk`

- Checks if a column is a foreign key,
- Takes two parameters.
- Required **`table`** parameter holds the table name.

- Required `column` parameter holds the column name.
- Returns `True|False`.

```
db._exist_fk(table='notes', column='user_id')
```

_exist_column

- Checks if a column exists.
- Takes two parameters.
- Required `table` parameter holds the table name.
- Required `column` parameter holds the column name.
- Returns `True|False`.

```
db._exist_column(table='notes', column='title')
```

_exist_table

- Checks if a table exists.
- Takes one parameter.
- Required `table` parameter holds the table name.
- Returns `True|False`.

```
db._exist_table(table='notes')
```

_exist_database

- Checks if a database (file – for SQLite) exists.
- Takes one parameter.
- Required `database` parameter holds the database name (file name for SQLite).

- Returns `True|False`.

```
db._exist_database(database='app.db')
```

Create Methods

`create`

- Inserts a single row into an existing table.
- Takes two parameters.
- Required `table` parameter holds the table name.
- Required `data` parameter holds a dictionary of key & value pairs. Keys represent the columns and values represent the data meant to be inserted into the table.
- Returns the inserted row id.

```
data = {
    'username': 'admin',
    'email':    'admin@test.com',
    'password': '123456'
}
db.create(table='users', data=data)
```

`create_multi`

- Inserts multi rows by looping the `create` method for a given list of data.
- Takes two parameters.
- Required `table` parameter holds the table name.
- Required `data` parameter holds a list of dictionaries of key & value pairs. Keys represent the columns and values represent the data meant to be inserted into the table.
- Returns the last inserted row id.

```
data = [
    {
        'username': 'user-1',
        'email':    'user-1@test.com',
        'password': '363454556'
    },
    {
        'username': 'user-2',
        'email':    'user-2@test.com',
        'password': '4345345345345'
    },
    {
        'username': 'user-3',
        'email':    'user-3@test.com',
        'password': '3476373675'
    },
]
db.create_multi(table='users', data=data)
```

_create_fk

- Adds a foreign key to an existing column for MySQL and Postgres.
- Can take up to seven parameters.
- Required **`table`** parameter hold the table name.
- Required **`column`** parameter holds the column name.
- Required **`r_table`** holds the reference table name.
- Required **`r_column`** parameter holds the reference column name.
- Optional **`fk_symbol`*** parameter holds the foreign key symbol name.
- Optional **`on_update`**** parameter represents the ON UPDATE statement.
- Optional **`on_delete`**** parameter represents the ON DELETE statement.
- Returns **`True|False`**.

* If you do not provide the `fk_symbol` it will uses the following syntax or the symbol:

```
fk_symbol = f'fk_{table}_{r_table}'
```

** The default value for `on_update` & `on_delete` param would be '`CASCADE`'.

Available values are: '`RESTRICT`', '`CASCADE`', '`SET NULL`', '`NO ACTION`', '`SET DEFAULT`'

Note:

The `_create_fk` method doesn't work with SQLite.

```
db._create_fk(table='notes', column='user_id', r_table='users', r_column='id')
```

_create_column

- Adds a new column to an existing table.
- Takes up to four parameters.
- Required `table` parameter holds the table name.
- Required `column` parameter holds the column name.
- Required `datatype`* parameter holds the datatype of the column.
- Optional `constraints` parameter holds the column constraints as a string type statement.
- Returns `True|False`.

* The developer methods (methods start with _) follow the standard SQL datatypes and rules.

Hence the datatype here is not the `AuroraSQL` safe types but the SQL datatypes which is slightly different for each database system.

```
db._create_column(table='notes', column='new', datatype="INTEGER")
```

_create_table

- Creates a table if not exists.
- Can take up to eight parameters.
- Required **`table`** parameter holds the table name.
- Required **`col_type`** parameter holds a dictionary of key & value pairs, in which each key represents a column name and the value represents the datatype of the column.
- Optional **`primary_key`*** holds the PRIMARY KEY name.
- Optional **`unique`** parameter represents the UNIQUE constraint, and can take a list of the column names that you want to be unique.
- Optional **`not_null`** parameter represents the NOT NULL constraint, and can take a list of the column names that you want not to be null.
- Optional **`default`** parameter represents the DEFAULT constraint, and can take a dictionary of the column names as keys and default values.
- Optional **`check`** parameter represents the CHECK constraint, and can take a dictionary of the column names as keys and check rules.
- Optional **`foreign_key`** parameter holds a dictionary with a special syntax** for creating the table foreign key(s).
- Returns **`True|False`**.

```
col_type = {
    'id':'INTEGER',
    'title': 'TEXT'
}
db._create_table(table='new', col_type=col_type, primary_key='id')
```

* For simplicity, AuroraSQL only supports on PRIMARY KEY for each table.

** Here is an example of foreign_key syntax:

```
foreign_key = {
  'user_id': {
      'r_table': 'users',
```

```
        'r_column': 'id',
        'on_update': RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT,
        'on_delete': RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT,
    },
}
```

`_create_database`

- Creates a new database (file – for SQLite).
- Takes one parameter.
- Required **`database`*** parameter holds the database name.
- Returns **`True|False`**.

> * For SQLite the name should be a file name with **`.db`**, **`.sqlite`**, or **`.sqlite3`** extension.

```
# MySQL & Postgres
db._create_database(database='test_db')

# SQLite
db._create_database(database='test.db')
```

Read Methods

`read`

- Reads table rows.
- Takes up to 7 parameters.
- Required **`table`** parameter holds the table name that you want to read data from.
- Optional **`cols`** parameter can hold a list of column names that you want to read. (The default value would be ['*']).

- Optional **`where`** parameter can hold a dictionary of column names as keys and values to be check as the WHERE statement. (You will learn more about this param at the following)
- The optional **`order_by`** parameter can hold a dictionary of column names as keys and **`'ASC'`** or **`'DESC'`** values as the ORDER BY statement.
- The optional **`group_by`** parameter can hold the name of a column and represents the GROUP BY statement.
- The optional **`limit`** parameter represents the LIMIT statement.
- The optional **`offset`** parameter represents the OFFSET statement.
- For fetching data, you need to use its own methods:
    - **`first`** – Fetches the first row as a dictionary matching the given SQL query.
    - **`last`** – Fetches the last row as a dictionary matching the given SQL query.
    - **`all`** – Fetches all the rows matching the given SQL query.
    - **`count`** – Counts the number of rows matching the given SQL query.
    - **`min`** – Fetches the minimum of the first given column (must be of type int or float).
    - **`max`** – Fetches the maximum of the first given column (must be of type int or float).
    - **`avg`** – Fetches the average of the first given column (must be of type int or float).
    - **`sum`** – Fetches the summary of the first given column (must be of type int or float).
- Returns the SQL query to the **`Read`** class of the AuroraSQL.

```
db.read(table='users', cols=['id','username','password']).first()
db.read(table='users', cols=['id','username','password']).last()
db.read(table='users', cols=['id','username','password'], where={'id':1}).all()
db.read(table='users', cols=['id','username','password']).count()
db.read(table='users', cols=['id']).min()
db.read(table='users', cols=['id']).max()
db.read(table='users', cols=['id']).avg()
db.read(table='users', cols=['id']).sum()
```

`join`

> Using join method directly from Aurora framework SQL class is somewhat complicated, because you need to provide the foreign key names, the primary keys, and also table names. Hence it is much easier to use it indirectly as a method of Model class.

- Joins related tables.
- Takes up to 11 parameters.
- Required **`table`** parameter holds the name of main table.
- Required **`f_keys`** parameter holds the list of foreign keys.
- Required **`f_tables`** parameter holds the list of related foreign tables.
- Required **`p_keys`** parameter holds the list of primary keys.
- Optional **`cols`** parameter can hold a list of column names that you want to read. (The default value would be [table.*']). (ex. ["table.*", " table1.name", " table2.address"])
- Optional **`where`** parameter can hold a dictionary of column names as keys and values to be check as the WHERE statement. (You will learn more about this param at the following)
- The optional **`order_by`** parameter can hold a dictionary of column names as keys and **`'ASC'`** or **`'DESC'`** values as the ORDER BY statement.
- The optional **`group_by`** parameter can hold the name of a column and represents the GROUP BY statement.
- The optional **`limit`** parameter represents the LIMIT statement.
- The optional **`offset`** parameter represents the OFFSET statement.
- For fetching data, you need to use its own methods:
  - **`first`** – Fetches the first row as a dictionary matching the given SQL query.
  - **`last`** – Fetches the last row as a dictionary matching the given SQL query.
  - **`all`** – Fetches all the rows matching the given SQL query.
  - **`count`** – Counts the number of rows matching the given SQL query.
- Returns the SQL query to the **`Join`** class of the AuroraSQL.

Using the `join` method directly, you need to use the table names before the column names for all parameters. (ex. `table.column`)

You can also use name aliases for columns with the same name for related tables. This way you can prevent name confliction. (ex. `table1.column as custom_column`)

Update Methods

update

- Updates row(s) of a table based on a given where clause.
- Takes up to four parameters.
- Required `table` parameter holds the table name.
- Required `data` parameter holds a dictionary of key & value pairs. Keys represent the columns and values represent the data meant to be updated in the table.
- Optional `where` parameter can hold a dictionary of column names as keys and values to be checked as the WHERE statement. (You will learn more about this param at the following)
- Optional `confirm` parameter can hold a boolean value. Normally, if you represent the where clause you don't need this parameter but if you omit the where parameter, you have to set this parameter to True. In that case it will update all columns of the table, hence you should use this option very carefully.
- Returns `True|False`.

```
data = {
    'username': 'admin',
    'password': '1234567890'
}
db.update(table='users', data=data, where={'id':1})
```

_update_column

- Updates a column with new datatype and constraints.
- Can also rename a column with new datatype and constraints.
- Takes up to five parameters.
- Required `table` parameter holds the table name.
- Required `old_col` parameter holds the old column name.
- Required `new_col` parameter holds the new column name.
- Required `datatype` parameter holds the new column datatype.
- Optional `constraints` parameter holds the column constraints as a string type statement.
- Returns True|False.

```
db._update_column(table='notes', old_col='title', new_col='subject', datatype='TEXT')
```

_update_table

- Renames a table if exists.
- Takes two parameters.
- Required `old_table` parameter holds the table old name.
- Required `new_table` parameter holds the table new name.
- Returns True|False.

```
db._update_table(old_table='notes', new_table='user_notes')
```

Delete Methods

delete

- Deletes row(s) of a table based on a given where clause.

- Takes up to three parameters.
- Required **`table`** parameter holds the table name.
- Optional **`where`** parameter can hold a dictionary of column names as keys and values to be checked as the WHERE statement. (You will learn more about this param at the following)
- Optional **`confirm`** parameter can hold a boolean value. Normally, if you represent the where clause you don't need this parameter but if you omit the where parameter, you have to set this parameter to True. In that case it will delete all columns of the table, hence you should use this option very carefully.
- Returns `True|False`.

```
db.delete(table='notes', where={'id':3})
```

_delete_fk

- Drops a foreign key from an existing table for MySQL and Postgres.
- Takes four parameters.
- Required **`table`** parameter hold the table name.
- Required **`column`** parameter holds the column name.
- Required **`fk_symbol`** parameter holds the foreign key symbol name.
- Required **`confirm`** parameter that should be set to True*.
- Returns **`True|False`**.

* For security reasons you have to conform all developer delete methods.

```
db._delete_fk(table='notes', column='user_id', fk_symbol='fk_notes_users', confirm=True)
```

Note:

The **`_delete_fk`** method doesn't work with SQLite.

_delete_column

- Drops a column from a table.
- Takes three parameters.
- Required `table` parameter hold the table name.
- Required `column` parameter holds the column name.
- Required `confirm` parameter that should be set to True.
- Returns `True|False`.

```
db._delete_column(table='notes', column='new', confirm=True)
```

_delete_table

- Drops a table if exists.
- Takes two parameters.
- Required `table` parameter hold the table name.
- Required `confirm` parameter that should be set to True.
- Returns `True|False`.

```
db._delete_table(table='new', confirm=True)
```

_delete_database

- Drops a database (Removes a database file for SQLite) if exists.
- Takes two parameters.
- Required `database` parameter hold the database name (file for SQLite).
- Required `confirm` parameter that should be set to True.

- Returns `True|False`.

```
# MySQL & Postgres
db._delete_database(database='test_db', confirm=True)

# SQLite
db._delete_database(database='test.db', confirm=True)
```

Security Notes

Important Note:

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

Danger!

If you are a pro user of SQL, 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.

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

Important Note:

For security reasons the developer methods – except for exist methods, – are only available in development! (`DEVELOPMENT = True`)

Custom SQL Queries

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

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 / Postgres

```
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.

Filtering Queries

Using the `where` parameter you can filter your queries for `read`, `update`, and `delete` methods.

Let's take a look at available options (flags) for this parameter:

Flag	Shorthand	Description	Example
`--equal`*	`--e`	Equal to	`Where = {'id--e':1}`
`--not-equal`	`--ne`	Not equal to	`Where = {'id--ne':1}`
`--greater-than`	`--gt`	Greater than	`Where = {'id--gt':2}`

--greater-equal	**--ge**	Greater than or equal to	Where = {'id--ge':2}
--less-than	**--lt**	Less than	Where = {'id--lt':5}
--less-equal	**--le**	Less than or equal to	Where = {'id--le':5}

--like	**--l**	LIKE	Where = {
			'username--l':'%user%'
			}

--not-like	**--nl**	NOT LIKE	Where = {
			'username--nl':'%user%'
			}

--between	**--b**	BETWEEN	Where = {
			'id--b': [1,5]
			}

--not-between	**--nb**	NOT BETWEEN	Where = {
			'id--nb': [1,3]
			}

--in	**--i**	IN	Where = {
			'id--i': [1,3,5]
			}

--not-in	**--ni**	NOT IN	Where = {
			'id--ni': [2,5,4]
			}

You can use either the flag or the shorthand.

* The equal to is the default flag and you can ignore it.

The `where` parameter also supports some prefixes for keys:

Prefix	Shorthand	Description	Example

and--*	**a--**	AND	Where = {
			'id--i': [2,5,4],
			'a--username--l': '%user%'
			}

or--	**o--**	OR	Where = {
			'id': 1,
			'o--id': 2
			}

* The AND is the default join statement for WHERE clause and you can ignore it.

Auto Commitment

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

When a CRUD request has been sent to the database via AuroraSQL, the framework automatically commits the changes and closed the database connection via the destructor method of AuroraSQL.

However, if for any reason your application still uses the database connection and you need to commit an or close the database connection explicitly, you can use the following methods:

- **`save`** – It commits the changes before the database connection has been closed.
- **`close`** – It closes the database connection if there is any.

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

Aurora Helpers

Aurora framework has a `helpers` library that provides many usefull helper functions.

At the following we will introduce some of them.

The general syntax to use them is:

```
From aurora.helpers import helper_name
```

App Helpers

app_exists(app:str)

- Checks if an app exists using its name.
- Returns a dictionary with two pairs of key values.

```
{
     'result': True|False,
     'url':    'app_url|An_Error'
}
```

app_url_exists(url:str)

- Checks if an app exists using its URL.
- Returns `True|False`.

controller_exists(app:str, controller:str)

- Checks if a controller name exists using its app name.
- Returns a dictionary with two pairs of key values.

```
{
```

```
    'result': True|False,
    'url':    'controller_url|An_Error'
}
```

controller_url_exists(app:str, url:str='')

- Checks if a controller URL exists using its app name.
- Returns `True|False`.

String Handling

random_string(size:int=8)

- Generates a random string.

snake_case(CamelCase:str)

- Converts CamelCase string to snake_case string.

to_snake_case(text:str)

- Converts a text to snake_case string.

delete_chars(text:str, char:str)

- Deletes the sequences of a character from a text.

remove_html(text:str)

- Removes HTML tags from a text.
- Returns plain text.

`fixed_chars(text:str, num:int)`

- Returns a fixed sized text.

`clean_text(text:str, num:int)`

- Removes HTML tags from a text.
- Returns fixed sized text.

JSON Helpers

`is_json(text:str)`

- Checks if a text is in correct JSON format.
- Returns `True|False`.

`json_dict(path:str)`

- Produces a dictionary from a json file.

`json_eval(text:str)`

- Evalutes a JSON object (text) into a python dictionary.

File Handling

`file_exist(path:str)`

- Checks if a file exists using its full path.

- Returns `True|False`.

`create_file(file_path:str, content:str='')`

- Creates a file if not exists.
- Optionaly can put some content in it.

`write_file(file_path:str, content:str='')`

- Writes a content to a file if exists using its full path.

`read_file(file_path:str)`

- Reads a file if exists, and returns its content using its full path.

`copy_file(src:str, dist:str)`

- Copies a file to a new destination.
- `src` -- The source file path
- `dist` -- The destination file path

`move_file(src:str, dist:str)`

- Moves a file to a new destination.
- `src` -- The source file path
- `dist` -- The destination file path

`rename_file(src:str, dist:str)`

- Renames a file if exists.

- **`src`** -- The source file path
- **`dist`** -- The destination file path

`delete_file(file_path:str)`

- Removes a file permanently if exists.
- **`file_path`** -- The absolute file path

`unzip_file(file_path:str, dist_dir:str)`

- unzips a zip file to a directory.
- **`file_path`** -- The absolute zip file path
- **`dist_dir`** -- The destination directory to unzip the file

`replace_file_string(file_path:str, old_str:str, new_str:str, regex:bool=False)`

- Replaces strings in a file with new ones.
- **`file_path`** -- The absolute file path
- **`old_str`** -- The old string
- **`new_str`** -- The new string
- **`regex`** -- For replacing a regular expression

`replace_file_line(file_path:str, old_line:str, new_line:str, regex:bool=False)`

- Replaces lines in a file contain a string with new line data
- **`file_path`** -- The absolute file path
- **`old_line`** -- The character to match in the line
- **`new_str`** -- The new line data

- **`regex`** -- For replacing a regular expression

Directory Handling

`dir_exist(dir:str)`

- Checks if a directory exists using its path.
- Returns **`True|False`**.

`dir_empty(dir:str)`

- Checks if a directory is empty.
- Returns **`True|False`**.

`create_dir(dir:str)`

- Creates a directory if not exists using a path.
- Returns **`True|False`**.

`rename_dir(src:str, dist:str)`

- Renames a directory if exists using its path.
- Returns **`True|False`**.

`delete_dir(dir:str)`

- Removes a directory and all its files and sub-directories permanently using its path.
- Returns **`True|False`**.

`dir_size(dir:str)`

- Calculates a directory size and returns it in Bytes.

Time Helpers

`current_time()`

- Generates current time in milliseconds.

`generate_time(date, format='%Y-%m-%d %H:%M:%S')`

- Generates time in milliseconds from a date
- **`date`** -- The date string
- **`format`** -- The date format

`create_time(seconds:int=0, minutes:int=0, hours:int=0, days:int=0, weeks:int=0, months:int=0, years:int=0)`

- Creates a time in milliseconds from given parameters
- **`seconds`** -- The seconds from or before the current time
- **`minutes`** -- The minutes from or before the current time
- **`hours`** -- The hours from or before the current time
- **`days`** -- The days from or before the current time
- **`weeks`** -- The weeks from or before the current time
- **`months`** -- The months from or before the current time
- **`years`** -- The years from or before the current time

`current_date(format:str='%Y-%m-%d %H:%M:%S')`

- Generates the current date
- **`format`** -- The date format

`generate_date(time_ms:int, format:str='%Y-%m-%d %H:%M:%S')`

- Generates datetime from a time in milliseconds
- **`time_ms`** -- The time in milliseconds
- **`format`** -- The date format

`create_date(seconds:int=0, minutes:int=0, hours:int=0, days:int=0, weeks:int=0, months:int=0, years:int=0, format:str='%Y-%m-%d %H:%M:%S')`

- Creates datetime from given parameters
- **`seconds`** -- The seconds from or before the current time
- **`minutes`** -- The minutes from or before the current time
- **`hours`** -- The hours from or before the current time
- **`days`** -- The days from or before the current time
- **`weeks`** -- The weeks from or before the current time
- **`months`** -- The months from or before the current time
- **`years`** -- The years from or before the current time
- **`format`** -- The date format

`week_number(time, first_day='Sunday', format='%Y-%m-%d')`

- Returns week number of a time (in milliseconds) or a date, counting from **`1`**
- **`time`** -- The time (or date)
- **`first_day`** -- The first day of week (Sunday|Monday)

- `format` -- The date format if time is date

`delta_seconds(time_one, time_two, format='%Y-%m-%d %H:%M:%S')`

- Returns seconds difference between two times (in milliseconds) or dates
- `time_one` -- The first time (or date)
- `time_two` -- The second time (or date)
- `format` -- The date format if times are date

`delta_days(time_one, time_two, format='%Y-%m-%d')`

- Returns days difference between two times (in milliseconds) or dates
- `time_one` -- The first time (or date)
- `time_two` -- The second time (or date)
- `format` -- The date format if times are date

`delta_weeks(time_one, time_two, first_day='Sunday', format='%Y-%m-%d')`

- Returns weeks difference between two times (in milliseconds) or dates
- `time_one` -- The first time (or date)
- `time_two` -- The second time (or date)
- `first_day` -- The first day of week (Sunday|Monday)
- `format` -- The date format if times are date

`delta_months(time_one, time_two, format='%Y-%m-%d')`

- Returns months difference between two times (in milliseconds) or dates
- `time_one` -- The first time (or date)

- `time_two` -- The second time (or date)
- `format` -- The date format if times are date

`delta_years(time_one, time_two, format='%Y-%m-%d')`

- Returns years difference between two times (in milliseconds) or dates
- `time_one` -- The first time (or date)
- `time_two` -- The second time (or date)
- `format` -- The date format if times are date

Discover More

Contribution Request

The term security is more complicated than what we discussed in this documentation.

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

Aurora framework is in a beta version and needs contribution to be developed.

I will do my best to make this framework something really great for all of us.

I warmly welcome and appreciate any contribution requests from anyone who wants to work on this open-source project.

Free Notes Project

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

You can find almost all of what 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 to it!

About the Author

I'm Hemin Satya, a freelance programmer who loves to write code.

Aurora is my first open-source project, I hope you like it. I have many ideas and codes that I 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 learn 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 one 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 was able to do something useful for you that worth your past time.

Sincerely yours.