dragonfly

Release 0.0.3

Feb 24, 2020

Contents:

CHAPTER 1

Installation

1.1 Installing Python

For dragonfly to work you must have Python 3.6 or above installed. For further details on how to do this please see here.

1.2 Installing Dragonfly

First, download the Dragonfly repository

- Via git

  ```
  git clone git@github.com:MattJMLewis/dragonfly-app.git
  ```

- Via GitHub

  Simply download the repository and unzip it.

1.3 Installing requirements

Next, enter the dragonfly directory and run the following command.

```
pip install -r requirements.txt
```

On Windows you may encounter an error installing `mysqlclient`. If this happens you can download the latest `.whl` here. There is also an unoffical repo of `.whl` files here. Then simply run `pip install name-of-the-whl-file.whl`.

After this run the following command to create the necessary directories for `dragonfly` to work.

```
python builder.py setup
```

1.4 Setting up a MySQL database

Dragonfly currently only supports `MySQL` databases. See here for more details on how to set up a server.

1.5 Running dragonfly

- Development

  Simply run the `main.py` file. However before you do this you should modify the `config.py` file to match your setup.

- Production

  Coming soon...

1.6 Usage

Please see *here* for the quick start guide on using dragonfly.

CHAPTER 2

Quick Start

This guide assumes you already have dragonfly installed. Please see the *installation* section if you do not.

2.1 Design Philosophy

Dragonfly is based on the model-view-controller architectural pattern. This means that the model structure, application logic and user interface are divided into separate components. As this is a brief quick start guide some features won't be shown. To get a full list of features please see the API reference.

2.2 Routing

Routing allows dragonfly to know the location to send a HTTP request. In the example below the GET request to `/testing` is routed to the `index` function on `TestController`. This is all registered in the `routes.py` file.

```
from dragonfly.routes import Router

Router.get('testing', 'TestController@test')
```

2.2.1 HTTP Methods

Dragonfly supports the following HTTP methods:

HTTP Method	Router method
GET	`.get()`
POST	`.post()`
PUT	`.put()`
PATCH	`.patch()`
OPTIONS	`.options()`
DELETE	`.delete()`
All of the above	`.any()`

2.2.2 Resource Routing

The `Router` class also has a special method called `resource`. Its second argument is an entire controller. Here is an example:

```
Router.resource('articles', 'ArticleController') # Notice there is no @{method_name}
```

Calling `.resource('ArticleController')` will register the following routes:

Route	HTTP Method	Controller function	Description
`/articles`	GET	`ArticleController@index`	Return all articles in the database.
`/articles/<id:int>`	GET	`ArticleController@show`	Return the article with the given id or fail.
`/articles/create`	GET	`ArticleController@create`	Show the form to create a new article.
`/articles`	POST	`ArticleController@store`	Store (and validate) the given values in the POST request.
`/articles/<id:int>/edit`	GET	`ArticleController@edit`	Show the form to edit a article.
`/articles/<id:int>`	PUT	`ArticleController@update`	Update the given article using the given values.
`/articles/<id:int>`	DELETE	`ArticleController@delete`	Delete the given article.

2.2.3 Route Parameters

Route parameters are a way of passing variables to the controller to help find a certain piece of data. In the example above if the URL `/articles/1` was called, the integer `1` would be passed to the controller `show` function through a variable named `id`. This allows for the look up and return of the given article from the database. A route parameter should follow the pattern below:

```
<name_of_variable:expected_type>
```

It is possible to have multiple parameters on a route. For example:

```
Route.get('/articles/<id:int>/<comment_id:int>')
```

Dragonfly supports the following types by default:

Type	Regex
`int`	`([0-9]+)`
`str`	`(.+)`

Custom types

It is very easy to define your own custom types. Simply add a new key (name of the type), value (regex to match) pair in the `PYTHON_TO_REGEX`dictionary in `config.py`. For example:

```
PYTHON_TO_REGEX = {"int": "([0-9]+)", "str": "(.+)",
                   "str_capitalised": "(\b[A-Z].*?\b)"}
```

2.3 Controllers

A controller should contain all of your application logic to do with that resource. The following command will create a file in the `controllers` directory called `article_controller`:

```
python builder.py generate --type=controller article_controller
```

Each time a request is routed a new instance of the registered controller will be instantiated and the registered function run.

The following is the basic structure of a controller:

```
from dragonfly import Controller
from models.Article import Article
from dragonfly import View


class ArticleController(Controller):

    def show(self, id):
        return View('articles.show', article=Article().find(id))
```

The following route would match to this controller method:

```
Route.get('/articles/<id:int>', 'ArticleController@show')
```

2.4 Middleware

Middleware provides a way to stop or modify a request cycle; before the request is routed, after a response is returned from the controller or both. Dragonfly comes with a few premade middleware. You can also create your own middleware using the following command:

```
python builder.py generate --type=middleware article_middleware
```

The following is an example of middleware that will run on any route that resolves the `show` method in the `ArticleController`. It is possible to assign a middleware to multiple actions by appending to the `actions` list. The `before` method here uses the singleton of the `DeferredResponse` class to set the header for the response before it has been generated (NOTE: This does **not** set the headers for any `Response` other than the one returned by the controller).

In the `before` and `after` method if any `Response` class or child of the `Response` class is returned the processing of the request will stop and the response returned.

```
from dragonfly import request
from dragonfly import ErrorResponse, deferred_response

class ArticleMiddleware:

    actions = ['ArticleController@show']

    def before(self):
        if visited in request.cookies:
            return ErrorResponse(404, "You have already visited the page.")

        deferred_response.header('Set-Cookie', 'visited=True')


    def after(self):
        pass
```

2.5 Models (ORM)

Models provide an easy way to read, write and update a table in the database through a Python class. To start using the ORM you first need to define the attributes of a model. This is all done through a model class file. This can be generated using the CLI:

```
python builder.py generate --type=model article
```

A new file will be created in the `models` directory. Below is an example of an articles model and the SQL it generates.

```
from dragonfly import models

class Article(models.Model):

    id = models.IntegerField(unsigned=True, auto_increment=True, primary_key=True)
    title = models.CharField(max_length=10)
    text = models.TextField()
```

There are many field types and options for each field type. For an exhaustive list of these please see the *API reference*. It is also important to note that you can add any function you would like to the model class. For example a way to generate the slug for an article:

```
def slug(self):
    return f"/articles/{self.id}"
```

Once you have defined the model you need to generate and execute the SQL to create the table. To do this simply run the following command.

```
python builder.py migrate
```

Once complete you should be able to manipulate the newly created `articles` table through the `Article` model. Below is an example of retrieving all articles in the database:

```
from models.article import Article

articles = Article().get()
```

The ORM has a large number of methods that are all listed in the *API reference*. However, below are listed some of the common methods:

2.6 Templates

Dragonfly provides an easier way to join Python and HTML. A view is stored in the `views` directory and should be a `html`file. The templates can also be in subdirectories of the `views` directory.

Below is an example of a html file saved in `views/articles/index.html`

```
<head>
    <meta charset="utf-8">
    <title>Articles | Index</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
</head>

<h1>Articles</h1>
<hr>

<body>
    <div>
        {% for article in articles }}
            <a href="http://{{ [article.slug()] }}">
                {{ [server.name] }}
            </a>
        {{ end for %}

        {% if display_help }}
            Looks like you need some help...
        {{ end if %}
    </div>
</body>
</html>
```

To call and render this view you would write the following in your controller:

```
return View('articles.index', articles=articles, display_help=True).make()
```

There are a few important things to note about the syntax of the templating system:

- To write variables simply wrap `{{ }}` around the variable name.
- Due to the way the template compiler works if the variable is one 'generated' by a for loop, like the `article` variable in the example above, it must also be wrapped by `[ ]`.
- Command structures are declared by having the opening clause begin with `{%` and the ending clause close with `%}`.
- The only command structures available currently are `if` and `for`.
- Each command structure must have a start and end clause (`if ...`, `end if`, `for ...`, `end for`).

2.7 Demo App

Please see .. _here: https://github.com/MattJMLewis/dragonfly-demo for an example project.

CHAPTER 3

API Reference

3.1 Database package

3.1.1 DB

class dragonfly.db.database.**DB**(*database_settings=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: object

An easy way to interact with the configured database.

chunk(*chunk_loc*, *chunk_size*)

This will run the given query and return the given number of results at the given location.

Parameters

- **chunk_loc** (*int*) – The location to chunk e.g the first chunk or second chunk.
- **chunk_size** (*int*) – The number of rows each chunk should contain.

comparison_operators = ['=', '<=>', '<>', '!=', '>', '>=', '<', '<=', 'IN()', 'NOT', '

custom_sql(*sql*, *n_rows=None*)

delete()

Deletes the given row/rows.

For this method to run the *where* method must have been called before this one.

first()

This will execute the query you have been building and return only the first result (uses LIMIT 1)

get()

This will execute the query you have been building and return all results.

insert(*insert_dict*)

Inserts the given values into the database.

Parameters **insert_dict** (*dict*) – The dictionary containing the column and the value to insert into that column

multiple_where(*where_dict*)
Allows for multiple where clauses through one command. Note this only supports the = operator.

Parameters **where_dict** (*dict*) – The values to match

select(**args*)
Equivalent to the `SELECT` clause in MySQL.

Add the column you would like as an argument to the function. You can choose many columns.

Example `DB().select('title', 'text')`

Note: If you would like to select all (`*`) columns then simply do not use the select argument when

---

building your query.

table(*table_name*)
The table that the query should be run on. This method must be run for any query to be executed.

update(*update_dict*)
Updates the given row/rows based on the dictionary.

For this method to run the *where* method must have been called before this one.

Parameters **update_dict** (*dict*) – The dictionary containing the column to update and the value to update it with.

where(*condition_1*, *comparison_operator*, *condition_2*)
Equivalent to the `WHERE` clause in SQL.

Parameters

- **condition_1** – The value to the left of the operator.
- **comparison_operator** (*str*) – The operator, e.g =
- **condition_2** – The value to the right of the operator.

3.1.2 Fields

Note:

- Please note that the majority of MySQL field types are available for usage. There name will just be the camel case of the MySQL type with `Field` appended.
- All fields accept the following parameters: `null`, `default`, `unique`, `primary_key`. These values are, by default, `False`. Some fields will have extra parameters which can be seen below.

---

class dragonfly.db.models.fields.**BigIntField**(***kwargs*)
Bases: *dragonfly.db.models.fields.IntField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**BinaryField**(***kwargs*)
Bases: *dragonfly.db.models.fields.StringField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**BitField**(*length=None*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**BoolField**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**CharField**(***kwargs*)
Bases: *dragonfly.db.models.fields.StringField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

class dragonfly.db.models.fields.**DateField**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**DateTimeField**(*fsp=None*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**DecimalField**(*digits=None*, *decimal_places=None*, *unsigned=False*, *zerofill=False*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**DoubleField**(*digits=None*, *decimal_places=None*, *unsigned=False*, *zerofill=False*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**Enum**(**args*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**Field**(*name=None*, *null=False*, *blank=False*, *default=None*, *unique=False*, *primary_key=False*)
Bases: abc.ABC

An abstract class that defines the interface each *Field* class should have.

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**FloatField**(*digits=None*, *unsigned=False*, *zerofill=False*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**ForeignKey**(**args*)
Bases: object

on(*table*)

references(**args*)

class dragonfly.db.models.fields.**IntField**(*length=None*, *unsigned=False*, *auto_increment=False*, *zerofill=False*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**LongBlob**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**MediumBlob**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**MediumIntField**(***kwargs*)
Bases: *dragonfly.db.models.fields.IntField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**MediumText**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**PrimaryKey**(**args*)
Bases: object

class dragonfly.db.models.fields.**Set**(**args*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**SmallIntField**(***kwargs*)
Bases: *dragonfly.db.models.fields.IntField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

class dragonfly.db.models.fields.**StringField**(*length=None*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**TextField**(***kwargs*)
Bases: *dragonfly.db.models.fields.StringField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

class dragonfly.db.models.fields.**TimeField**(*fsp=None*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**TimestampField**(*fsp=None*, *on=None*, ***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**TinyBlobField**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**TinyIntField**(***kwargs*)
Bases: *dragonfly.db.models.fields.IntField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

class dragonfly.db.models.fields.**TinyTextField**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

class dragonfly.db.models.fields.**Unique**(**args*)
Bases: object

class dragonfly.db.models.fields.**VarCharField**(***kwargs*)
Bases: *dragonfly.db.models.fields.StringField*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

class dragonfly.db.models.fields.**YearField**(***kwargs*)
Bases: *dragonfly.db.models.fields.Field*

to_database_type()
This instructs the database migrator on how to generate the SQL for the model.

to_python_type(*value*)
This is how the value from the database should be converted to python. Note that at the moment this is not currently in use as the MySQL adapter does this automatically

3.1.3 Model

class dragonfly.db.models.model.**Model**(*data=None*)
Bases: object

A way to easily interact with rows in a table.

add_relationship(*relationship_class*, *update=False*)

all()
Get all rows in the database.

create(*create_dict*)

data_to_attributes(*data*)
Converts the given data to class attributes in the class instance.

The given data (from the database) is first converted to its equivalent python type and then assigned to the class dictionary as well as a `database_values` dictionary that is used to retrieve the correct row from the database when updating the model.

Parameters **data** – The data to assign to the model instance

Returns

data_to_model(*data*)
Starts the process of converting data from the database to model instances.

Parameters **data** – The data to convert

Returns

delete()

find(*primary_key*)
Find a row by passing in the value of the desired row's primary key.

Note that if you would like to find a model with more than one key pass through a dictionary containing the column and value.

Example `Article().find({'id': 1, 'author': 1})`

first()
Get the first row in the table.

get()
Get all rows in the table that correspond to the other specified selectors.

get_attributes()

multiple_where(*where_dict*)

paginate(*size*, *to_json=False*)
Paginates the data in the table by the given size.

Note that if `to_json` is `True` a response will be returned containing the appropriate JSON, otherwise a list of rows that correspond to the page requested will be returned (the page number is known from the request object).

Parameters

- **size** (*int*) – The number of rows on each page
- **to_json** (*bool*) – If the function should return a JSON response, default is False

save()
Permeate the changes to the model attributes, to the database.

select(**args*)
Same as the *DB* class *select* method. Note that a dictionary will be returned as a model cannot be represented with incomplete data.

to_dict()

update(*update_dict*)

where(*column*, *comparator*, *value*)
Same as the *DB* class *where* method.

dragonfly.db.models.model.**default**(*o*)
Function from StackOverflow - https://stackoverflow.com/questions/12122007/python-json-encoder-to-support-datetime

Warning: The following classes should not be called directly.

3.1.4 DatabaseMigrator

class dragonfly.db.database_migrator.**DatabaseMigrator**(*path='models'*)
Bases: object

Generates the SQL to create a table that corresponds to the defined model/s

3.1.5 Table

class dragonfly.db.table.**Table**
Bases: object

Returns the MySQL code to create a column in a table with the given type.

static bigint(**args*, ***kwargs*)

static binary(**args*, ***kwargs*)

static bit(**args*, ***kwargs*)

static blob(**args*, ***kwargs*)

static boolean(**args*, ***kwargs*)

static char(**args*, ***kwargs*)

static date(**args*, ***kwargs*)

static datetime(**args*, ***kwargs*)

static decimal(**args*, ***kwargs*)

static double(**args*, ***kwargs*)

static enum(**args*, ***kwargs*)

static float(**args*, ***kwargs*)

static foreign_key(*constraint_name*, *table*, *local_keys*, *foreign_keys*)

static integer(**args*, ***kwargs*)

static longblob(**args*, ***kwargs*)

static longtext(**args*, ***kwargs*)

static mediumblob(**args*, ***kwargs*)

static mediumint(**args*, ***kwargs*)

static mediumtext(**args*, ***kwargs*)

static primary_key(**args*)

static set(**args*, ***kwargs*)

static smallint(**args*, ***kwargs*)

static text(**args*, ***kwargs*)

static time(**args*, ***kwargs*)

static timestamp(**args*, ***kwargs*)

static tinyblob(**args*, ***kwargs*)

static tinyint(**args*, ***kwargs*)

static tinytext(**args*, ***kwargs*)

static unique(**args*, *constraint_name=None*)

static varbinary(**args*, ***kwargs*)

static varchar(**args*, ***kwargs*)

static year(**args*, ***kwargs*)

dragonfly.db.table.**handle_options**(*func*)

Handles any extra options for the methods on the *Table* class

3.2 Middleware package

Warning: The following classes should not be called directly.

3.2.1 MiddlewareController

class dragonfly.middleware.middleware_controller.**MiddlewareController**
Bases: object

Compiles all registered middleware and controls their execution.

create_action_cache(*action*)
Create a cache of middleware that are currently in use.

register_middleware()
Register the middleware that is defined in the config file.

run_after(*action*, *response*)
Run all the after methods on middleware that are assigned to the given action.

run_before(*action*)
Run all the before methods on middleware that are assigned to the given action

Parameters **action** – The action currently being executed.

3.3 Routes

3.3.1 Router

class dragonfly.routes.router.**Router**
Bases: object

Routes the given route to the defined Controller and returns its generated *Response*.

add_route(*uri*, *action*, *method*)
Adds a route to the *RouteCollection* object.

Parameters

- **uri** (*str*) – The uri of the route
- **action** (*str*) – The action of the route e.g 'HomeController@home'
- **method** (*str*) – The HTTP method verb e.g 'GET'

any(*uri*, *action*)

delete(*uri*, *action*)

dispatch_route()
Dispatches the appropriate route based on the request method and path.

get(*uri*, *action*)

get_routes()

options(*uri*, *action*)

patch(*uri*, *action*)

post(*uri*, *action*)

put(*uri*, *action*)

resource(*uri*, *controller*)

dragonfly.routes.router.**to_snake**(*name*)

Warning: The following classes should not be called directly.

3.3.2 RouteCollection

class dragonfly.routes.route_collection.**RouteCollection**
Bases: object

A way to store registered routes.

add(*uri*, *action*, *method*)
Add a new route to either the static or dynamic routes dictionary.

Parameters

- **uri** (*str*) – The route uri
- **action** (*str*) – The route action
- **method** (*str*) – The route HTTP method

match_route(*uri*, *method*)
Match the given route using its URI and method. First we check if it is a static route before checking all dynamic routes

3.3.3 RouteRule

class dragonfly.routes.route_rule.**RouteRule**(*uri*)
Bases: object

Used to register dynamic routes. Allows for an easy check of whether a given route matches a dynamic route.

match(*uri*)
Matches the given route to an action and extracts any router parameters.

Parameters **uri** (*str*) – The URI to match.

Returns A dictionary containing the the action and any route parameters.

Return type dict

3.4 Templates

3.4.1 View

class dragonfly.template.template.**View**(*view*, ***kwargs*)
Bases: object

Returns a HTML version of the requested template. The class first finds the desired view. If a pre-compiled python version of the template does not exist or is out of date, the class will generate one. Otherwise it imports the compiled python file and runs the get_html method, passing in any variables that the user.py has given to the constructor (via **kwargs). It then returns a *Response* with this HTML. :param view: The view to return :type view: str

make()
Returns a response with the generated HTML. :return: The response :rtype: Response

Warning: The following classes should not be called directly.

3.4.2 Converter

3.4.3 Parser

3.4.4 Clause

3.4.5 Snippet

3.4.6 Stack

3.5 Web

3.5.1 Request

class dragonfly.request.**Request**(*environ*)

Bases: object

get_data()

update_environ(*new_environ*)

3.5.2 Response

class dragonfly.response.**Response**(*content=''*, *content_type='text/html'*, *status_code=200*, *reason_phrase=None*)

Bases: object

The base *Response* class that is readable by the WSGI server.

Parameters

- **content** (*str*) – The content that will be delivered to the user.py. This defaults to empty.
- **content_type** (*str*) – The MIME type. This defaults to 'text/html'.
- **status_code** (*int*) – The HTTP status code. This defaults to success (200).
- **reason_phrase** – A written meaning of the HTTP status code. If left as *None* the reason phrase will be

chosen from a pre-determined list. :type reason_phrase: int

header(*field_name*, *field_value*)

Updates an existing header or creates a new one.

Parameters

- **field_name** (*str*) – The header field name.
- **field_value** (*str*) – The header field value.

set_content()

Converts the given content to bytes.

set_status(*status_code*, *reason_phrase*)
Sets the status of the *Response* object. If the reason_phrase is None then a reason phrase that corresponds to the status code will be retrieved from a constants file.

Parameters

- **status_code** (*int*) – The status code of the response.
- **reason_phrase** (*str*) – The reason phrase to accompany the status code. This can be *None*.

translate_deferred(*deferred*)
Merges the given *DeferredResponse* object to this *Response* instance.

Parameters **deferred** (DeferredResponse) – The *DeferredResponse* object.

3.5.3 ErrorResponse

class dragonfly.response.**ErrorResponse**(*error_message*, *status_code*)
Bases: *dragonfly.response.Response*

A *Response* object that returns an error page.

Parameters

- **error_message** (*str*) – The error message.
- **status_code** (*int*) – The status code.

3.5.4 DeferredResponse

class dragonfly.response.**DeferredResponse**
Bases: object

Allows headers for a future response to be set before it exists.

This singleton enables attributes of any *Response* object returned in the normal fashion, i.e through the *dispatch_route* method, to be set before it exists. The primary use of this class would be in the *before* method of a middleware.

To do: - Support modifying content and status.

header(*field_name*, *field_value*)
Define the headers to be set on the real *Response* object.

CHAPTER 4

To do

Danger:

- [X] Fix `Request` so that it does not crash when cookies are not present.
- [X] Prevent MySQL injection on `DB`.

Caution:

- Production server.
- [X] Make all single-use classes into singletons. This is done via module level variables
- [X] Reduce length of namespace for classes.
- [X] Add a `RedirectResponse`.
- `create` & `delete` pk auto detection works for non composite keys. How to solve for composite?
- Add `created_at`, `updated_at` and `id` fields to class by default.
- [X] Add a `url` method that extracts the base url of the site from the config.
- Add a `{{ csrf_token }}` template variable to prevent CSRF attacks.
- Add a built in auth system (including auth middleware, `User` class and a way to easily retrieve the authenticated `User`).
- Look at Laravel/Django and add more features to `Response` & `Request` and doc blocks.

Attention:

- Add a caching system
- Is there a way to speed up SQL commands?

- Make `View` syntax more natural.
- Allow the user to specify a secondary database connection and use that on a model.