apy Documentation

Release 1.0

Felix Carmona, stagecoach.io

July 19, 2014

Contents

Contents:

CHAPTER 1

Starting up an Application

The `apy.main.Application` class is the heart of the apy system, this class will be your main entry point, and will manage your configuration files, run your web server, manage the router and the dispatcher, etc.

1.1 The directory structure

```
myapplication/
    config/
        settings_prod.yml
    __init__.py
    index.py
```

The `config/` dir contains the configuration files.

- The `settings_*.yml` file is the main configuration file. Example:

```
imports:
  - others_parameters.yml
  - foo.yml

server_adapter: tornado

server:
  port: 80
  workers: 5

routes:
  hello_world:
    path:       /hello/world
    methods:    [GET]
    controller: myapplication.hello.HelloWorldController
  foo_bar:
    path:       /foo/bar
    methods:    [GET, POST]
    controller: myapplication.foo.FooBarController

debug: no

error_handler: apy.application.DefaultErrorHandler

parameters:
  foo: bar
```

```
  hello: world

services:
  my_mailer:
    class: myapplication.mymailer.Mailer
    arguments:
      name: '{{ foo }}'
```

- The `imports` will include the config of the desired.yml (you can nest as many as you want).

  The `parameters`, `routes` and `services` imported sub-elements will be merged with the importer's sub-elements. All other parameters will be overwritten with no-merge. At the two cases (merge and no-merge), at conflicts, the importer has more priority and will overwrite the imported others.

  You can use the `imports` to split your routes, parameters, etc in single files, etc.

- The `routes` sets the application routes. (see *routing*)
- The `services` and `parameters` defines the container. (see *container*)

1.2 Running the application as server

In the previous directory structure example, the next file is named `index.py`, but you can name it as you want.

```
1 from apy.main import Application
2
3
4 application = Application()
5 application.run_server()
```

Note: By default the `Application(environment='prod', config_dir='config')` will use the `config` relative directory, (you can use other directory name, or specify an absolute path) and the `settings_prod.yml` will be chosen (if you set the environment as `dev`, the application will use the `settings_dev.yml`).

CHAPTER 2

Routing

2.1 Routing in Action

A route is a map from a URL path to a controller.

For example, suppose you want to match any URL like /blog/my-post or /blog/all-about-cats and send it to a controller that can look up and render that blog entry info. The route is simple:

```
# config/settings_prod.yml
...

routes:
    blog_show:
        path:       /blog/{slug}
        methods:    [GET]
        controller: myapplication.blog.BlogController

...
```

The path defined by the `blog_show` route acts like `/blog/*` where the wildcard is given the name slug. For the URL `/blog/my-blog-post`, the slug variable gets a value of `my-blog-post`, which is available for you to use in your controller (keep reading). The `blog_show` is the internal name of the route, which doesn't have any meaning yet and just needs to be unique. Later, you'll use it to generate URLs.

The `controller` parameter is a special key that tells **apy** which controller should be executed when a URL matches this route. The `controller` string value point to a specific python module > class:

```
# myapplication/blog.py
from apy.handler import Controller
from apy.http import Response


class BlogController(Controller):
    def action(self, slug):
        # use the slug variable to query the database
        post_info = ...

        response = Response()

        response.data = post_info

        return response
```

Congratulations! You've just created your first route and connected it to a controller. Now, when you visit `/blog/my-post`, the `action` method of the `myapplication.blog.BlogController` class will be executed and the `slug` variable will be equal to `my-post`.

This is the goal of the **apy** router: to map the URL of a request to a controller. Along the way, you'll learn all sorts of tricks that make mapping even the most complex URLs easy.

2.1.1 Customizing the path matching Requirements

2.2 Creating Routes

2.2.1 Customizing the action method name of the controller class

2.2.2 Adding HTTP Method Requirements

In addition to the URL, you can also match on the *method* of the incoming request (i.e. GET, HEAD, POST, PUT, DELETE). Suppose you have a contact form with two controllers - one for displaying the form info (on a GET request) and one for processing the form when it's submitted (on a POST request). This can be accomplished with the following route configuration:

Despite the fact that these two routes have identical paths (`/contact`), the first route will match only GET requests and the second route will match only POST requests. This means that you can display the form info and submit the form via the same URL, while using distinct controllers for the two actions.

Note: If no method is specified, the route will match with *all* valid methods.

According to the *RFC 2616*, the valid HTTP request methods are:

`GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT` and `PATCH`

Note: You can specify multiples methods specifying a yaml list.

All of the following method configurations are valid:

```
methods: [GET]

methods: [GET, POST]

methods:
  - GET
  - DELETE

methods:
  - OPTIONS
```

2.2.3 Adding Protocol Requirements to use HTTPS or HTTP

Sometimes, you want to secure some routes and be sure that they are only accessed via the HTTPS protocol.

This can be accomplished with the following route configuration:

```
routes:
    contact:
        path:       /contact
```

```
        controller: myapplication.main.ContactController
        protocols:  [https]
```

Note: If the `protocols` directive is not specified, by default the route will match with **HTTP** and **HTTPS**.

Note: The protocols will be specified using a yaml list.

All of the following method configurations are valid:

```
protocols: [http]

protocols: [http, https]

protocols:
  - https
  - http

protocols:
  - https
```

2.2.4 Adding Host Requirements

CHAPTER 3

Container

3.1 Parameters

The parameters container is a container for key/value pairs.

You controller class has a `self._parameters` attribute with the configured `parameters` in your `config_*.yml` file.

The available methods are:

- `set(key, value)`: Sets a parameter by name.
- `get(key, default=None)`: Returns a parameter by name. If the key don't exists, the default parameter will be returned.
- `has(key)`: Returns *True* if the parameter exists, *False* otherwise.
- `remove(key)`: Removes a parameter.
- `add(parameters)`: Adds a dict of parameters
- `all()`: Returns the parameters.
- `count()`: Returns the number of parameters.
- `keys()`: Returns the parameter keys.
- `parse_text(text)`: Returns the parameters in the string resolved.

Note: You can construct some parameters containers for your application instantiating a `apy.container.Parameters` class (the class is constructed passing an optional parameters dict to the `__init__(parameters=None)` constructor.

Note: You can reference other parameters wrapping it between `{{ }}` characters:

example: `'foo': '{{ bar }}', 'bar': 'aaa'`, if you get the `foo` parameter, the return value should be `aaa` because `foo -> {{ bar }} -> bar -> aaa`

Also, you can combine multiple parameters with text, etc. ie:

```
parameters:
    name:       Felix
    surname:    Carmona
    hello_message:  "Hello {{ name }} {{ surname }}!"
```

`self._parameters.get('hello_message')` will return *Hello Felix Carmona!*

You can escape brackets processing with "\". Example:

```
parameters:
    name:       Felix
    hello_message:  "Hello \{\{ name \}\}"
```

`self._parameters.get('hello_message')` will return *Hello {{ name }}!*

3.2 Services

3.2.1 What is a Service Container

A Service Container (or *dependency injection container*) is simply a python object that manages the instantiation of services (objects). For example, suppose you have a simple python class that delivers email messages. Without a service container, you must manually create the object whenever you need it:

```
from myapplication.mailer import Mailer

mailer = Mailer('sendmail')
mailer.send('felix@example.com', ...)
```

This is easy enough. The imaginary *Mailer* class allows you to configure the method used to deliver the email messages (e.g. *sendmail*, *smtp*, etc).

But what if you wanted to use the mailer service somewhere else? You certainly don't want to repeat the mailer configuration every time you need to use the Mailer object. What if you needed to change the *transport* from *sendmail* to *smtp* everywhere in the application? You'd need to hunt down every place you create a *Mailer* service and change it.

The Services container allows you to standardize and centralize the way objects are constructed in your application.

3.2.2 Creating/Configuring Services in the Container

A better answer is to let the service container create the *Mailer* object for you. In order for this to work, you must teach the container how to create the *Mailer* service. This is done via configuration, which would be specified in *YAML*:

```
# config/settings_prod.yml
...
services:
    my_mailer:
        class:        myapplication.mailer.Mailer
        arguments:    [sendmail]
...
```

An instance of the *myapplication.mailer.Mailer* object is now available via the service container. The services container is available in any traditional apy controller where you can access the services of the container via the `self._services.get(name)` method:

```
# myapplication/hello.py
from apy.handler import Controller
...
class HelloController(Controller):
    def action(self):
```

```
        ...
        mailer = self._services.get('my_mailer')
        mailer.send('felix@example.com', ...)
        ...
```

When you ask for the *my_mailer* service from the container, the container constructs the object and returns it. This is another major advantage of using the service container. Namely, a service is never constructed until it's needed. If you define a service and never use it on a request, the service is never created. This saves memory and increases the speed of your application. This also means that there's very little or no performance hit for defining lots of services. Services that are never used are never constructed.

As an added bonus, the *Mailer* service is only created once and the same instance is returned each time you ask for the service. This is almost always the behavior you'll need (it's more flexible and powerful).

You can pass the arguments as list or dict.

You can call functions after object instantiation with:

```
...
services:
    my_mailer:
        class:        myapplication.mailer.Mailer
        arguments:    [sendmail]
        calls:
            - [ set_name, 'Felix Carmona']
            - [ inject_something,  [1, 2, 3]]
            - [ inject_something,  [2, 3]]
            - [ set_location,  {'city': 'Barcelona', 'country': 'Spain'}]
...
```

The available methods for the service container (available in the controller as `self._services`):

- `set(key, value)`: Sets a service object by name.
- `get(key)`: Returns a service object by name.
- `has(key)`: Returns *True* if the service definition exists or if the service object is instantiated, *False* otherwise.
- `remove(key)`: Removes a service object and service definition by name.
- `add(parameters)`: Adds a dict of services objects.
- `keys()`: Returns the services keys.

3.2.3 Using the Parameters to build Services

The creation of new services (objects) via the container is pretty straightforward. Parameters make defining services more organized and flexible:

```
parameters:
    my_mailer_class:      myapplication.mailer.Mailer
    my_mailer_transport:  sendmail

services:
    my_mailer:
        class:        "{{ my_mailer_class }}"
        arguments:    ["{{ my_mailer_transport }}"]
```

The end result is exactly the same as before - the difference is only in how you defined the service. By surrounding the *my_mailer.class* and *my_mailer.transport* strings in double bracket keys (`{{ }}`) signs, the container knows to look

for parameters with those names. Parameters can deep reference other parameters that references other parameters, and will be resolved anyway.

The purpose of parameters is to feed information into services. Of course there was nothing wrong with defining the service without using any parameters. Parameters, however, have several advantages:

- separation and organization of all service "options" under a single parameters key
- parameter values can be used in multiple service definitions

The choice of using or not using parameters is up to you.

3.2.4 Importing Configuration with imports

The service container is built using this single configuration resource (`config/settings_*.yml` by default). All other service configuration must be imported from inside this file (writing they in this file, or including them via the imports directive). This gives you absolute flexibility over the services in your application.

So far, you've placed your *my_mailer* service container definition directly in the application configuration file (e.g. `config/settings_prod.yml`).

First, move the *my_mailer* service container definition into a new yaml file.

```
# config/services.yml
parameters:
    my_mailer_class:      myapplication.mailer.Mailer
    my_mailer_transport:  sendmail

services:
    my_mailer:
        class:        "{{my_mailer._lass}}"
        arguments:    ["{{my_mailer_transport}}"]
```

The definition itself hasn't changed, only its location. Of course the service container doesn't know about the new resource file. Fortunately, you can easily import the resource file using the imports key in the application configuration.

```
# config/settings_prod.yml
imports:
    - services.yml
...
```

The imports directive allows your application to include service container configuration resources from any other location. The resource location, for files, is the relative (from config path) or absolute path to the resource file.

3.2.5 Referencing (Injecting) Services

You can of course also reference services

Start the string with @ to reference a service in YAML.

```
parameters:
    my_mailer:
        class:      myapplication.mailer.Mailer
        transport:  sendmail

services:
    my_mailer:
        class: "{{ my_mailer.class }}"
        arguments:
```

```
            - "{{ my_mailer.transport }}"
    my_mailer_manager:
        class: myapplication.mailer.MailerManager
        arguments:
            - "@my_mailer"
```

the *my_mailer* service will be injected in the *my_mailer_manager*

Note: Use `@@` to escape the `@` symbol. `@@my_mailer` will be converted into the string "`@my_mailer`" instead of referencing the *my_mailer* service.

CHAPTER 4

Testing

Whenever you write a new line of code, you also potentially add new bugs. To build better and more reliable applications, you should test your code using both functional and unit tests.

4.1 Acceptance Tests

Acceptance tests check the integration of all layers of your application (from the request and routing to the final response).

4.2 Your first Acceptance Test

Acceptance tests are simple `test_*.py` files that typically live in the `tests/acceptance` directory of your application. If you want to test the pages handled by your HelloWorldController class, start by creating a new `test_hello_world_controller.py` file with a class that extends a special `apy.testing.ApplicationTestCase` class.

For example:

```
from apy.testing import ApplicationTestCase


class HelloWorldTestCase(ApplicationTestCase):
    def setUp(self):
        self.boot()

    def test_hello_world_text(self):
        response = self.request('GET', 'hello/world')
        self.assertEqual('Hello World!', response.get_content())
```

To run your acceptance tests, the class will configure your application with the `self.boot(application_environment='test', environment_variable_of_config_dir='APPLICATION_C` method.

You will set your `application config dir` in a system environment variable (by default is `APPLICATION_CONFIG_DIR`)

By default your test will look for a `settings_test.yml` (*environment test*) config file in the `config path`, but you can customize it in the first parameter of the `self.boot()`.

The following method

```
self.request(
    method,
    path=None,
    query=None,
    data=None,
    files=None,
    headers=None,
    host='',
    protocol='http',
    remote_ip=None,
    version=None
)
```

allows you to do requests to your application, so this method returns an instance of `apy.http.Response`.

CHAPTER 5

Internals

CHAPTER 6

Contributing

You can contribute to the project in a number of ways. Code is always good, bugs are interesting but tests make your famous!

Bug reports or feature enhancements that include a test are given preferential treatment. So instead of voting for an issue, write a test.

6.1 Docs

We use sphinx to build the docs. make html is your friend, see docstrings for details on params, etc.

6.2 Bug Reports

If you encounter a bug or some surprising behavior, please file an issue on our tracker