fantastico Documentation

Release 0.0.1-b28

Radu Viorel Cosnita

May 02, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Why another python framework?

The main reason for developing a new framework is simple: I want to use it for teaching purposes. I have seen many projects which fail either because of poor coding or because they become legacy very fast. I will not get into details why and what could have been done. It defeats the purpose.

Each piece of code that is being added to fantastico will follow these simple rules:

1. *The code is written because is needed and there is no clean way to achieve the requirement with existing fantastico features*.
2. The code is developed using TDD (Test Driven Development).
3. The code quality is 9+ (reported by pylint).
4. The code coverage is 90%+ (reported by nose coverage).
5. The code is fully documented and included into documentation.

1.1.1 What do you want to teach who?

I am a big fan of Agile practices and currently I own a domain called scrum-expert.ro. This is meant to become a collection of hands on resource of how to develop good software with high quality and in a reasonable amount of time. Resources will cover topics like

1. Incremental development always ready for rollout.
2. TDD (Test Driven Development)
3. XP (eXtreme programming)
4. Scrum
5. Projects setup for Continuous Delivery

and many other topics that are required for delivering high quality software but apparently so many companies are ignoring nowadays.

1.2 Fantastico's initial ideas

- Very fast and pluggable routing engine.
- Easily creation of REST apis.
- Easily publishing of content (dynamic content).

- Easily composition of available content.
- Easily deployment on non expensive infrastructures (AWS, RackSpace).

Once the features above are developed there should be extremely easy to create the following sample applications:

1. Blog development
2. Web Forms development.
3. Personal web sites.

CHAPTER

TWO

GETTING STARTED

2.1 Installation manual

In this section you can find out how to configure fantastico framework for different purposes.

2.1.1 Developing a new fantastico project

Currently fantastico is in early stages so we did not really use it to create new projects. The desired way we want to provide this is presented below:

pip-3.2 install fantastico

Done, now you are ready to follow our tutorials about creating new projects.

2.1.2 Contributing to fantastico framework

Fantastico is an open source MIT licensed project to which any contribution is welcomed. If you like this framework idea and you want to contribute do the following (I assume you are on an ubuntu machine):

```
#. Create a github account.
#. Ask for permissions to contribute to this project (send an email to radu.cosnita@gmail.com) - I w
#. Create a folder where you want to hold fantastico framework files. (e.g worspace_fantastico)
#. cd ~/workspace_fantastico
#. git clone git@github.com:rcosnita/fantastico
#. sudo apt-get install python3-setuptools
#. sh virtual_env/setup_dev_env.sh
#. cd ~/workspace_fantastico
#. git clone git@github.com:rcosnita/fantastico fantastico-doc
#. git checkout gh-pages
```

Now you have a fully functional fantastico workspace. I personally use PyDev and spring toolsuite but you are free to use whatever editor you want. The only rule we follow is *always keep the code stable*. To check the stability of your contribution before commiting the code follow the steps below:

```
#. cd ~/workspace_fantastico/fantastico/fantastico
#. sh run_tests.sh (we expect no failure in here)
#. sh run_pylint.sh (we expect 9+ rated code otherwise the build will fail).
#. cd ~/workspace_fantastico/fantastico
#. export BUILD_NUMBER=1
#. ./build_docs.sh (this will autogenerate documentation).
#. Look into ~/workspace_fantastico/fantastico-doc
#. Here you can see the autogenerated documentation (do not commit this as Jenkins will do this for y
#. Be brave and push your newly awesome contribution.
```

2.2 Fantastico settings

Fantastico is configured using a plain settings file. This file is located in the root of fantastico framework or in the root folder of your project. Before we dig further into configuration options lets see a very simple settings file:

```
class BasicSettings(object):
    @property
    def installed_middleware(self):
        return ["fantastico.middleware.request_middleware.RequestMiddleware",
                "fantastico.middleware.routing_engine.RoutingEngineMiddleware"]

    @property
    def supported_languages(self):
        return ["en_us"]
```

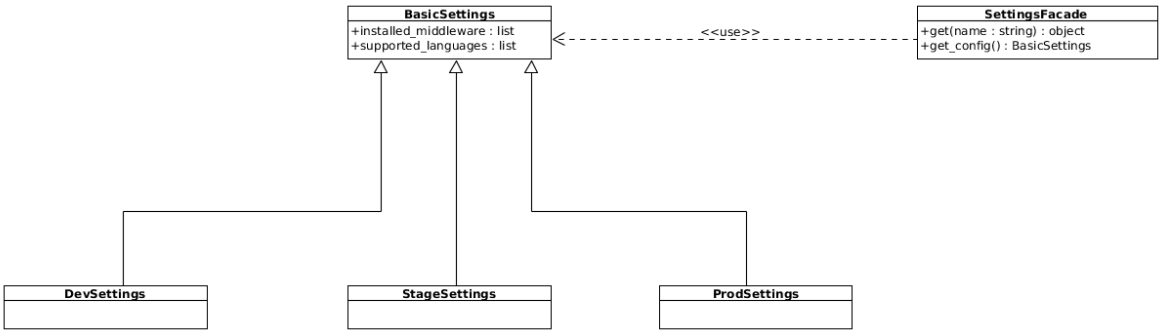
The above code sample represent the minimum required configuration for fantastico framework to run. The order in which middlewares are listed is the order in which they are executed when an http request is made.

2.2.1 Settings API

Below you can find technical information about settings.

class fantastico.settings.**BasicSettings**

This is the core class that describes all available settings of fantastico framework. For convenience all options have default values that ensure minimum functionality of the framework. Below you can find an example of three possible configuration: Dev / Stage / Production.



As you can see, if you want to overwrite basic configuration you simply have to extend the class and set new values for the attributes you want to overwrite.

installed_middleware
Property that holds all installed middlewares.

routes_loaders
This property holds all routes loaders available.

supported_languages
Property that holds all supported languages by this fantastico instance.

2.2.2 Create Dev configuration

Let's imagine you want to create a custom dev configuration for your project. Below you can find the code for this:

```
class DevSettings(BasicSettings):
    @property
    def supported_languages(self):
        return ["en_us", "ro_ro"]
```

The above configuration actually overwrites supported languages. This mean that only en_us is relevant for **Dev** environment. You can do the same for **Stage**, **Prod** or any other custom configuration.

2.2.3 Using a specifc configuration

class fantastico.settings.**SettingsFacade**(*environ=None*)

For using a specific fantastico configuration you need to do two simple steps:

- Set **FANTASTICO_ACTIVE_CONFIG** environment variable to the fully python qualified class name you want to use. E.g: fantastico.settings.BasicSettings
- In your code, you can use the following snippet to access a specific setting:

```
from fantastico.settings import SettingsFacade

print(SettingsFacade().get("installed_middleware"))
```

If no active configuration is set in the fantastico.settings.BasicSettings will be used.

get(*name*)

Method used to retrieve a setting value.

Parameters

- **name** – Setting name.
- **type** – string

Returns The setting value.

Return type object

get_config()

Method used to return the active configuration which is used by this facade.

Return type fantastico.settings.BasicSettings

Returns Active configuration currently used.

2.3 Contribute

Fantastico framework is open source so every contribution is welcome. For the moment we are looking for more developers willing to contribute.

2.3.1 Code contribution

If you want to contribute with code to fantastico framework there are a simple set of rules that you must follow:

- Write unit tests (for the code / feature you are contributing).
- Write integration tests (for the code / feature you are contributing).
- Make sure your code is rated above 9.5 by pylint tool.

- In addition integration tests and unit tests must cover 95% of your code.

In order for each build to remain stable the following hard limits are imposed:

1. Unit tests must cover >= 95% of the code.
2. Integration tests must cover >= 95% of the code.
3. Code must be rated above 9.5 by pylint.
4. Everything must pass.

When you push on master a set of jobs are cascaded executed:

1. Run all unit tests job.
2. Run all integration tests job (only if unit tests succeeds).
3. Generate documentation and publish it (only if integration tests job succeeds).

You can follow the above build process by visiting Jenkins build. Login with your github account and everything should work smoothly.

In the end do not forget that in Fantastico framework we love to develop against a **stable** base. We really think code will have high quality and zero bugs.

Writing unit tests

For better understanding how to write unit tests see the documentation below:

class fantastico.tests.base_case.**FantasticoUnitTestsCase**(*methodName='runTest'*)
This is the base class that must be inherited by each unit test written for fantastico.

```
class SimpleUnitTest(FantasticoUnitTestsCase):
    def init(self):
        self._msg = "Hello world"

    def test_simple_flow_ok(self):
        self.assertEqual("Hello world", self._msg)
```

Writing integration tests

For better understanding how to write integration tests see the documentation below:

class fantastico.tests.base_case.**FantasticoIntegrationTestCase**(*methodName='runTest'*)
This is the base class that must be inherited by each integration test written for fantastico.

```
class SimpleIntegration(FantasticoIntegrationTestCase):
    def init(self):
        self.simple_class = {}

    def test_simple_ok(self):
        def do_stuff(env, env_cls):
            self.assertEqual(simple_class[env], env_cls)

        self._run_test_all_envs(do_stuff)
```

_envs
Private property that holds the environments against which we run the integration tests.

_run_test_all_envs(*callable_obj*)

This method is used to execute a callable block of code on all environments. This is extremely useful for avoid boiler plate code duplication and executing test logic against all environments.

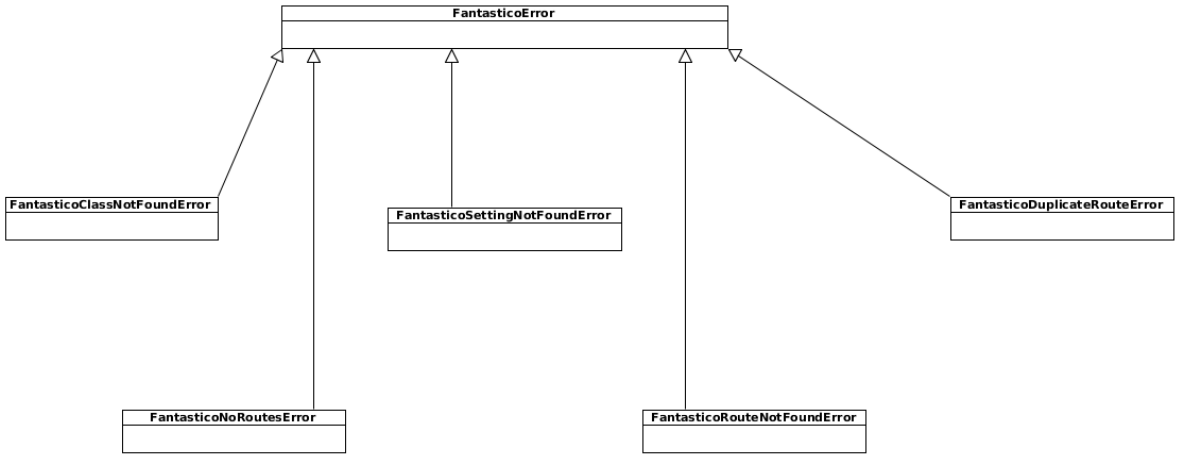
CHAPTER

THREE

FANTASTICO FEATURES

3.1 Exceptions hierarchy

class fantastico.exceptions.**FantasticoError**



FantasticoError is the base of all exceptions raised within fantastico framework. It describe common attributes that each concrete fantastico exception must provide. By default all fantastico exceptions inherit FantasticoError exception. We do this because each raised unhandled FantasticoError is map to a specific exception response. This strategy guarantees that at no moment errors will cause fantastico framework wsgi container to crash.

class fantastico.exceptions.**FantasticoClassNotFoundError**
This exception is raised whenever code tries to dynamically import and instantiate a class which can not be resolved.

class fantastico.exceptions.**FantasticoSettingNotFoundError**
This exception is raised whenever code tries to obtain a setting that is not available in the current fantastico configuration.

class fantastico.exceptions.**FantasticoDuplicateRouteError**
This exception is usually raised by routing engine when it detects duplicate routes.

class fantastico.exceptions.**FantasticoNoRoutesError**
This exception is usually raised by routing engine when no loaders are configured or no routes are registered.

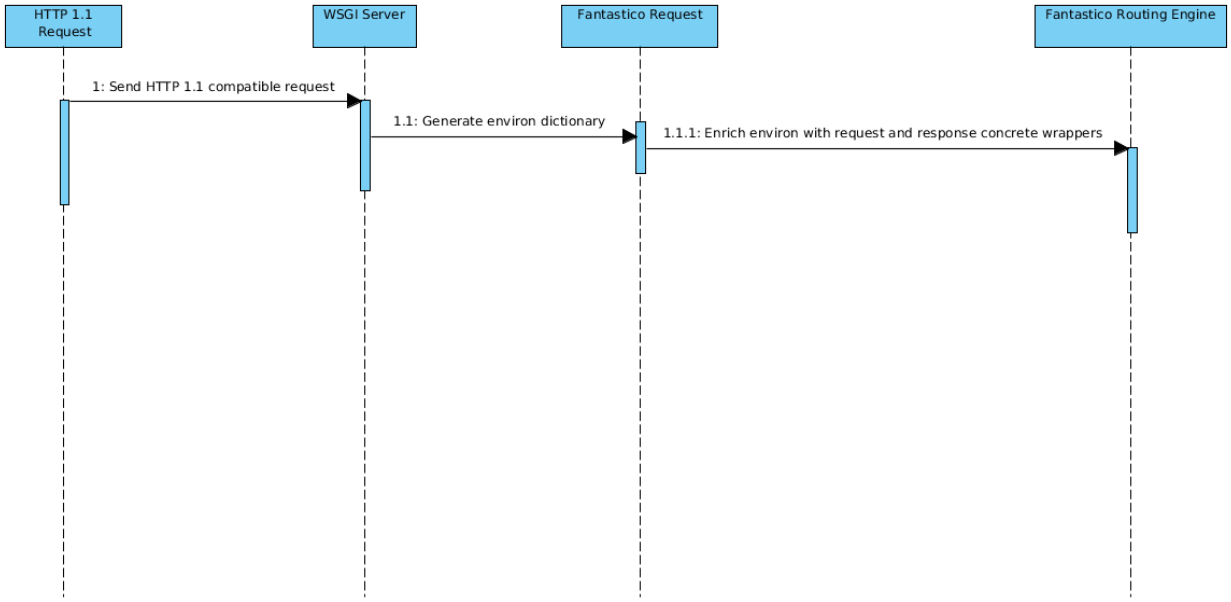
class fantastico.exceptions.**FantasticoRouteNotFoundError**
This exception is usually raised by routing engine when a requested url is not registered.

3.2 Request lifecycle

In this document you can find how a request is processed by fantastico framework. By default WSGI applications use a dictionary that contains various useful keys:

- HTTP Headers
- HTTP Cookies
- Helper keys (e.g file wrapper).

In fantastico we want to hide the complexity of this dictionary and allow developers to use some standardized objects. Fantastico framework follows a Request / Response paradigm. This mean that for every single http request only one single http response will be generated. Below, you can find a simple example of how requests are processed by fantastico framework:



In order to not reinvent the wheels fantastico relies on WebOb python framework in order to correctly generate request and response objects. For more information read WebOB Doc.

3.2.1 Request context

In comparison with WebOb **Fantastico** provides a nice improvement. For facilitating easy development of code, each fantastico request has a special attribute called context. Below you can find the attributes of a request context object:

- settings facade (*Fantastico settings*)
- session (not yet supported)
- **language** The current preferred by user. This is determined based on user lang header.
- user (not yet supported)

class `fantastico.middleware.request_context.`**`RequestContext`**(*settings*, *language*)

This class holds various attributes useful giving a context to an http request. Among other things we need to be able to access current language, current session and possible current user profile.

`language`

Property that holds the current language that must be used during this request.

settings
Property that holds the current settings facade used for accessing fantastico configuration.

3.2.2 Obtain request language

class fantastico.locale.language.**Language**(*code*)
Class used to define how does language object looks like. There are various use cases for using language but the simplest one is in request context object:

```
language = request.context.language

if language.code = "en_us":
   print("English (US) language").
else:
   raise Exception("Language %s is not supported." % language.code)
```

code
Property that holds the language code. This is readonly because once instantiated we mustn't be able to change it.

3.2.3 Obtain settings using request

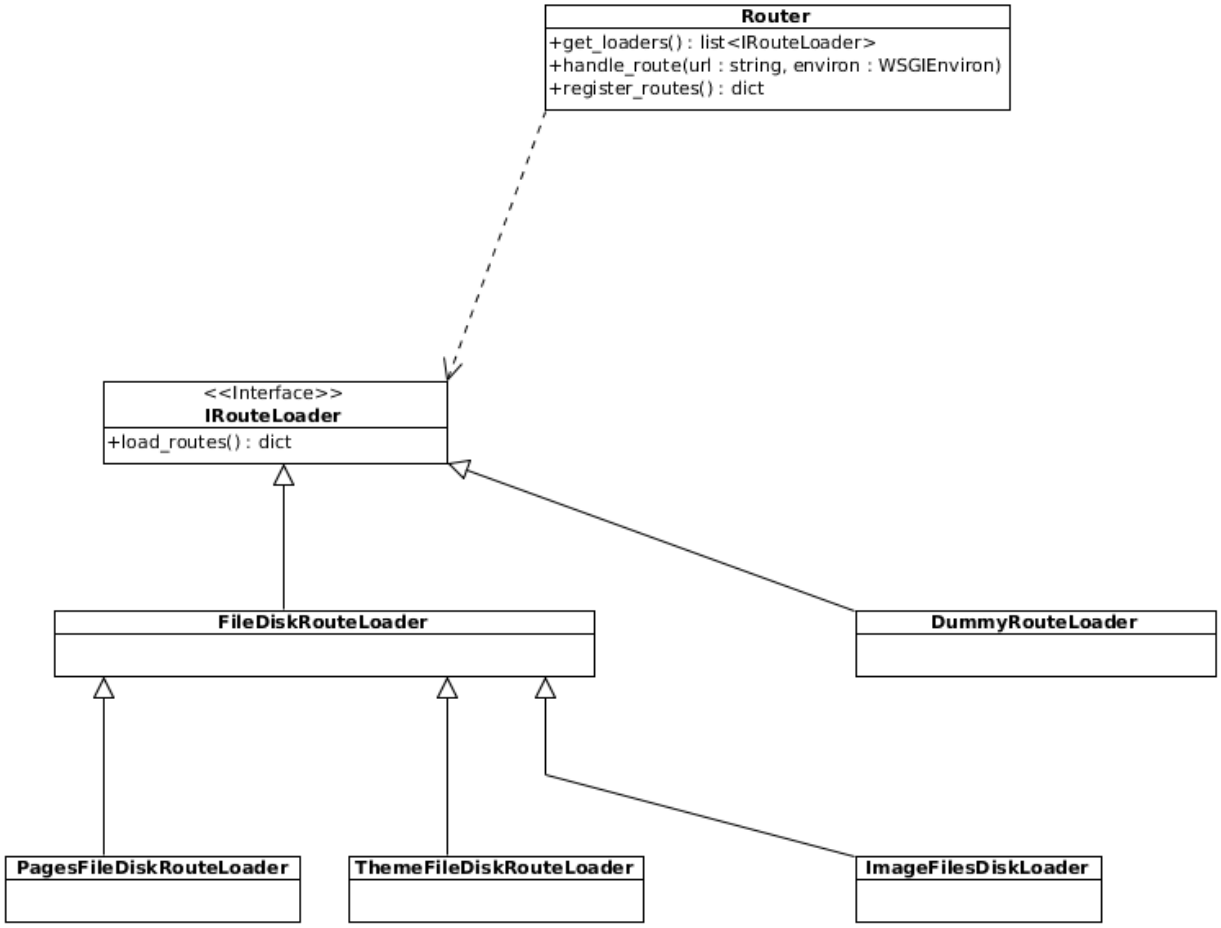
It is recommended to use *request.context* object to obtain fantastico settings. This hides the complexity of choosing the right configuration and accessing attributes from it.

```
installed_middleware = request.context.settings.get("installed_middleware")

print(installed_middleware)
```

For more information about how to configure **Fantastico** please read *Fantastico settings*.

3.3 Routing engine



Fantastico routing engine is design by having extensibility in mind. Below you can find the list of concerns for routing engine:

1. Support multiple sources for routes.
2. Load all available routes.
3. Select the controller that can handle the request route (if any available).

class `fantastico.routing_engine.router.`**`Router`**(*settings_facade=<class 'fantastico.settings.SettingsFacade'>*)

This class is used for registering all available routes by using all registered loaders.

`get_loaders`()

Method used to retrieve all available loaders. If loaders are not currently instantiated they are by these method. This method also supports multi threaded workers mode of wsgi with really small memory footprint. It uses an internal lock so that it makes sure available loaders are instantiated only once per wsgi worker.

`handle_route`(*url*, *environ*)

Method used to identify the given url method handler. It enrich the environ dictionary with a new entry that holds a controller instance and a function to be executed from that controller.

register_routes()
Method used to register all routes from all loaders. If the loaders are not yet initialized this method will first load all available loaders and then it will register all available routes. Also, this method initialize available routes only once when it is first invoked.

3.3.1 Routes loaders

Fantastico routing engine is designed so that routes can be loaded from multiple sources (database, disk locations, and others). This give huge extensibility so that developers can use Fantastico in various scenarios:

- Create a CMS that allows people to create new pages (mapping between page url / controller) is hold in database. Just by adding a simple loader in which the business logic is encapsulated allows routing engine extension.
- Create a blog that loads articles from disk.

I am sure you can find other use cases in which you benefit from this extension point.

3.3.2 How to write a new route loader

Before digging in further details see the RouteLoader class documentation below:

class fantastico.routing_engine.routing_loaders.**RouteLoader**(*settings_facade*)
This class provides the contract that must be provided by each concrete implementation. Each route loader is responsible for implementing its own business logic for loading routes.

```
class DummyRouteLoader(RouteLoader):
    def __init__(self, settings_facade=SettingsFacade):
        self_settings_facade = settings_facade()

    def load_routes(self):
        return {"/index.html", "fantastico.plugins.static_assets.StaticAssetsController.resolve_
                "/images/image.png", "fantastico.plugins.static_assets.StaticAssetsController.re
```

load_routes()
This method must be overriden by each concrete implementation so that all loaded routes can be handled by fantastico routing engine middleware.

As you can, each concrete route loader receives in the constructor settings facade that can be used to access fantastico settings. In the code example above, DummyRouteLoader maps a list of urls to a controller method that can be used to render it. Keep in mind that a route loader is a stateless component and it can't in anyway determine the wsgi environment in which it is used. In addition this design decision also make sure clear separation of concerned is followed.

Once your **RouteLoader** implementation is ready you must register it into settings profile. The safest bet is to add it into BaseSettings provider. For more information read *Fantastico settings*.

3.3.3 Configuring available loaders

You can find all available loaders for the framework configured in your settings profile. You can find below a sample configuration of available loaders:

```
class CustomSettings(BasicSettings):
    @property
    def routes_loaders(self):
        return ["fantastico.routing_engine.custom_loader.CustomLoader"]
```

The above configuration tells **Fantastico routing engine** that only CustomLoader is a source of routes. If you want to learn more about multiple configurations please read *Fantastico settings*.

3.3.4 DummyRouteLoader

class fantastico.routing_engine.dummy_routeloader.**DummyRouteLoader**(*settings_facade*)

This class represents an example of how to write a route loader. **DummyRouteLoader** is available in all configurations and it provides a single route to the routing engine: */dummy/route/loader/test*. Integration tests rely on this loader to be configured in each available profile.

CHAPTER

FOUR

BUILD STATUS

If you want to see the current build status of the project visit Build status.

CHAPTER

FIVE

LICENSE

Symbols

B

C

D

F

G

H

I

L

R

S