# fantastico Documentation

***Release 0.0.1-b43***

**Radu Viorel Cosnita**

May 06, 2013

**1 Introduction** **1**
1.1 Why another python framework? . . . 1
1.2 Fantastico's initial ideas . . . 1

**2 Getting started** **3**
2.1 Installation manual . . . 3
2.2 Fantastico settings . . . 4
2.3 Contribute . . . 5
2.4 Development mode . . . 7

**3 Fantastico features** **9**
3.1 Exceptions hierarchy . . . 9
3.2 Request lifecycle . . . 10
3.3 Routing engine . . . 12
3.4 Model View Controller . . . 14

**4 How to articles** **17**
4.1 MVC How to . . . 17

**5 Build status** **19**

**6 License** **21**

**Index** **23**

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_middleware.RoutingMiddleware"]

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

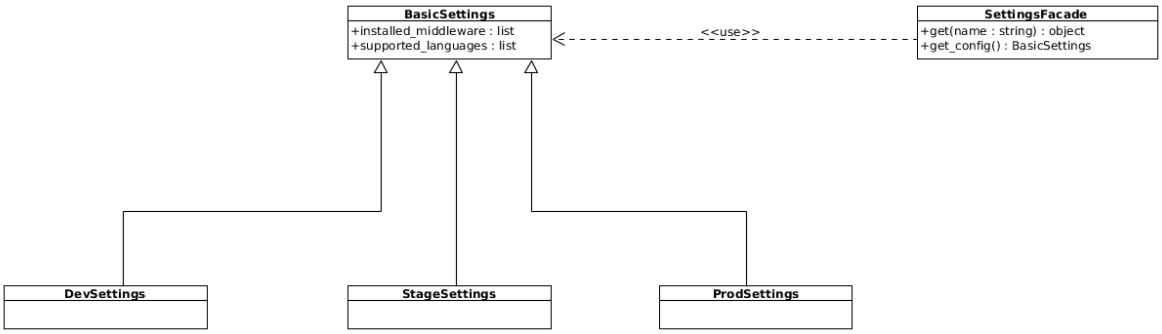
The above code sample represent the minimum required configuration for fantastico framework to run. The order in which middlewares are listed is the order in which they are executed when an http request is made.

### 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.

**dev_server_host**

This property holds development server hostname. By default this is localhost.

**dev_server_port**

This property holds development server port. By default this is 12000.

**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 cleanup(self):
        self.simple_class = None

    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)
```

If you used this class you don't have to mind about restoring call methods from each middleware once they are wrapped by fantastico app. This is a must because otherwise you will crash other tests.

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

**_restore_call_methods**()
This method restore original call methods to all affected middlewares.

**_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.

**_save_call_methods**(*middlewares*)
This method save all call methods for each listed middleware so that later on they can be restored.

## 2.4 Development mode

**Fantastico** framework is a web framework designed to be developers friendly. In order to simplify setup sequence, fantastico provides a standalone WSGI compatible server that can be started from command line. This server is fully compliant with WSGI standard. Below you can find some easy steps to achieve this:

1. Goto fantastico framework or project location
2. sh run_dev_server.sh

This is it. Now you have a running fantastico server on which you can test your work.

By default, **Fantastico** dev server starts on port 12000, but you can customize it from `fantastico.settings.BasicSettings`.

### 2.4.1 Hot deploy

Currently, this is not implemented, but it is on todo list on short term.

### 2.4.2 API

For more information about Fantastico development server see the API below.

**class** fantastico.server.dev_server.**DevServer**(*settings_facade=<class 'fantastico.settings.SettingsFacade'>*)
This class provides a very simple wsgi http server that embeds Fantastico framework into it. As developer you can use it to simply test your new components.

**start**(*build_server=<function make_server at 0x4a3ea68>, app=<class 'fantastico.middleware.fantastico_app.FantasticoApp'>*)
This method starts a WSGI development server. All attributes like port, hostname and protocol are read from configuration file.

**started**
Property used to tell if development server is started or not.

**stop**()
This method stops the current running server (if any available).

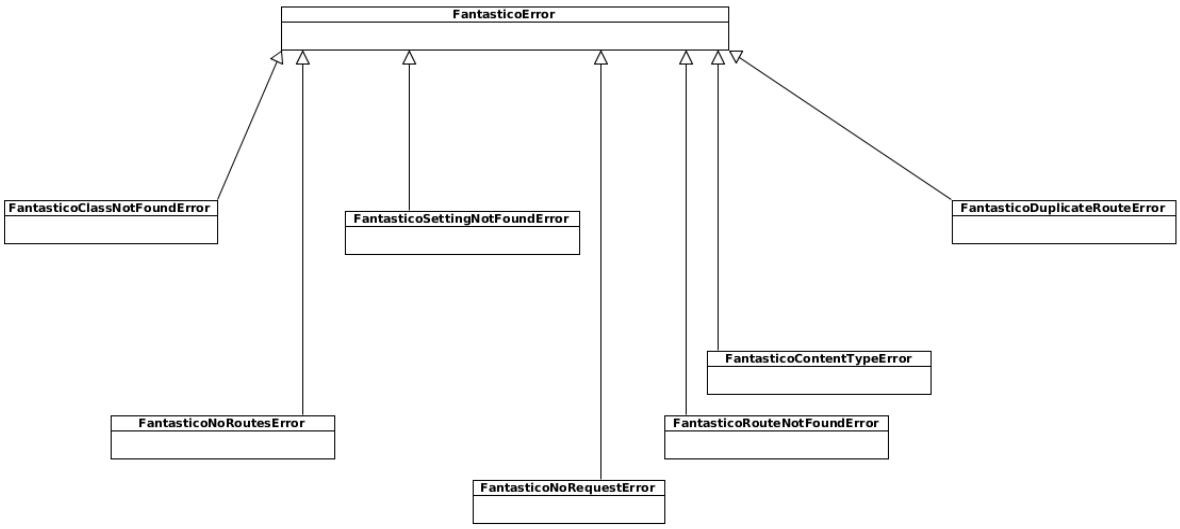
CHAPTER

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.

**class** fantastico.exceptions.**FantasticoNoRequestError**
This exception is usually raised when some components try to use fantastico.request from WSGI environ before fantastico.middleware.request_middleware.RequestMiddleware was executed.

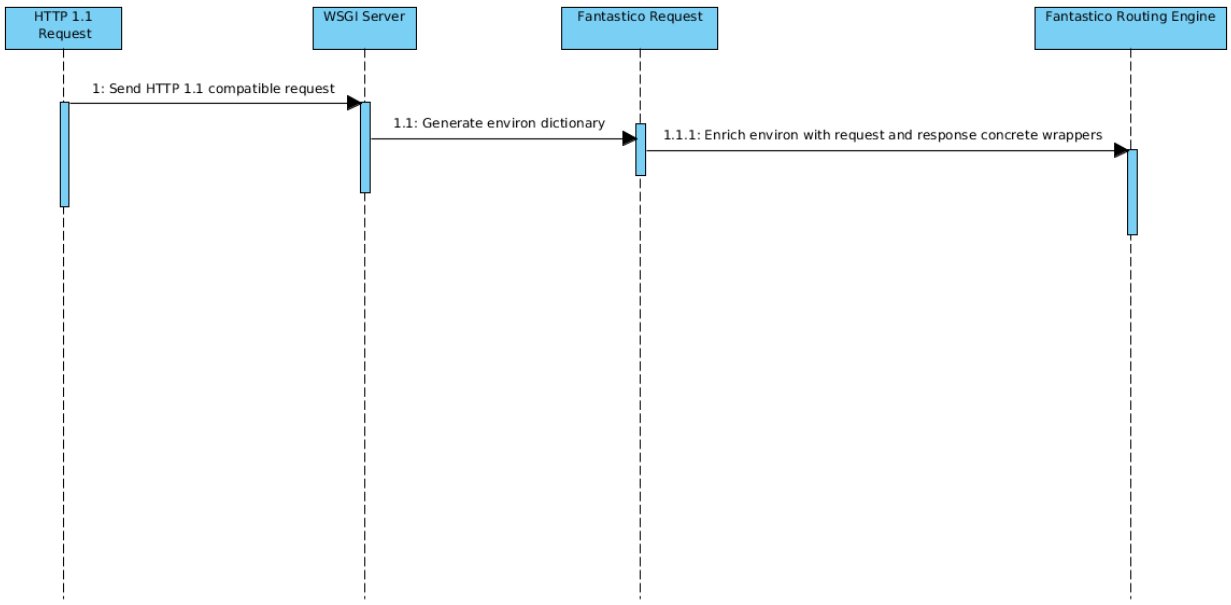
**class** fantastico.exceptions.**FantasticoContentTypeError**
This exception is usually thrown when a mismatch between request content type and received content type differ. In Fantastico we think it's mandatory to fulfill requests correctly and to take in consideration sent headers.

## 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 middleware

To have very good control of how WSGI environ is wrapped into **WebOb request** object a middleware component is configured. This is the first middleware that is executed for every single http request.

**class** fantastico.middleware.request_middleware.**RequestMiddleware**(*app*)
This class provides the middleware responsible for converting wsgi environ dictionary into a request. The result is saved into current WSGI environ under key **fantastico.request**.

### 3.2.2 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.

### 3.2.3 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.

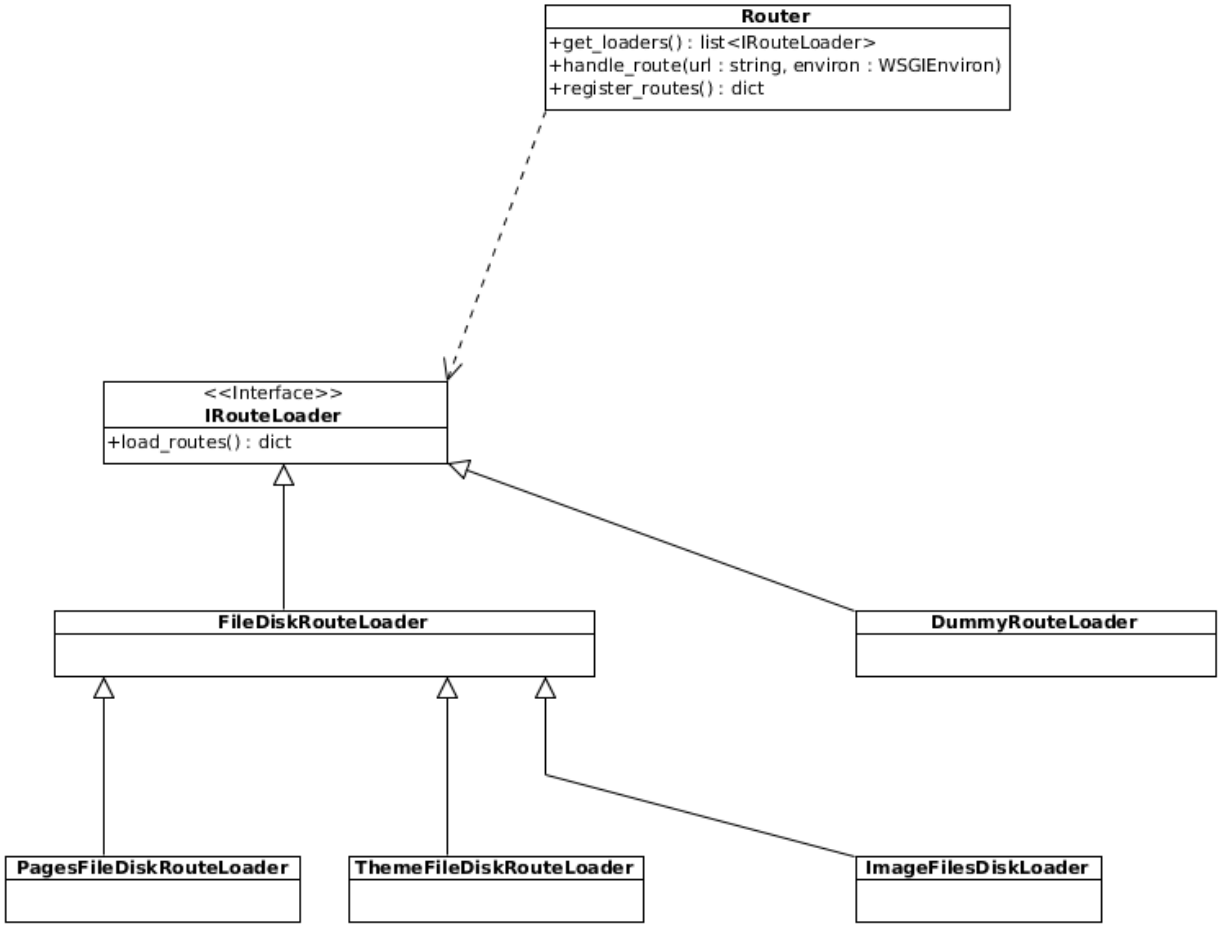
### 3.2.4 Obtain settings using request

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.

**display_test**(*request*)

This method handles **/dummy/route/loader/test route**. It is expected to receive a response with status code 400. We do this for being able to test rendering and also avoid false positive security scans messages.

### 3.3.5 Routing middleware

**Fantastico** routing engine is designed as a standalone component. In order to be able to integrate it into Fantastico request lifecycle (:doc:/features/request_response.) we need an adapter component.

**class** fantastico.middleware.routing_middleware.**RoutingMiddleware**(*app*, *router_cls=<class 'fantastico.routing_engine.router.Router'>*)

Class used to integrate routing engine fantastico component into request / response lifecycle. This middleware is responsible for:

1. instantiating the router component and make it available to other components / middlewares through WSGI environment.
2. register all configured fantastico loaders (`fantastico.routing_engine.router.Router.get_loaders()`).
3. register all available routes (`fantastico.routing_engine.router.Router.register_routes()`).
4. handle route requests (`fantastico.routing_engine.router.Router.handle_route()`).

It is important to understand that routing middleware assume a **WebOb request** available into WSGI environ. Otherwise, `fantastico.exceptions.FantasticoNoRequestError` will be thrown. You can read more about request middleware at *Request lifecycle*.

## 3.4 Model View Controller

**Fantastico** framework provides quite a powerful model - view - controller implementation. Here you can find details about design decisions and how to benefit from it.

### 3.4.1 Classic approach

Usually when you want to work with models as understood by MVC pattern you have in many cases boiler plate code:

1. Write your model class (or entity)
2. Write a repository that provides various methods for this model class.
3. Write a facade that works with the repository.
4. Write a web service / page that relies on the facade.

5. Write one or multiple views.

As this is usually a good in theory, in practice you will see that many methods from facade are converting a data transfer object to an entity and pass it down to repository.

### 3.4.2 Fantastico approach

**Fantastico** framework provides an alternative to this classic approach (you can still work in the old way if you really really want).

**class** fantastico.mvc.controller_decorators.**Controller**(*url*, *method='GET'*, *models=None*)

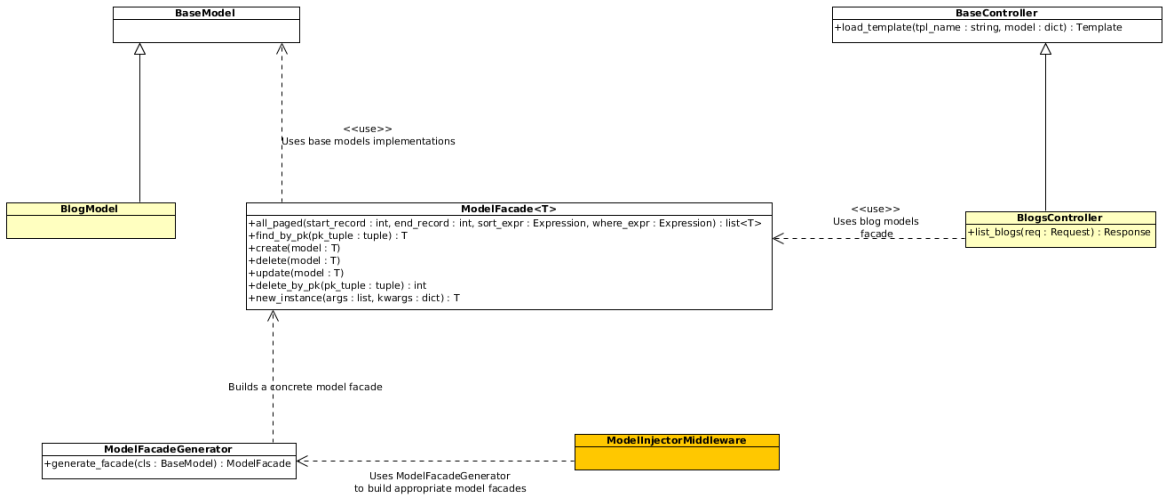
This class provides a decorator for magically registering methods as route handlers. This is an extremely important piece of Fantastico framework because it simplifies the way you as developer can define mapping between a method that must be executed when an http request to an url is made:

```
@Controller(url="/blogs/", method="GET", models={"Blog": "fantastico.plugins.blog.models.blog.Bl
def list_blogs(self, request):
    Blog = request.models.Blog

    blogs = Blog.all_paged(start_record=1, end_record=0, sort_expr=[asc(Blog.create_date), desc(

    return Response(blogs)
```

The above code assume the following:

1. As developer you created a model called blog (this is already mapped to some sort of storage).
2. Fantastico framework generate the facade automatically (and you never have to know anything about underlining repository).
3. Fantastico framework takes care of data conversion.
4. As developer you create the method that knows how to handle **/blog/** url.
5. Write your view.

Below you can find the design for MVC provided by **Fantastico** framework:




**fn_handler**

This property retrieves the method which is executed by this controller.

**classmethod get_registered_routes**()
This class methods retrieve all registered routes through Controller decorator.

**method**
This property retrieves the method(s) for which this controller can be invoked. Most of the time only one value is retrieved.

**models**
This property retrieves all the models required by this controller in order to work correctly.

**url**
This property retrieves the url used when registering this controller.

If you want to find more details and use cases for controller read *Controller* section.

### 3.4.3 Model

A model is a very simple object that inherits `fantastico.mvc.models.BaseModel`.

### 3.4.4 View

A view can be a simple html plain file or html + jinja2 enriched support. You can read more about **Jinja2** here. Usually, if you need some logical block statements in your view (if, for, ...) it is easier to use jinja 2 template engine. The good news is that you can easily embed jinja 2 markup in your views and it will be rendered automatically.

### 3.4.5 Controller

A controller is the *brain*; it actually combines a model execute some business logic and pass data to the desired view that needs to be rendered. In some cases you don't really need view in order to provide the logic you want:

- A REST Web service.
- A RSS feed provider.
- A file download service

Though writing REST services does not require a view, you can load external text templates that might be useful for assembling the response:

- An invoice generator service
- An xml file that must be filled with product data
- A vCard. export service.

If you want to read a small tutorial and to start coding very fast on Fantastico MVC read *MVC How to*. Controller API is documented `fantastico.mvc.controller_decorator.Controller`.

CHAPTER

FOUR

# HOW TO ARTICLES

## 4.1 MVC How to

In this article you can see how to assemble various pieces together in order to create a feature for a virtual blog application. If you follow this step by step guide in the end you will have a running blog which can list all posts.

### 4.1.1 Code the model

Below you can find how to easily create **post** model.

1. Create a new package called **blog**
2. Create a new package called **blog.models**
3. Create a new module called posts and paste the following code into it:

```
class Post(BaseModel):
    __tablename__ = "posts"

    id = Column("id", Integer, primary_key=True)
    blog_id =
    title = Column("title", String(150))
    tags = Column("tags", String(150))
    created_date = Column("registered_date", DateTime(), default=datetime.now)
    content = Column("content", Text(100))
```

Now you have a fully functional post model mapped over **posts** table.

### 4.1.2 Code the controller

1. Create a new package called **blog.controllers**
2. Create a new module called **blog_controller** and paste the following code into it:

```
@Controller(url="/blogs/1/posts/", method="GET",
            models={"Post": "fantastico.plugins.blog.models.posts.Post"])
def list_blog_posts(self, request):
    Post = request.models.Post

    blog_id = int(request.params.id)

    posts = Post.all_paged(start_record=1,
                           sort_expr=[asc(Post.created_date), desc(Post.title)],
                           where_expr=[eq_(Post.blog_id, blog_id)])
```

```
        response = Response()
        response.text = self.load_template("/posts_listing.html",
                                           {"posts": posts,
                                            "blog_id": blog_id})

        return response
```

Now you have a fully functional controller that will list all posts.

### 4.1.3 Code the view

1. Create a new folder called **blog.views**
2. Create a new view under **blog.views** called *posts_listing.html* and paste the following code into it:

```
<html>
    <head>
        <title>List all available posts from blog {{blog_id}}</title>
    </head>

    <body>
        <ul>
        {% for post in posts %}
            <li>{{post.title}} | {{post.created_date}}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

### 4.1.4 Test your application

1. Start fantastico dev server by executing script **run_dev_server.sh** (*Development mode*)
2. Open a browser and visit http://localhost:12000/blogs/1/posts.

CHAPTER

FIVE

# BUILD STATUS

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

CHAPTER

SIX

# LICENSE

## Symbols

_envs (fantastico.tests.base_case.FantasticoIntegrationTestCase attribute), 7
_restore_call_methods() (fantastico.tests.base_case.FantasticoIntegrationTestCase method), 7
_run_test_all_envs() (fantastico.tests.base_case.FantasticoIntegrationTestCase method), 7
_save_call_methods() (fantastico.tests.base_case.FantasticoIntegrationTestCase method), 7

## B

BasicSettings (class in fantastico.settings), 4

## C

code (fantastico.locale.language.Language attribute), 11
Controller (class in fantastico.mvc.controller_decorators), 15

## D

dev_server_host (fantastico.settings.BasicSettings attribute), 4
dev_server_port (fantastico.settings.BasicSettings attribute), 4
DevServer (class in fantastico.server.dev_server), 7
display_test() (fantastico.routing_engine.dummy_routeloader.DummyRouteLoader method), 14
DummyRouteLoader (class in fantastico.routing_engine.dummy_routeloader), 14

## F

FantasticoClassNotFoundError (class in fantastico.exceptions), 9
FantasticoContentTypeError (class in fantastico.exceptions), 10
FantasticoDuplicateRouteError (class in fantastico.exceptions), 9
FantasticoError (class in fantastico.exceptions), 9
FantasticoIntegrationTestCase (class in fantastico.tests.base_case), 6
FantasticoNoRequestError (class in fantastico.exceptions), 9
FantasticoNoRoutesError (class in fantastico.exceptions), 9
FantasticoRouteNotFoundError (class in fantastico.exceptions), 9
FantasticoSettingNotFoundError (class in fantastico.exceptions), 9
FantasticoUnitTestsCase (class in fantastico.tests.base_case), 6
fn_handler (fantastico.mvc.controller_decorators.Controller attribute), 15

## G

get() (fantastico.settings.SettingsFacade method), 5
get_config() (fantastico.settings.SettingsFacade method), 5
get_loaders() (fantastico.routing_engine.router.Router method), 12
get_registered_routes() (fantastico.mvc.controller_decorators.Controller class method), 15

## H

handle_route() (fantastico.routing_engine.router.Router method), 12

## I

installed_middleware (fantastico.settings.BasicSettings attribute), 4

## L

Language (class in fantastico.locale.language), 11
language (fantastico.middleware.request_context.RequestContext attribute), 11
load_routes() (fantastico.routing_engine.routing_loaders.RouteLoader method), 13

## M

method (fantastico.mvc.controller_decorators.Controller attribute), 16

models (fantastico.mvc.controller_decorators.Controller attribute), 16

## R

register_routes() (fantastico.routing_engine.router.Router method), 12
RequestContext (class in fantastico.middleware.request_context), 11
RequestMiddleware (class in fantastico.middleware.request_middleware), 10
RouteLoader (class in fantastico.routing_engine.routing_loaders), 13
Router (class in fantastico.routing_engine.router), 12
routes_loaders (fantastico.settings.BasicSettings attribute), 4
RoutingMiddleware (class in fantastico.middleware.routing_middleware), 14

## S

settings (fantastico.middleware.request_context.RequestContext attribute), 11
SettingsFacade (class in fantastico.settings), 5
start() (fantastico.server.dev_server.DevServer method), 7
started (fantastico.server.dev_server.DevServer attribute), 7
stop() (fantastico.server.dev_server.DevServer method), 7
supported_languages (fantastico.settings.BasicSettings attribute), 4

## U

url (fantastico.mvc.controller_decorators.Controller attribute), 16