fantastico Documentation

Release 0.0.1-b65

Radu Viorel Cosnita

May 31, 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_middleware.RoutingMiddleware"]

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

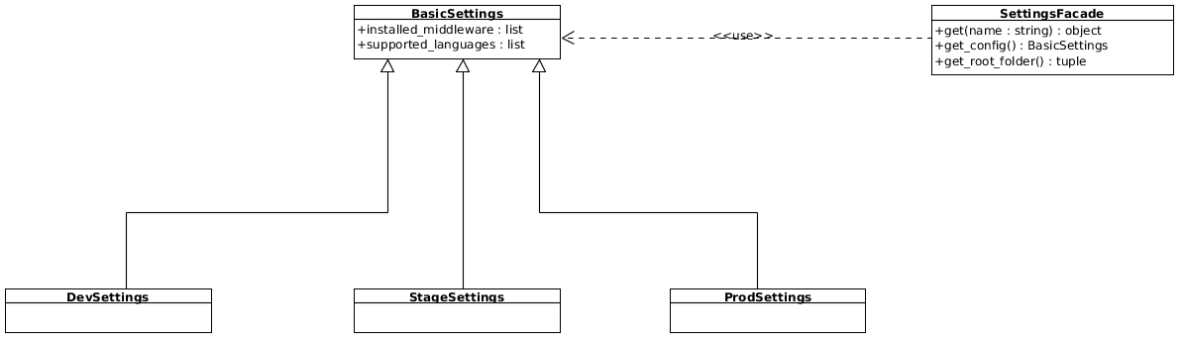
The above code sample represent the minimum required configuration for fantastico framework to run. The order in which middlewares are listed is the order in which they are executed when an http request is made.

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.

database_config

This property holds the configuration of database. It is recommended to have all environment configured the same. An exception can be done for host but the rest must remain the same. Below you can find an example of functional configuration:

```
config = {"drivername": "mysql+mysqlconnector",
          "username": "fantastico",
          "password": "12345",
          "port": 3306,
          "host": "localhost",
          "database": "fantastico",
          "additional_params": {"charset": "utf8"},
          "show_sql": True}
```

As you can see, in your configuration you can influence many attributes used when configuring the driver / database. **show_sql** key tells orm engine from **Fantastico** to display all generated queries.

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.

templates_config
This property holds configuration of templates rendering engine. For the moment this influence how Jinja2 acts.

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.

get_root_folder()
Method used to return the root folder of the current fantastico project (detected starting from settings) profile 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)
```

_get_class_root_folder()
This methods determines the root folder under which the test is executed.

_get_root_folder()
This method determines the root folder under which core is executed.

classmethod setup_once()
This method is overriden in order to correctly mock some dependencies:

- fantastico.mvc.controller_decorators.Controller

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.

class fantastico.server.tests.itest_dev_server.**DevServerIntegration**(*methodName='runTest'*)
This class provides the foundation for writing integration tests that do http requests against a fantastico server.

```
class DummyLoaderIntegration(DevServerIntegration):
    def init(self):
        self._exception = None
```

```
    def test_server_runs_ok(self):
        def request_logic(server):
            request = Request(self._get_server_base_url(server, DummyRouteLoader.DUMMY_ROUTE))
            with self.assertRaises(HTTPError) as cm:
                urllib.request.urlopen(request)

            self._exception = cm.exception

        def assert_logic(server):
            self.assertEqual(400, self._exception.code)
            self.assertEqual("Hello world.", self._exception.read().decode())

        self._run_test_all_envs(lambda env, settings_cls: self._run_test_against_dev_server(requ

    self._run_test_all_envs(lambda env, settings_cls: self._run_test_against_dev_server(request_
```

As you can see from above listed code, when you write a new integration test against Fantastico server you only need to provide the request logic and assert logic functions. Request logic is executed while the server is up and running. Assert logic is executed after the server has stopped.

_check_server_started(*server*)
This method holds the sanity checks to ensure a server is started correctly.

_get_server_base_url(*server*, *route*)
This method returns the absolute url for a given relative url (route).

_run_test_against_dev_server(*request_logic*, *assert_logic=None*)
This method provides a template for writing integration tests that requires a development server being active. It accepts a request logic (code that actually do the http request) and an assert logic for making sure code is correct.

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 0x517c9e0>*, *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).

2.4.3 Database config

Usually you will use **Fantastico** framework together with a database. When we develop new core features of **Fantastico** we use a sample database for integration. You can easily use it as well to play around:

1. Goto fantastico framework location
2. export MYSQL_PASSWD=***** (your mysql password)
3. export MYSQL_HOST=<hostname> (your mysql hostname: e.g localhost)
4. sh run_setup_db.sh

run_setup_db.sh create an initial fantastico database and a user called fantastico identified by **12345** password. After database is successfully created, it scans for all available **module_setup.sql** files and execute them against newly created database.

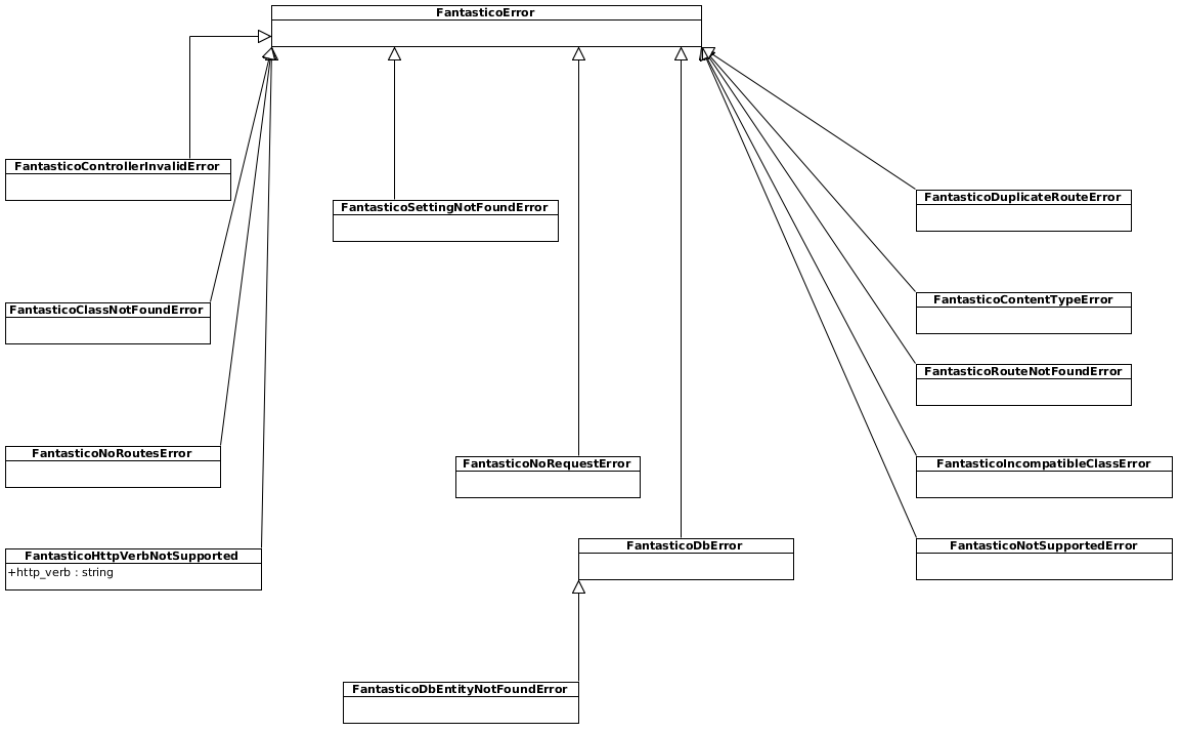
CHAPTER

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.**FantasticoControllerInvalidError**

This exception is raised whenever a method is decorated with fantastico.mvc.controller_decorators.Controlle and the number of arguments is not correct. Usually developer forgot to add request as argument to the controller.

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.**FantasticoNotSupportedError**
This exception is raised whenever code tries to do an operation that is not supported.

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 accept and response content type. In Fantastico we think it's mandatory to fulfill requests correctly and to take in consideration sent headers.

class fantastico.exceptions.**FantasticoHttpVerbNotSupported**(*http_verb*)
This exception is usually thrown when a route is accessed with an http verb which does not support.

http_verb
This property returns the http verb that caused the problems.

class fantastico.exceptions.**FantasticoTemplateNotFoundError**
This exception is usually thrown when a controller tries to load a template which it does not found.

class fantastico.exceptions.**FantasticoIncompatibleClassError**
This exception is usually thrown when we want to decorate / inject / mixin a class into another class that does not support it. For instance, we want to build a fantastico.mvc.model_facade.ModelFacade with a class that does not extend **BASEMODEL.**

class fantastico.exceptions.**FantasticoDbError**
This exception is usually thrown when a database exception occurs. For one good example where this is used see fantastico.mvc.model_facade.ModelFacade.

class fantastico.exceptions.**FantasticoDbNotFoundError**
This exception is usually thrown when an entity does not exist but we try to update it. For one good example where this is used see fantastico.mvc.model_facade.ModelFacade.

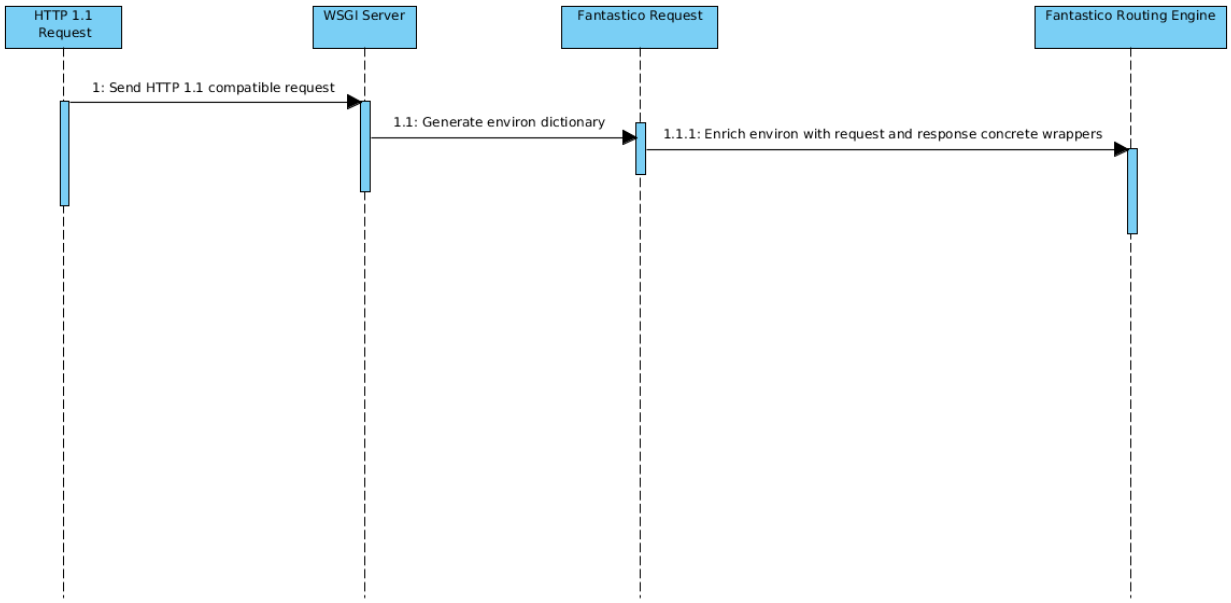
3.2 Request lifecycle

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.

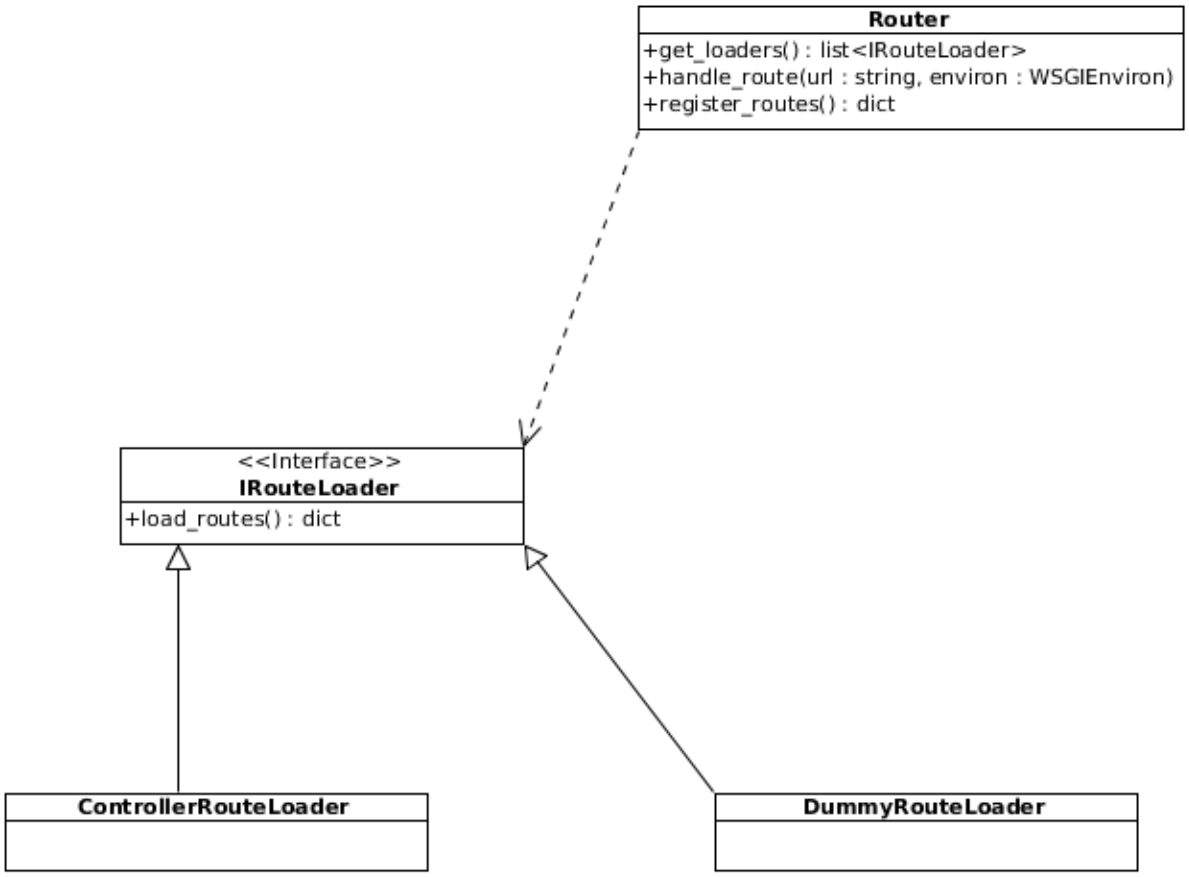
3.2.4 Obtain settings using request

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):
        self_settings_facade = settings_facade

    def load_routes(self):
        return {"/index.html": {"method": "fantastico.plugins.static_assets.StaticAssetsControll
                                "http_verbs": ["GET"]},
                "/images/image.png": {"method": "fantastico.plugins.static_assets.StaticAssetsCo
                                      "http_verbs": ["GET"]}
```

`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*, *model_facade=<class 'fantastico.mvc.model_facade.ModelFacade'>*)

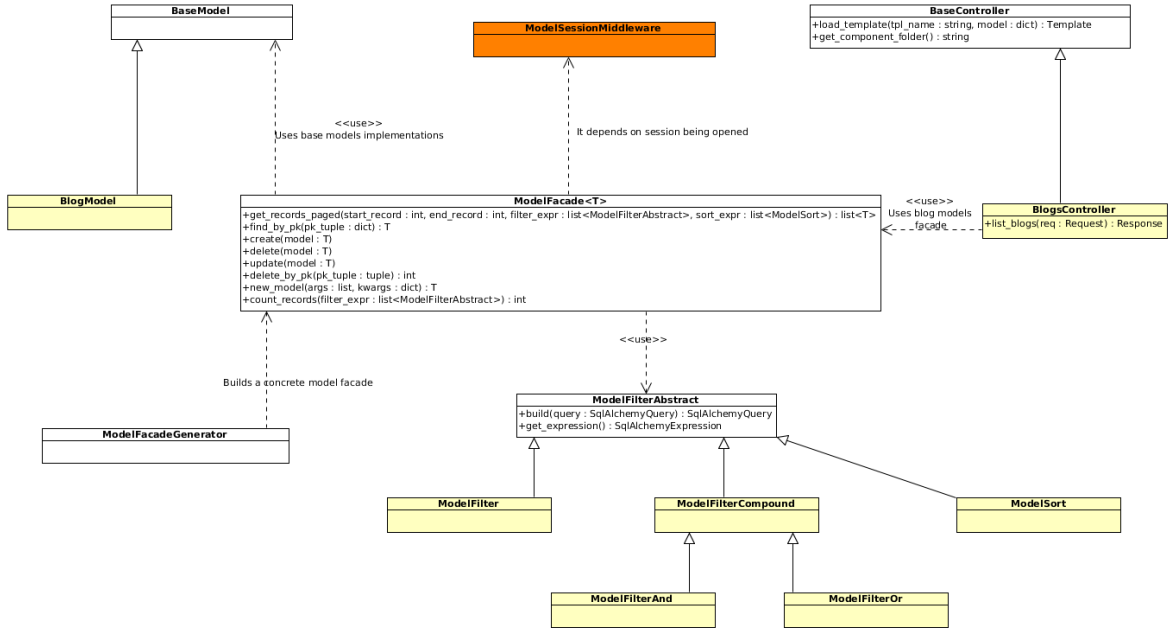
This class provides a decorator for magically registering methods as route handlers. This is an extremely important piece of Fantastico framework because it simplifies the way you as developer can define mapping between a method that must be executed when an http request to an url is made:

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

        blogs = Blog.get_records_paged(start_record=0, end_record=5,
                                sort_expr=[ModelSort(Blog.model_cls.create_date, ModelSort.ASC,
                                           ModelSort(Blog.model_cls.title, ModelSort.DESC)],
                                filter_expr=ModelFilterAnd(
                                               ModelFilter(Blog.model_cls.id, 1, ModelFilter.GT)
                                               ModelFilter(Blog.model_cls.id, 5, ModelFilter.LT)


        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`.

In order for models to work correctly and to be injected correctly into controller you must make sure you have a valid database configuration in your settings file. By default, `fantastico.settings.BasicSettings` provides a usable database configuration.

```
# fantastico.settings.BasicSettings
@property
def database_config(self):
    return {"drivername": "mysql+mysqldb",
            "username": "fantastico",
            "password": "12345",
            "host": "localhost",
            "port": 3306,
            "database": "fantastico",
            "show_sql": True}
```

By default, each time a new build is generated for fantastico each environment is validated to ensure connectivity to configured database works.

There are multiple ways in how a model is used but the easiest way is to use an autogenerated model facade:

class fantastico.mvc.model_facade.**ModelFacade**(*model_cls*, *session*)

This class provides a generic model facade factory. In order to work **Fantastico** base model it is recommended to use autogenerated facade objects. A facade object is binded to a given model and given database session.

count_records(*filter_expr=None*)

This method is used for counting the number of records from underlining facade. In addition it applies the filter expressions specified (if any).

```
records = facade.count_records(
                            filter_expr=ModelFilterAnd(
                                            ModelFilter(Blog.id, 1, ModelFilter.GT),
                                            ModelFilter(Blog.id, 5, ModelFilter.LT)))
```

Parameters **filter_expr** (*list*) – A list of fantastico.mvc.models.model_filter.ModelFilterAbstra which are applied in order.

Raises fantastico.exceptions.FantasticoDbError This exception is raised whenever an exception occurs in retrieving desired dataset. The underlining session used is automatically rollbacked in order to guarantee data integrity.

create(*model*)

This method add the given model in the database.

```
class PersonModel(BASEMODEL):
    __tablename__ = "persons"

    id = Column("id", Integer, autoincrement=True, primary_key=True)
    first_name = Column("first_name", String(50))
    last_name = Column("last_name", String(50))

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

facade = ModelFacade(PersonModel, fantastico.mvc.SESSION)

model = facade.new_model("John", last_name="Doe")
facade.create(model)
```

Returns The newly generated primary key or the specified primary key (it might be a scalar value or a tuple).

Raises fantastico.exceptions.FantasticoDbError Raised when an unhandled exception occurs. By default, session is rollback automatically so that other consumers can still work as expected.

delete(*model*)

This method deletes a given model from database. Below you can find a simple example of how to use this:

```
class PersonModel(BASEMODEL):
    __tablename__ = "persons"

    id = Column("id", Integer, autoincrement=True, primary_key=True)
```

```
        first_name = Column("first_name", String(50))
        last_name = Column("last_name", String(50))

        def __init__(self, first_name, last_name):
            self.first_name = first_name
            self.last_name = last_name

    facade = ModelFacade(PersonModel, fantastico.mvc.SESSION)
    model = facade.find_by_pk({PersonModel.id: 1})
    facade.delete(model)
```

Raises fantastico.exceptions.FantasticoDbError Raised when an unhandled exception occurs. By default, session is rollback automatically so that other consumers can still work as expected.

find_by_pk(*pk_values*)

This method returns the entity which matches the given primary key values.

```
    class PersonModel(BASEMODEL):
        __tablename__ = "persons"

        id = Column("id", Integer, autoincrement=True, primary_key=True)
        first_name = Column("first_name", String(50))
        last_name = Column("last_name", String(50))

        def __init__(self, first_name, last_name):
            self.first_name = first_name
            self.last_name = last_name

    facade = ModelFacade(PersonModel, fantastico.mvc.SESSION)
    model = facade.find_by_pk({PersonModel.id: 1})
```

get_records_paged(*start_record*, *end_record*, *filter_expr=None*, *sort_expr=None*)

This method retrieves all records matching the given filters sorted by the given expression.

```
    records = facade.get_records_paged(start_record=0, end_record=5,
                               sort_expr=[ModelSort(Blog.create_date, ModelSort.ASC,
                                          ModelSort(Blog.title, ModelSort.DESC)],
                               filter_expr=ModelFilterAnd(
                                           ModelFilter(Blog.id, 1, ModelFilter.GT),
                                           ModelFilter(Blog.id, 5, ModelFilter.LT))))
```

Parameters

- **start_record** (*int*) – A zero indexed integer that specifies the first record number.
- **end_record** (*int*) – A zero indexed integer that specifies the last record number.
- **filter_expr** (*list*) – A list of `fantastico.mvc.models.model_filter.ModelFilterAbstract` which are applied in order.
- **sort_expr** (*list*) – A list of `fantastico.mvc.models.model_sort.ModelSort` which are applied in order.

Returns A list of matching records strongly converted to underlining model.

Raises fantastico.exceptions.FantasticoDbError This exception is raised whenever an exception occurs in retrieving desired dataset. The underlining session used is automatically rollbacked in order to guarantee data integrity.

model_cls
This property holds the model based on which this facade is built.

new_model(**args*, ***kwargs*)
This method is used to obtain an instance of the underlining model. Below you can find a very simple example:

```
class PersonModel(BASEMODEL):
    __tablename__ = "persons"

    id = Column("id", Integer, autoincrement=True, primary_key=True)
    first_name = Column("first_name", String(50))
    last_name = Column("last_name", String(50))

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

facade = ModelFacade(PersonModel, fantastico.mvc.SESSION)

model = facade.new_model("John", last_name="Doe")
```

Parameters

- **args** (*list*) – A list of positional arguments we want to pass to underlining model constructor.
- **kwargs** (*dict*) – A dictionary containing named parameters we want to pass to underlining model constructor.

Returns A BASEMODEL instance if everything is ok.

update(*model*)
This method updates an existing model from the database based on primary key.

```
class PersonModel(BASEMODEL):
    __tablename__ = "persons"

    id = Column("id", Integer, autoincrement=True, primary_key=True)
    first_name = Column("first_name", String(50))
    last_name = Column("last_name", String(50))

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

facade = ModelFacade(PersonModel, fantastico.mvc.SESSION)

model = facade.new_model("John", last_name="Doe")
model.id = 5
facade.update(model)
```

Raises

- **fantastico.exceptions.FantasticoDbNotFoundError** – Raised when the given model does not exist in database. By default, session is rollback automatically so that other consumers can still work as expected.

- **fantastico.exceptions.FantasticoDbError** – Raised when an unhandled exception occurs. By default, session is rollback automatically so that other consumers can still work as expected.

If you are using the **Fantastico MVC** support you don't need to manually create a model facade instance because `fantastico.mvc.controller_decorators.Controller` injects defined models automatically.

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`.

class `fantastico.mvc.controller_registrator.`**ControllerRouteLoader**(*settings_facade=<class 'fantastico.settings.SettingsFacade'>*, *scanned_folder=None*, *ignore_prefix=None*)

This class provides a route loader that is capable of scanning the disk and registering only the routes that contain a controller decorator in them. This happens when **Fantastico** servers starts. In standard configuration it ignores tests subfolder as well as test_* / itest_* modules.

load_routes()

This method is used for loading all routes that are mapped through `fantastico.mvc.controller_decorators.Controller` decorator.

class `fantastico.mvc.base_controller.`**BaseController**(*settings_facade*)

This class provides common methods useful for every concrete controller. Even if no type checking is done in Fantastico it is recommended that every controller implementation inherits this class.

get_component_folder()

This method is used to retrieve the component folder name under which this controller is defined.

load_template(*tpl_name*, *model_data=None*, *get_template=<function get_template at 0x242f408>*)

This method is responsible for loading a template from disk and render it using the given model data.

```
@ControllerProvider()
class TestController(BaseController):
    @Controller(url="/simple/test/hello", method="GET")
    def say_hello(self, request):
        return Response(self.load_template("/hello.html"))
```

The above snippet will search for **hello.html** into component folder/views/.

Available filters

class fantastico.mvc.models.model_filter_compound.**ModelFilterCompound**(*operation*, **args*)

This class provides the api for compounding ModelFilter objects into a specified sql alchemy operation.

build(*query*)

This method transform the current compound statement into an sql alchemy filter.

get_expression()

This method transforms calculates sqlalchemy expression held by this filter.

class fantastico.mvc.models.model_filter.**ModelFilter**(*column*, *ref_value*, *operation*)

This class provides a model filter wrapper used to dynamically transform an operation to sql alchemy filter statements. You can see below how to use it:

```
id_gt_filter = ModelFilter(PersonModel.id, 1, ModelFilter.GT)
```

build(*query*)

This method appends the current filter to a query object.

column

This property holds the column used in the current filter.

get_expression()

Method used to return the underlining sqlalchemy exception held by this filter.

static get_supported_operations()

This method returns all supported operations for model filter. For now only the following operations are supported:

- •GT - greater than comparison
- •GE - greater or equals than comparison
- •EQ - equals comparison
- •LE - less or equals than comparison
- •LT - less than comparison
- •LIKE - like comparison
- •IN - in comparison.

operation

This property holds the operation used in the current filter.

ref_value

This property holds the reference value used in the current filter.

class fantastico.mvc.models.model_filter_compound.**ModelFilterAnd**(**args*)
This class provides a compound filter that allows **and** conditions against models. Below you can find a simple example:

```
id_gt_filter = ModelFilter(PersonModel.id, 1, ModelFilter.GT)
id_lt_filter = ModelFilter(PersonModel.id, 5, ModelFilter.LT)
name_like_filter = ModelFilter(PersonModel.name, '%%john%%', ModelFilter.LIKE)

complex_condition = ModelFilterAnd(id_gt_filter, id_lt_filter, name_like_filter)
```

class fantastico.mvc.models.model_filter_compound.**ModelFilterOr**(**args*)
This class provides a compound filter that allows **or** conditions against models. Below you can find a simple example:

```
id_gt_filter = ModelFilter(PersonModel.id, 1, ModelFilter.GT)
id_lt_filter = ModelFilter(PersonModel.id, 5, ModelFilter.LT)
name_like_filter = ModelFilter(PersonModel.name, '%%john%%', ModelFilter.LIKE)

complex_condition = ModelFilterOr(id_gt_filter, id_lt_filter, name_like_filter)
```

class fantastico.mvc.models.model_sort.**ModelSort**(*column*, *sort_dir=None*)
This class provides a filter that knows how to sort rows from a query result set. It is extremely easy to use:

```
id_sort_asc = ModelSort(PersonModel.id, ModelSort.ASC)
```

build(*query*)
This method appends sort_by clause to the given query.

column
This property holds the column we are currently sorting.

get_expression()
This method returns the sqlalchemy expression held by this filter.

get_supported_sort_dirs()
This method returns all supported sort directions. Currently only ASC / DESC directions are supported.

sort_dir
This property holds the sort direction we are currently using.

3.4.6 Database session management

We all know database session management is painful and adds a lot of boiler plate code. In fantastico you don't need to manage database session by yourself. There is a dedicated middleware which automatically ensures there is an active session ready to be used:

class fantastico.middleware.model_session_middleware.**ModelSessionMiddleware**(*app*, *settings_facade=<class 'fantastico.settings.SettingsFaca*
This class is responsible for managing database connections across requests. It also takes care of connection data pools. By default, the middleware is automatically configured to open a connection. If you don't need mvc (really improbable but still) you simply need to change your project active settings profile. You can read more on fantastico.settings.BasicSettings

3.5 Component model

In Fantastico there is no enforced component model for your code but there are a set of recommendations that will make your life a lot easier when organizing projects. A typical **component** structure looks like:

- **<your project folder>**
  - **component_1**
    - models (sql alchemy models)
    - static (static files holder)
    - views (all views used by this component controllers')
    - sql (sql scripts required to setup the component)
    - __init__.py
    - *.py (controller module files)

You can usually structure your code as you want, but Fantastico default *Model View Controller* registrators are assuming component name is the parent folder of the controller module. This is why is best to follow the above mentioned structure. None of the above folders are mandatory which gives you, developer, plenty of flexibility but also responsibility. For more information about **models**, **views** and **controllers** read *MVC How to* section.

3.5.1 Static folder

By default, static folder holds all static assets belonging to a component. You can find more information about this in *Static assets* section.

3.5.2 Sql folder

Sql folder is used to hold all sql scripts required for a component to work correctly. In our continuous delivery process we scan all available sql folders and execute **module_setup.sql** scripts. By default, we want to give developers the chance to provide a setup script for each component in order to easily install the component database dependencies.

Sql folder example

Assume you want to create a blog module that requires a storage for **Authors** and **Posts**. module_setup.sql script is the perfect place to provide the code. We recommend to make this code idempotent, meaning that once dependencies are created they should not be altered anymore by this script.

An example of such a script we use in integration tests can be found under: **/<fantastico_framework>/samples/mvc/sql/module_setup.sql**.

```
###############################################################################################
# Copyright 2013 Cosnita Radu Viorel
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software
# and associated documentation files (the "Software"), to deal in the Software without
# restriction, including without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom
# the Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
```

```
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
# INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
# PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
# ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
# ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
# IN THE SOFTWARE.
##############################################################################################


DROP TABLE IF EXISTS mvc_friendly_messages;
CREATE TABLE mvc_friendly_messages(
   Id INT AUTO_INCREMENT,
   Message TEXT,
   PRIMARY KEY(id));
```

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:

```
@ControllerProvider()
class BlogsController(BaseController):
    @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=[ModelSort(Post.model_cls.created_date, ModelSort.A
                                               ModelSort(Post.title, ModelSort.DESC)],
                                    where_expr=[ModelFilter(Post.model_cls.blog_id, blog_id, Mode

        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>
```

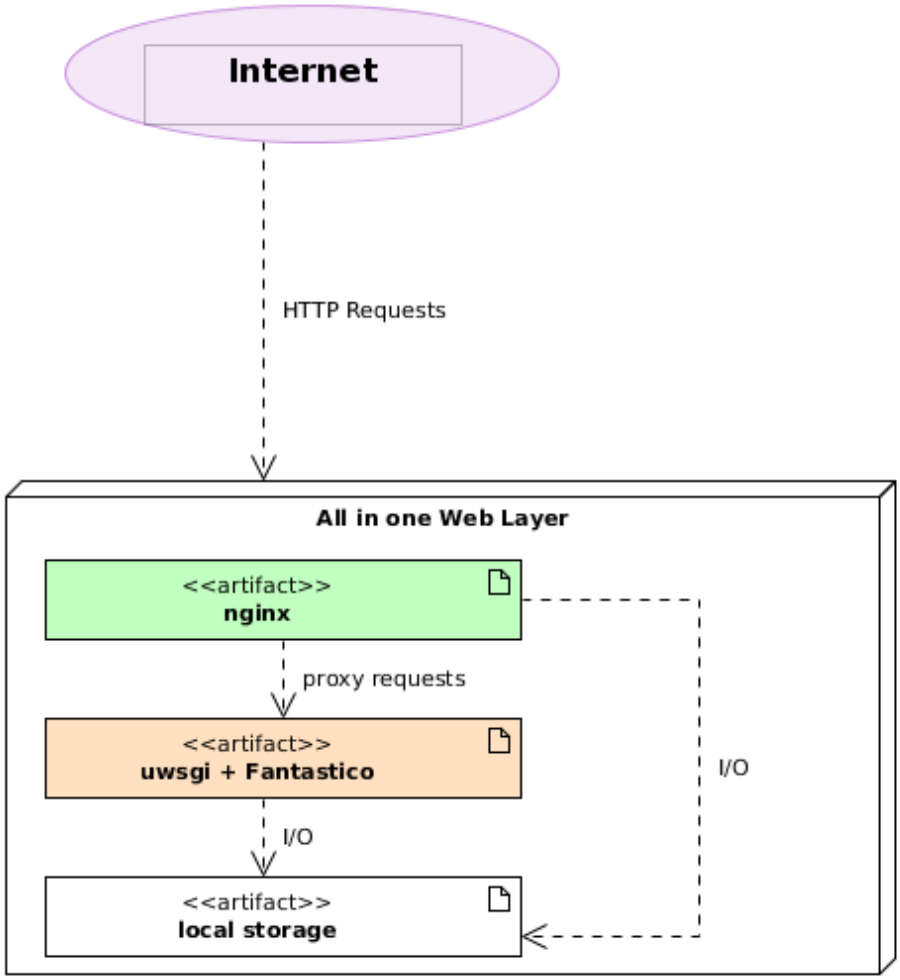
4.1.4 Test your application

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.

4.2 Fantastico deployment how to

In this how to we guide you to Fantastico deployment to production. Below you can find various deployment scenarios that can be used for various needs.

4.2.1 Low usage (simplest scenario)



Above diagram described the simplest scenario for rolling out Fantastico to production. You can use this scenario for minimalistic web applications like:

- Presentation website
- Personal website
- Blog

We usually recommend to start with this deployment scenario and the migrate to more complex scenarios when you application requires it.

Advantages	Disadvantages
Extremely easy to deploy	Does not scale well for more than couple of requests / second
Minimal os configuration	All components are bundled on one node without any failover.
Automatic scripts for configuring the os	Does not support vertical scaling out of the box.
Easy to achieve horizontal scaling for all components at once.	Static files are not served from a cdn.

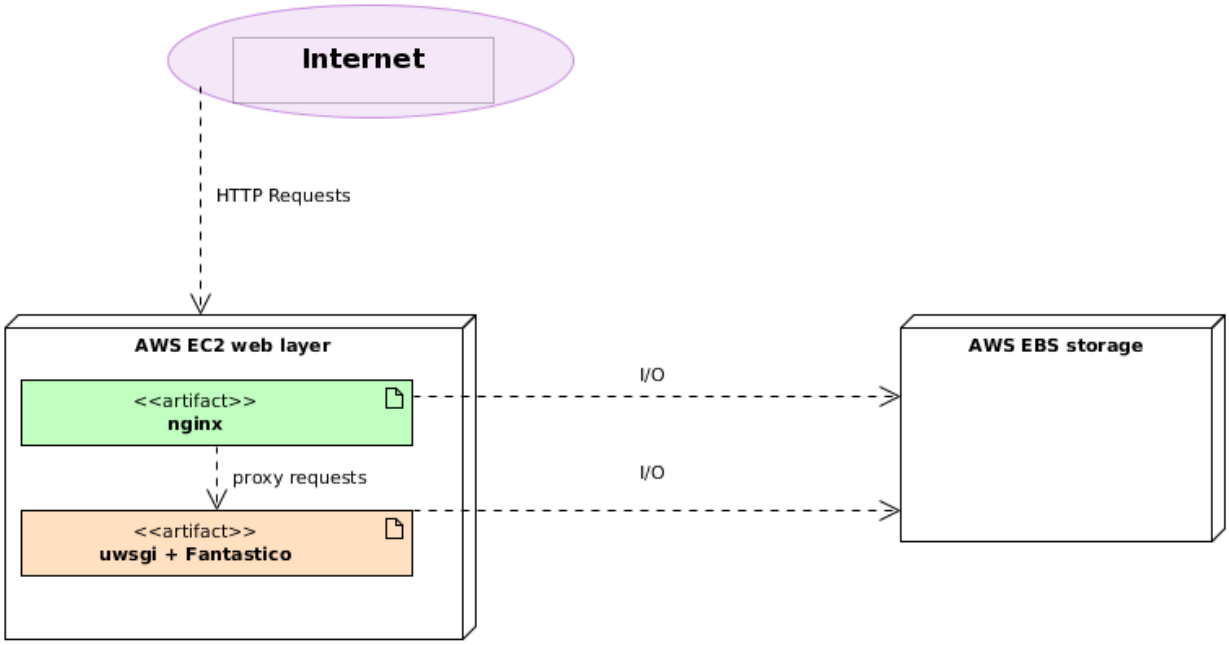
Setup

1. Install Fantastico framework on the production machine (*Installation manual*.).
2. Goto $FANTASTICO_ROOT/deployment
3. export ROOT_PASSWD=<your root password>
4. sh setup_low_usage_<os_distribution).sh –ipaddress <desired_ip> –vhost-name <desired_vhost> –uwsgi-port <uwsgi port> –root-folder <desired root folder> –modules-folder <desired modules folder> (e.g sh setup_low_usage_ubuntu.sh –ipaddress 127.0.0.1 –vhost-name fantastico-framework.com –uwsgi-port 12090 –root-folder ‘pwd‘ –modules-folder /fantastico/samples)
5. Done.

It is usually a good idea to change the number of parallel connections supported by your linux kernel:

1. sudo nano /etc/sysctl.conf
2. Search for **net.core.somaxconn**.
3. If it does not exist you can add net.core.somaxconn = 8192 to the bottom of the file.
4. Restart the os.

4.2.2 Low usage AWS



This scenario is a little bit more complex than *Low usage (simplest scenario)* but it provides some advantages:

Advantages	Disadvantages
Can be autoscaled.	Requires AWS EC2 instances
Easier crash recovery	Requires manual configuration
Very easy monitoring support (CloudWatch)	Requires AWS EBS.
	Requires some AWS know how.
	Static files are not served from a cdn.

This scenario is recommended if you want to rollout you application on AWS infrastructure. Usually it is non expensive to do this as it requires micro instances and low cost storage. For more information about AWS required components read:

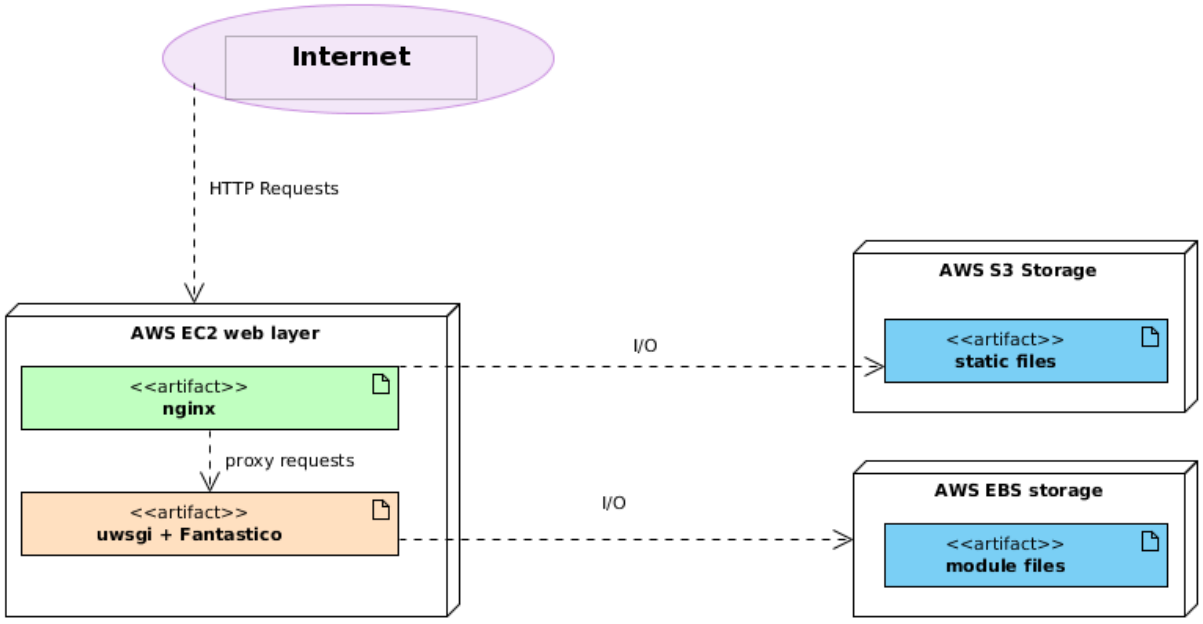
1. AWS Instance types.
2. AWS EBS.

Setup

1. Create an AWS account. (AWS Getting Started).
2. Create an EC2 instance from AWS Management Console (EC2 setup).
3. SSH on EC2 instance.
4. Install Fantastico framework on the production machine (*Installation manual*.).
5. Goto $FANTASTICO_ROOT/deployment
6. sh setup_low_usage_<os_distribution).sh (e.g sh setup_low_usage_ubuntu.sh)
7. Done.

Optimization

This scenario can be easily optimized by using **AWS S3** buckets for static files. This ensures faileover for static files and very easy horizontal scaling for sites. Below you can find the new diagram:



You can read more about **AWS S3** storage on http://aws.amazon.com/s3/. In this version of fantastico there is no way to sync static module files with S3 buckets. This feature is going to be implemented in upcoming **Fantastico** features. As a workaround you can easily copy **static** folder content from each module on S3 using the tool provided from AWS Management Console.

You can see how to use AWS Management Console S3 tool on http://www.youtube.com/watch?v=1qrjFb0ZTm8

Setup with S3

1. export ROOT_PASSWD=<your root password>
2. Create an AWS account. (AWS Getting Started).
3. Create an EC2 instance from AWS Management Console (EC2 setup).
4. SSH on EC2 instance.
5. Install Fantastico framework on the production machine (*Installation manual*.).
6. Goto $FANTASTICO_ROOT/deployment
7. sh setup_low_usage_s3_<os_distribution).sh –ipaddress <desired_ip> –vhost-name <desired_vhost> –uwsgi-port <uwsgi port> –root-folder <desired root folder> –modules-folder <desired modules folder> (e.g sh setup_low_usage_s3_ubuntu.sh –ipaddress 127.0.0.1 –vhost-name fantastico-framework.com –uwsgi-port 12090 –root-folder ‘pwd‘ –modules-folder /fantastico/samples)
8. Done.

It is usually a good idea to change the number of parallel connections supported by your linux kernel:

1. sudo nano /etc/sysctl.conf
2. Search for **net.core.somaxconn**.
3. If it does not exist you can add net.core.somaxconn = 8192 to the bottom of the file.
4. Restart the os.

4.3 Static assets

By default, static assets can be any file that is publicly available. Most of the time, here you can place:

- css files
- png, jpg, gif files
- downloadable pdf
- movie files
- any other file format you can think about

For Production environment, requests to these files are handled by the web server you are using. You only need to place them under **static** folder of your component (*Component model*).

There are several scenario in which Fantastico projects are deployed which influence where your component static files are stored. I recommend you read *Fantastico deployment how to* section.

4.3.1 Static assets on dev

Of course, on development environment you are not required to have a web server in front of your Fantastico dev server. For this purpose, fantastico framework provides a special controller which can easily serve static files. Even this works as expected, please do not use it in production. It does not send headers required by browser for caching purposes.

Static assets routes are the same between **prod** and **dev** environments.

CHAPTER

FIVE

BUILD STATUS

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

CHAPTER

SIX

LICENSE

Symbols

B

C

D

F

G

H

I

L

M

N

O

R

S

T

U