fantastico Documentation

Release 0.5.0-b121

Radu Viorel Cosnita

September 25, 2013

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.

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:

```python
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.
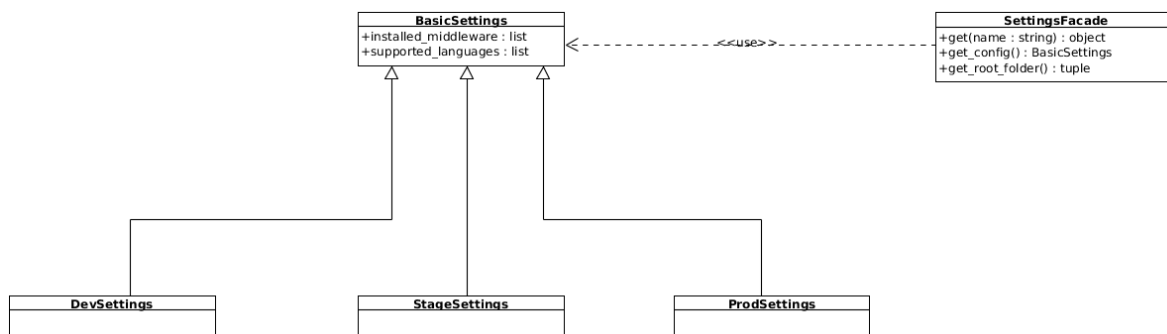
> As you can see, if you want to overwrite basic configuration you simply have to extend the class and set new values for the attributes you want to overwrite.
>
> **database_config**
>
> > This property holds the configuration of database. It is recommended to have all environment configured the same. An exception can be done for host but the rest must remain the same. Below you can find an example of functional configuration:
> >
> > ```python
> > 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.

doc_base
: This property defines public location of **Fantastico** documentation.

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:

```python
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:

    ```python
    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.

`check_original_methods`(*cls_obj*)
This method ensures that for a given class only original non decorated methods will be invoked. Extremely useful when you want to make sure @Controller decorator does not break your tests. It is strongly recommended to invoke this method on all classes which might contain @Controller decorator. It ease your when committing on CI environment.

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.

fantastico_cfg_os_key

This property holds the name of os environment variable used for setting up active fantastico configuration.

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.

```python
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

        # you can also pass only request logic without assert logic
        # self._run_test_all_envs(lambda env, settings_cls: self._run_test_against_dev_server(re
```

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 0x4ab69e0>*, *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.

FANTASTICO FEATURES

3.1 Exceptions hierarchy

class `fantastico.exceptions.`**`FantasticoError`**
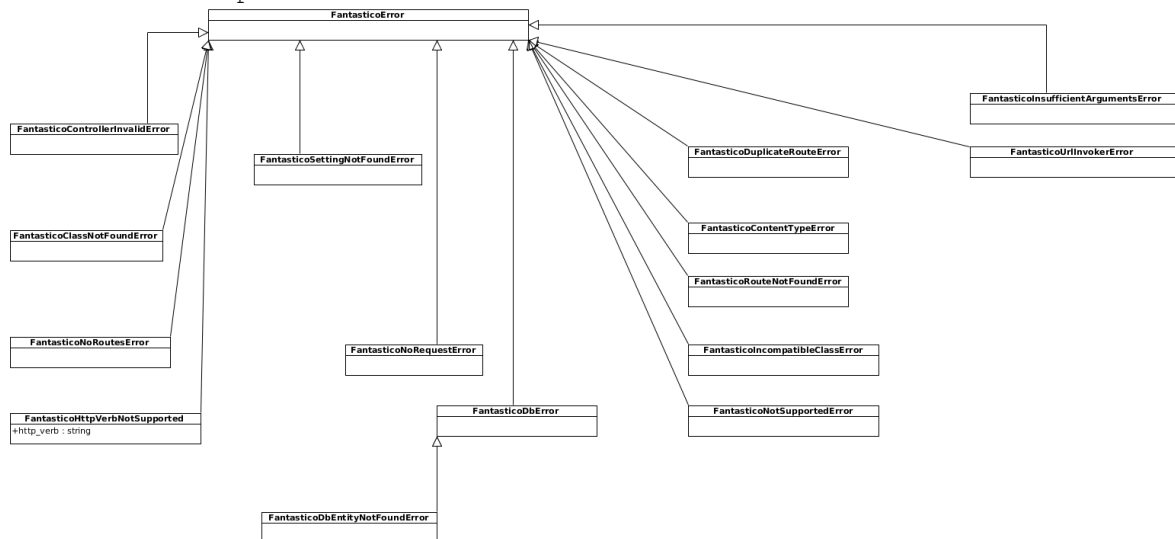
FantasticoError is the base of all exceptions raised within fantastico framework. It describe common attributes that each concrete fantastico exception must provide. By default all fantastico exceptions inherit FantasticoError exception. We do this because each raised unhandled FantasticoError is map 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`.

class `fantastico.exceptions.`**`FantasticoInsufficientArgumentsError`**
> This exception is usually thrown when a component extension received wrong number of arguments. See `fantastico.rendering.component.Component`.

class `fantastico.exceptions.`**`FantasticoUrlInvokerError`**
> This exception is usually thrown when an internal url invoker fails. For instance, if a component reusage rendering fails then this exception is raised.

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**. In addition each new request receives an identifier. If subsequent requests are triggered from that request then they will also receive the same request id.

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.

wsgi_app
> Property that holds the WSGI application instance under which the request is handled.

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*.

3.2.5 Redirect using request

In Fantastico is fairly simply to redirect client to a given location.

class `fantastico.routing_engine.custom_responses.`**`RedirectResponse`**(*destination*, *query_params=None*)
> This class encapsulates the logic for programatically redirecting client from a fantastico controller.

```
@Controller(url="/redirect/example")
def redirect_to_google(self, request):
    return request.redirect("http://www.google.ro/")
```

There are some special cases when you would like to pass some query parameters to redirect destination. This is also easily achievable in Fantastico:

```
@Controller(url="/redirect/example")
def redirect_to_google(self, request):
    return request.redirect("http://www.google.ro/search",
                            query_params=[("q", "hello world")])
```

The above example will redirect client browser to http://www.google.ro/search?q=hello world

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* *'fantas-tico.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 foot-print. 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.

```python
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:

```python
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*
> > > > > > > > > > > *'fantas-*
> > > > > > > > > > > *tico.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*, ***kwargs*)

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)

        # convert blogs to desired format. E.g: json.

        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:

BaseModel

ModelSessionMiddleware

BaseController
+load_template(tpl_name : string, model : dict) : Template
+get_component_folder() : string

<<use>>
Uses base models implementations

It depends on session being opened

BlogModel

ModelFacade<T>
+get_records_paged(start_record : int, end_record : int, filter_expr : list<ModelFilterAbstract>, sort_expr : list<ModelSort>) : list<T>
+find_by_pk(pk_tuple : dict) : T
+create(model : T)
+delete(model : T)
+update(model : T)
+delete_by_pk(pk_tuple : tuple) : int
+new_model(args : list, kwargs : dict) : T
+count_records(filter_expr : list<ModelFilterAbstract>) : int

<<use>>
Uses blog models
facade

BlogsController
+list_blogs(req : Request) : Response

Builds a concrete model facade

ModelFacadeGenerator

<<use>>

ModelFilterAbstract
+build(query : SqlAlchemyQuery) : SqlAlchemyQuery
+get_expression() : SqlAlchemyExpression

ModelFilter

ModelFilterCompound

ModelSort

ModelFilterAnd

ModelFilterOr

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.

```python
# 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.ModelFilterAbstrac` 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.

> > ```python
> > 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:

> > ```python
> > class PersonModel(BASEMODEL):
> >     __tablename__ = "persons"
> >
> >   id = Column("id", Integer, autoincrement=True, primary_key=True)
> > ```

```python
    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.

```python
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.

```python
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.

3.4. Model View Controller 21

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:

```python
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.

```python
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.

> **`scanned_folder`**
>> This property returns the currently scanned folder from where mvc routes are collected.

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.

> **`curr_request`**
>> This property returns the current http request being processed.

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 0x1531408>*)
>   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.**ModelFilterAbstract**
>   This is the base class that defines the contract a model filter must follow. A model filter is a class that decouples sqlalchemy framework from Fantastico MVC. This is required because in the future we might want to change the ORM that powers Fantastico without breaking all existing code.

>   For seeing how to implement filters (probably you won't need to do this) see some existing filters:
>
>   - fantastico.mvc.models.model_filter.ModelFilter
>
>   - fantastico.mvc.models.model_filter_compound.ModelFilterCompound
>
>   - fantastico.mvc.models.model_filter_compound.ModelFilterAnd
>
>   - fantastico.mvc.models.model_filter_compound.ModelFilterOr

build(*query*)
>   This method is used for appending the current filter to the query using sqlalchemy specific language.

get_expression()
>   This method is used for retrieving native sqlalchemy expression held by this filter.

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.SettingsFacade*

> 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 ROA (Resource Oriented Architecture)

Resource Oriented Architecture (REST) is incredible popular nowadays for the following reasons:

- Increased scalability of applications.
- Easy integration of systems.
- Intuitive modelling of business problems.
- Stateful imperative programming pains removed.

You can find many information about advantages of REST and why it is recommended to use such an architecure. For further reading you can visit http://en.wikipedia.org/wiki/Representational_state_transfer.

In Fantastisco framework we firmly encourage REST approach into projects. We even go a step further in this direction, by standardising REST APIs and providing REST APIs generator over implemented models.

3.5.1 Examples

3.5.2 Application settings stored in database

Imagine you have a model called **AppSetting** meant to define custom settings attributes which influence your application.

```python
class AppSetting(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(50), unique=True, nullable=False)
    value = Column("value", Text, nullable=False)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

In a standard **MVC** (*Model View Controller*) web application once you have defined the above mentioned fantastico model you would have to do the following: (for providing minimal CRUD)

1. Create a `fantastico.mvc.controller_decorators.Controller`

2. Implement listing of custom settings

    (a) Support pagination.

    (b) Support filtering.

    (c) Support ordering.

3. Implement individual custom setting retrieval (by id).

4. Implement Create custom setting.

5. Implement Update custom setting.

6. Implement Delete custom setting.

7. For each operation implemented provide validation logic.

8. For each operation implemented provide error handling logic.

This is an extremely repetitive task and involves quite a lot of boiler plate. In addition no standard is imposed for how pagination, sorting and filtering work.

A more convenient way for this problem is to provide some additional information about the model:

```python
@Resource(name="app-setting", url="/app-settings")
class AppSetting(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(50), unique=True, nullable=False)
    value = Column("value", Text, nullable=False)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

Once the model is decorated, I expect to have a fully functional API which I can easily invoke through HTTP calls:

- GET - /api/latest/app-settings - list all application settings (supports filtering, ordering and pagination)

- POST - /api/latest/app-settings - create a new app setting.

- PUT - /api/latest/app-settings/:id - update an existing application setting.

- DELETE - /api/latest/app-settings/:id - delete an existing application setting.

3.5.3 Versioning

It is always a good practice to support API versioning. Going a step further with AppSetting resource:

```python
@Resource(name="app-setting", url="/app-settings", version=1.0)
class AppSetting(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(50), unique=True, nullable=False)
    value = Column("value", Text, nullable=False)

    def __init__(self, name, value):
        self.name = name
        self.value = value

@Resource(name="app-setting", url="/app-settings", version=2.0)
class AppSettingV2(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(80), unique=True, nullable=False)
    value = Column("value", Text, nullable=False)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

The above example will actually provide the following endpoints which can be easily accessible:

- /api/1.0/app-settings
- /api/2.0/app-settings
- /api/latest/app-settings (which at this moment points to the most recent version of the api)

If we want to retrieve all application settings using version 1.0 we open a browser and point it to **/api/1.0/app-settings**. For avoiding multiple APIs chaos we strongly encourage to use the latest available API.

3.5.4 Validation

Each resource requires validation for create / update operations. Validation is harder to be achieved through code introspection so in Fantastico for each defined resource you can define a validator which will be invoked automatically.

```python
class AppSettingValidator(ResourceValidator):
    def validate(self, resource):
        errors = []

        if resource.name == "unsupported":
            errors.append("Invalid setting name: %s" % resource.name)

        if len(resource.value) == 0:
            errors.append("Setting %s value can not be empty. %s" % resource.name)

        if len(errors) == 0:
            return

        raise FantasticoResourceError(errors)

@Resource(name="app-setting", url="/app-settings", version=2.0, validator=AppSettingValidator)
class AppSettingV2(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(80), unique=True, nullable=False)
```

```
   value = Column("value", Text, nullable=False)

   def __init__(self, name, value):
      self.name = name
      self.value = value
```

If no validator is provided no validation is done on the given resource. Also, it is important to always remember that validators are only invoked for **Create** and **Update**:

- POST - /api/latest/app-settings - create a new app setting. (validate method will be invoked).

- PUT - /api/latest/app-settings/:id - update an existing app setting. (validate method will be invoked).

We are aware that there are some common validation cases which can be reused:

1. Email validation

2. Phone number validation

3. Credit Card number validation

All common validation cases are provided out of the box as methods part of ResourceValidator class. You can easily use them into your resource validator.

3.5.5 Partial object representation

There are cases when a resource contains many fields but you actually need only a few of them:

```
@Resource(name="address", url="/addresses", version=1.0)
class Address(BASEMODEL):
   id = Column("id", Integer, primary_key=True, autoincrement=True)
   line1 = Column("line1", String(200), nullable=False)
   line2 = Column("line2", String(200))
   line3 = Column("line3", String(200))
   line4 = Column("line4", String(200))
   line5 = Column("line5", String(200))
   line6 = Column("line6", String(200))
   city = Column("city", String(80))
   country = Column("country", String(80))
   zip_code = Column("zip_code", String(10))

   def __init__(self, line1=None, line2=None, line3=None, line4=None, line5=None, line6=None,
               city=None, country=None, zip_code=None):
      self.line1 = line1
      self.line2 = line2
      self.line3 = line3
      self.line4 = line4
      self.line5 = line5
      self.line6 = line6
      self.city = city
      self.country = country
      self.zip_code = zip_code
```

When working with the Address resource there will be cases when we do not need all fields to be transferred to client. For this, partial representation is supported out of the box into Fantastico:

```
// retrieve only city,zip_code and line1 of a given address
var url = "/api/1.0/addresses/1?fields=city,zip_code,line1";
```

A possible response for this request might be:

```
{
    city: "Bucharest",
    zip_code: "B00001",
    line1: "First line of this wonderful address"
}
```

fields Query parameter is optional. If you omit this query parameter all fields are retrieved in the response. **fields** query parameter makes sense for:

1. Listing a collection

2. Retrieving information about an individual item.

All other operations simply ignore **fields**.

3.5.6 Resource composed attributes

There are resources which have attributes which points to another resource:

```
@Resource(name="person", url="/persons", version=1.0,
          subresources={"bill_address": ["bill_address_id"],
                        "mail_address": ["mail_address_id"],
                        "ship_address:" ["ship_address_id"])
class Person(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    first_name = Column("first_name", String(80))
    last_name = Column("last_name", String(50))
    bill_address_id = Column("bill_address_id", ForeignKey("addresses.id"))
    bill_address = relationship(Address, primaryjoin=bill_address_id == Address.id)
    ship_address_id = Column("ship_address_id", ForeignKey("addresses.id"))
    ship_address = relationship(Address, primaryjoin=ship_address_id == Address.id)
    mail_address_id = Column("ship_address_id", ForeignKey("addresses.id"))
    ship_address = relationship(Address, primaryjoin=mail_address_id == Address.id)
```

The above definition shows you how to mark the subresources of **person** resource. By default they will not be retrieved in requests. Only subresource identifier keys pointing from **person** to various **address** objects are retrieved. If you want to obtain details about a specific address (e.g bill_address) you can use example below:

```
var url = "/api/1.0/persons/1?fields=first_name, last_name, bill_address(line1, city, zip_code)"
```

The above example url might return:

```
{
    "first_name": "John",
    "last_name": "Doe",
    "bill_address": {
        "line1": "First line of this wonderful address",
        "city": "Bucharest",
        "zip_code": "B00001"
    }
}
```

Composed attributes usage is limited to below mentioned operations:

• Listing collections.

• Retrieving information about an individual item.

We do not support update / create of multiple resources using one single request.

3.5.7 Advantages

- Extremely fast development of uniform APIs which behave predictable.

- Extremely easy to enforce exception handling logic.

- Extremely easy to enforce security for APIs.

- Extremely easy to keep APIs in sync with resource changes.

- DRY (don't repeat yourself).

REST API standard

In this document you can find the standard behavior imposed for Resource generated apis (*ROA (Resource Oriented Architecture)*). For better understanding how APIs will behave let's assume we have the following resource defined:

```python
@Resource(name="app-setting", url="/app-settings", version=2.0)
class AppSettingV2(BASEMODEL):
    id = Column("id", Integer, primary_key=True, autoincrement=True)
    name = Column("name", String(80), unique=True, nullable=False)
    value = Column("value", Text, nullable=False)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

Resource collection

Each resource will have an API entry point which lists all available resources of same type (e.g **AppSettingV2**). This entry point supports the following additional operations:

- Pagination of resources.

- Sorting of resources.

- Filtering of resources.

The main entry point for **AppSettingV2** collection of resources is **/api/2.0/app-settings**.

HTTP Verb	URL	Description
GET	/api/2.0/app-settings?offset=0&limit=100	Get the first 100 settings.
GET	/api/2.0/app-settings?order=[desc(name), asc(value)]	Order settings by name (descending) and value (ascending).
GET	/api/2.0/app-settings?filter=<complex filter>	See *Filtering*.
POST	/api/2.0/app-settings	Create a new custom setting

Pagination When requesting a given resource collection a subset of this collection will be retrived.

- **offset** - defines which is the start record of the API. (Default value is 0)

- **limit** - defines the maximum number of items I want to retrieve. (Default value is 10)

A possible result for **AppSettingV2** collection retrieval looks like:

```
{
    "items":
        [{"id": 1, "name": "default_locale", "value": "en_US"},
         {"id": 2, "name": "vat", "value": 0.19}],
    "totalItems": 1000
}
```

Sorting When requesting a given resource collection sorted you can specify the sorting criteria:

- **order** - a JSON list containing asc / desc function calls. This list is evaluated from left to right with left most element having the highest priority.

- **asc** - is a function with one or more arguments which tells API an ascending order by given attributes.

    ```
    // retrieve all application settings ordered in ascending order of name and value
    var url = "/api/2.0/app-settings?order=[asc(name, value)]";
    ```

- **desc** - is a function with one or more arguments which tells API a descending order by given attributes.

    ```
    // retrieve all application settings ordered in descending order of name and value.
    var url = "/api/2.0/app-settings?order=[desc(name, value)]";
    ```

- **complex ordering** - you can easily specify different ordering criterias by resource attributes.

    ```
    // retrieve all application settings ordered in descending order of name and ascending order
    var url = "/api/2.0/app-settings?order=[desc(name), asc(value)]";
    ```

A possible result for **AppSettingV2** collection retrieval (**/api/2.0/app-settings?order=[desc(name)]**) looks like:

```
{
    "items":
        [{"id": 2, "name": "vat", "value": 0.19},
         {"id": 1, "name": "default_locale", "value": "en_US"}],
    "totalItems": 1000
}
```

Filtering In fantastico, APIs filtering is done by following a very simple Resource Query Language (RQL):

HTTP Verb	URL	Description
GET	/api/2.0/app-settings?filter=eq(name, "vat")	Get all settings named **vat**.
GET	/api/2.0/app-settings?filter=like(name, "%vat%")	Get all settings which name contains **vat**.
GET	/api/2.0/app-settings?filter=gt(value, 0.19)	Get all settings which have value greater than **0.19**.
GET	/api/2.0/app-settings?filter=ge(value, 0.19)	Get all settings which have value greater / equals than / with **0.19**.
GET	/api/2.0/app-settings?filter=lt(value, 0.19)	Get all settings which have value less than **0.19**.
GET	/api/2.0/app-settings?filter=le(value, 0.19)	Get all settings which have value less / equals than / with **0.19**.
GET	/api/2.0/app-settings?filter=in(name, ["vat", "default_locale"])	Get all settings which name is **vat** or **default_locale**.
GET	/api/2.0/app-settings?filter=and(eq(name, "vat"), eq(value, "en_US"))	Get all settings which name is **vat** and value is **en_US**.
GET	/api/2.0/app-settings?&filter=or(eq(name, "vat"), eq(value, "en_US"))	Get all settings which name is **vat** or value is **en_US**.

You can see in the above example that the query language supported by Fantastico APIs facilitate very complex filtering on resources.

Resource item

A collection is composed of multiple items (same resource type). You can used individual item endpoints in order to:

1. Update an existing item.

2. Delete an existing item.

HTTP Verb	URL	Description
POST	/api/2.0/app-settings	Create a new application setting.
PUT	/api/2.0/app-settings/1	Update application setting uniquely identified by **id** 1.
DELETE	/api/2.0/app-settings/1	Delete application setting uniquely identified by **id** 1.

Create a new item   In order to create a new resource (e.g application setting resource) you must use the collection entry point and do a POST request:

```
POST /api/2.0/app-settings
Content-Type: application/json
Content-Length: 49

{"name": "default_user_locale", "value": "en_US"}
```

Update an existing item   In order to update an default_locale application setting resource you must do the following request:

```
PUT /api/2.0/app-settings/1
Content-Type: application/json
Content-Length: 44

{"name": "default_locale", "value": "ro_RO"}
```

Of course partial requests are also supported:

```
PUT /api/2.0/app-settings/1
Content-Type: application/json
Content-Length: 18

{"value": "ro_RO"}
```

It is recommended to send the minimum amount of data to the API in order to optimize your application.

Delete an existing item   Delete requests are pretty simple as they do not have any body in the response.

REST Responses

When working with resources API it is important to understand what responses might be returned in order to correctly consume them into your clients. In this document you can find success / exception responses coming from APIs.

Common responses

For each call there are some common responses that might be returned by APIs:

Internal Server Error When working in distributed environments there are unexpected situations occuring (server failing, dns failing, and so on). In all this cases a 500 HTTP Status code will be returned and an html generated page will be sent in the body of the error. Do not make assumptions about the format of this response as it might change in the future.

Concrete errors In Fantastico generated APIs, concrete errors follows a given format:

```
{
    "error_code": <a numeric error code used for uniquely identifying the situation>,
    "error_description": <a user friendly message in English describing the error>,
    "error_details": <an http link where you can find more details about the error which occured>
}
```

Resource Item

Retrieve resource item When retrieving a resource from a collection (**GET /api/2.0/app-settings/1**) the following responses might be returned:

- 200 OK

  ```
  200 OK
  Content-Type: application/json
  Content-Length: 53

  {"id": 1, "name": "default_locale", "value": "en_US"}
  ```

- 404 Not Found

  ```
  404 Not Found
  Content-Type: application/json
  Content-Length: 160

  {"error_code": 10001, "error_description": "Resource 1 does not exist.", "error_details": "h
  ```

Create a new resource item When creating a resource into a collection (**POST /api/2.0/app-settings**) the following responses might be returned:

- 201 Created

  ```
  201 Created
  Content-Type: application/json
  Content-Length: 0
  Location: /api/2.0/app-settings/123
  ```

  If you want to fetch the newly created resource just follow the location header.

- 400 Bad Request

  ```
  400 Bad Request
  Content-Type: application/json
  Content-Length: 173

  {"error_code": 10010, "error_description": "Resource 1 requires field (name|value).", "error
  ```

Update an existing resource

When updating an existing resource (**PUT /api/2.0/app-settings**) the following responses might be returned:

- 204 No Content

```
204 No Content
Content-Type: application/json
Content-Length: 0
```

- 400 Bad Request

```
400 Bad Request
Content-Type: application/json
Content-Length: 173

{"error_code": 10010, "error_description": "Resource 1 requires field (name|value).", "error
```

Delete an existing resource   When deleting an existing resource (**DELETE /api/2.0/app-settings**) the following responses might be returned:

- 204 No Content

```
204 No Content
Content-Type: application/json
Content-Length: 0
```

- 400 Bad Request

```
400 Bad Request
Content-Type: application/json
Content-Length:

{"error_code": 10020, "error_description": "Resource 1 is still in use.", "error_details": "
```

Bulk creation of items   Fantastico APIs do not support bulk creation of items. We do not intend to add this kind of capability.

3.6 OAUTH2

In Fantastico, a very modern authorization framework (**OAUTH2**) was choosen for guaranteeing:

1. Easy security for REST APIs.

2. Easy integration of 3rd party applications.

3. Easy integration of various Identity Providers.

OAUTH2 specification contains many scenarios for its usage and provide various flows:

1. Authorizaton code grant.

2. Implicit grant.

3. Resource owner password credentials grant.

4. Client credentials grant.

In order to understand all this flows you can read the official OAUTH2 documentation.

3.6.1 Fantastico security

In order to keep things as simple as possible, in Fantastico we currently support only **authorization code grant**. Moreover, you can find some particularities of Fantastico implementation:

- We only support **Authorization code grant** (for all use cases where protected resources are involved).

- We only support **Resource owner password credentials grant** for Anonymous access to resources.

- For ease of tokens introspection, we use JWT as bearer tokens

- We fully support scopes

- We support state parameter for avoiding Cross Site Request Forgery

Example (Simple Menu API)

Lets consider a virtual resource called SimpleMenu which has the following API:

HTTP Verb	URL	Description	Required permissions
GET	/api/1.0/simple-menus	Retrieve all menus.	
POST	/api/1.0/simple-menus	Create a new system menu.	simple_menus.create
PUT	/api/1.0/simple-menus/:id	Update an existing menu.	simple_menus.update
DELETE	/api/1.0/simple-menus/:id	Delete an existing menu.	simple_menus.delete

Terminology

In OAUTH, we always talk about three concepts:

1. Resource (Image, Melody, Menu).

2. Resource Owner (an Identity Provider which can authenticatate the entities owning resources).

3. Client (an application which wants access to resources owned by Resources owner in order to provide useful features).

In the above example, Simple Menu is a resource owned by a system (global resource). Resource owners are all persons who are granted at least one scope required by the resource:

- Everyone has anonymous access to the menu

- Everyone granted **simple_menus.create**,**simple_menus.update** or **simple_menus.delete** permissions is Resource Owner.

It is important to understand that permissions are called **scopes** in OAUTH2 specification.

Display all menus
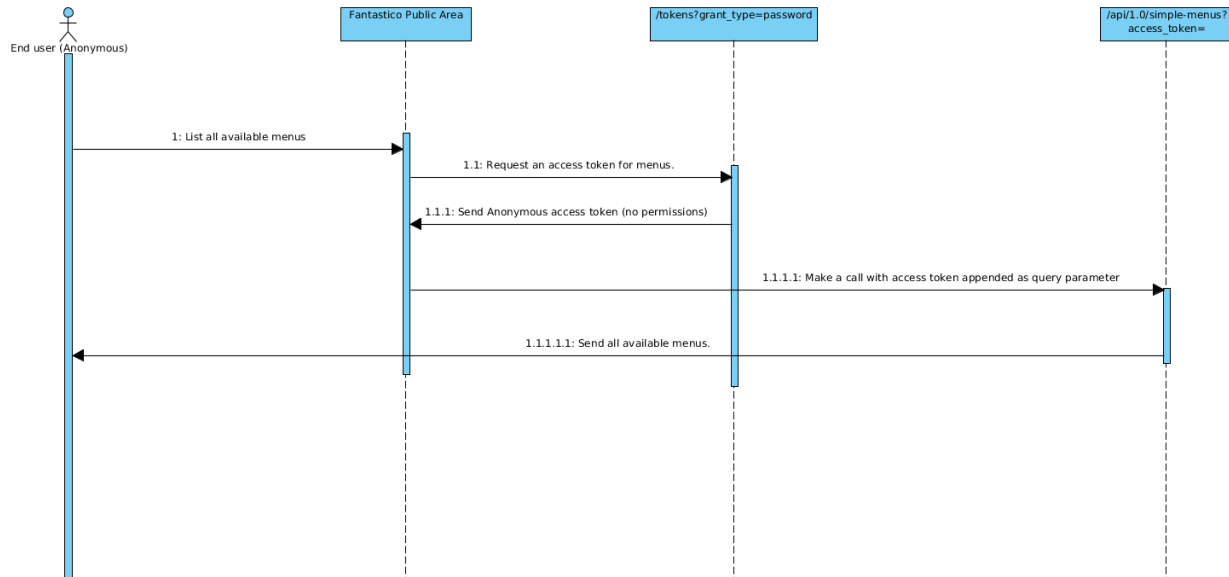
As you can see in the above example, each **Fantastico** project will benefit from an **Anonymous** user. This user is considered the **Resource owner** for all public accessible resources. This mean, that each visitor of a Fantastico application will not be forced to authenticate when it is not necessary (**e.g display website menubar**).

Create a new menu
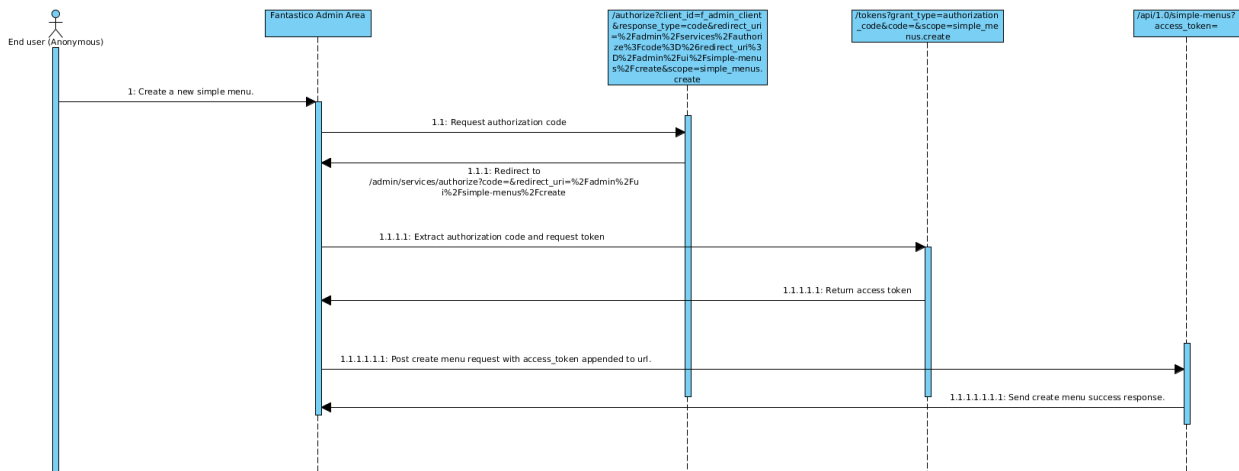
The above diagram specifies only the standard steps from OAUTH2 specification. In reality, when authorize endpoint is reached **End user** browser is redirected to a login screen. Once logged in, **End user** browser is redirected back to **/authorize** url from where he is granted an authorization code.

Once the user is granted an authorization code, he will exchange the authorization code for an access_token. With the newly obtained access token he can easily create a new simple menu.

Update an existing menu

It is similar to *Create a new menu* with the only difference that the final API called is **PUT /api/1.0/simple-menus/:id**.

Delete an existing menu

It is similar to *Create a new menu* with the only difference that the final API called is **DELETE /api/1.0/simple-menus/:id**.

Cautions

You must understand the following steps are one time:

1. Authenticate user

2. Obtain authorization code

3. Obtain access token

Once the token expires, the flow will start again from step **1.1**.

OAUTH2 Fantastico Tokens

In Fantastico, authorization codes and access tokens are opaque values for all humans (encrypted). They contain information into them used for easily invalidating them and expiring them. In addition, they do not require a persistent store and allows endpoint authorization without storage calls (increased performance and response time).

Authorization code structure

Authorization codes are opaque values for humans with the following structure (encrypted):

```
{
    "client_id": "s6BhdRkqt3",
    "scopes": ["simple_menus.create", "simple_menus.update", "simple_menus.delete"],
    "userid": "john.doe@gmail.com",
    "state": "unique generated value",
    "creation_time": "1380137542",
    "expiration_time": "1380137651"
}
```

Each authorization code from Fantastico OAUTH2 implementation contains the following fields:

1. client_id

    • The client unique identifier for which authorization code was requested.

1. scopes

    • A JSON list of scopes for which this token was generated. (this will be transmitted to the access token)

2. user id

    • user unique identifier (possibly email address) which can be used to obtain additional information about the user.

3. state

    • A key uniquely generated and hard to guess.

    • It is used to protect against **Cross Site Request Forgery (CSRF)**.

4. creation_time

> • The timestamp when this token was generated.

5. expiration_time

> • The timestamp when this token expires (must be a really short period - 1min).

Access token structure

```
{
   "client_id": "s6BhdRkqt3",
   "scopes": ["simple_menus.create", "simple_menus.update", "simple_menus.delete"],
   "user": {
      "id": 1,
      "email: "john.doe@gmail.com",
      "first_name": "John",
      "last_name": "Doe"
   },
   "state": "unique generated value",
   "creation_time": "1380137651",
   "expiration_time": "1380163800",
   "token_type": "bearer"
}
```

Each access token from Fantastico OAUTH2 implementation contains the following fields:

1. client_id

> • The client unique identifier for which token was requested.

2. scopes

> • A JSON list of scopes for which this token was generated. (used for endpoint authorization).

3. user id

> • User unique identifier (used for obtaining additional information about an user).

4. user email address

> • User email address (as stored in his profile).

5. user first name

> • Might be used in certain secure components.

6. user last name

> • Might be used in certain secure components.

7. state

> • A key uniquely generated and hard to guess.
>
> • It is used to protect against **Cross Site Request Forgery (CSRF)**.

8. creation_time

> • A timestamp indicating the date when this token was generated.

9. expiration_time

> • A timestamp indicating the date when this token is going to expire.

Obtaining access tokens

In order to obtain an access token which can be used for calling **Fantastico** APIs requires the following:

1. Obtain an authorization code

   ```
   GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz&redirect_uri=https%3A%2F%2Fclie
   Host: oauth2.fantasticoproject.com
   ```

2. client.fantasticoproject.com/cb?code=<unique one time only code>&state=<unique generated state>

   - Automatically obtain an access token starting from the given autorization code.

     ```
     POST /token HTTP/1.1
     Host: client.fantasticoproject.com
     Content-Type: application/json


     {
         "grant_type": "authorization_code",
         "code": <unique one time only code>,
         "redirect_uri": "https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb",
         "state": "unique generated value"
     }
     ```

3. provide access token for client usage

   ```
   {
       "access_token": "<encrypted token containing all specified information"
   }
   ```

OAUTH2 Fantastico Tokens Storage

For OAUTH2, we need a Relational storage.

Relational storage

In the relational storage we will hold all information describing registered clients as well as their configuration. Each client is described by the following attributes:

1. client_id

   - Client unique identifier.

2. client_name

   - A user friendly client_name: e.g (fantastico_admin)

3. client_secret

   - A client secret key used for signing every authorization code and every token type.

4. required_scopes

   - Holds a list of all required scopes of this client.

5. grant_type

   - Holds the supported grant type for this client.

   - Only **authorization_code** and **password** are valid values.

6. idp

   • Holds the Identity Provider Identifier which can authenticate users for this client.

7. otp_expiration_interval

   • Holds the number of seconds for which a one time token is valid (e.g **60**).

8. access_expiration_time

   • Holds the number of seconds for which access tokens are valid (e.g **3600**)

Clients registration   Clients registration in **Fantastico** is done manually by inserting new records into **clients** table.

Considerations

At the moment all generated tokens are transient and can be validated based on their decrypted content. This introduces a minor limitation at endpoint invocation because each client needs to also append its client id as a header.

In addition, no tokens audit is hold and we can not revoke tokens till they expire.

OAUTH2 Fantastico Error Responses

3.7 SDK

Starting with version **0.3.0** of Fantastico framework all dispersed shell scripts are unified under Fantastico Software Development Kit. In addition, the sdk is complemented by autogenerated documentation.

3.7.1 Intro

Fantastico sdk was developed with the following requirements in my mind:

   • Allow developers to manage Fantastico projects easily (using a single uniform command line). This is similar to many other frameworks (e.g **android sdk**).

   • Allow easily extension of sdk through plugins (e.g: activate off the shelf components into my project).

   • Create a uniform way to provide feedback to developers (prompt user for data, show help messages, support parameters).

   • Make the sdk compliant with linux way of developing command lines.

3.7.2 Usage

In this section you can find samples of how to use the sdk and how to make it available in older projects.

```
# For versions prior to **0.3.0**
pip install fantastico -U

fsdk --help
```

When you invoke fantastico sdk with **--help** argument it will list all available commands. Similar to other linux command lines you can obtain help hierarchical:

```
# Show help screen for fantastico <command>
fsdk <command> --help
```

In order for Fantastico SDK to work correctly make sure your project is on the **PYTHONPATH**. If **PYTHONPATH** is not set correctly you will not be able to use some sdk extensions.

3.7.3 Supported commands

Activate extension command

This Fantastico command helps developers integrate existing components into their project very easy. One use case is to activate in your projects contrib components (e.g: *Dynamic menu*). It is strongly recommended to use this sdk command because it works on every supported operating system.

class fantastico.sdk.commands.command_activate_extension.**SdkCommandActivateExtension**(*argv*, *cmd_factory*

This class provides the functionality for activating off the shelf fantastico extensions. As developer, it is extremely easy to integrate provided functionality into fantastico. For now, it supports only local extensions provided into fantastico.contrib package. In the future, we plan to support activation of remote components into projects.

```
# replace <project_root_path> with your fantastico project location.
cd <project_root_path>

# replace <component_root_path> with your actual folder.
fsdk activate-extension --name dynamic_menu --comp-root <component_root_path>
```

exec(*os_lib=<module 'os' from '/mnt/jenkins_ebs/continous_integration/fantastico_doc_workspace/pip-deps/lib/python3.2/os.py'>*)
This method is executed to activate the given extension name.

get_arguments()
This method returns support arguments of activate-extension command.

get_help()
This method returns the friendly help message describing the method.

Fantastico command

SDK

Syncdb command

Syncdb command is used to keep **Fantastico** projects databases updated. It is extremely easy to use it and in addition works on every operating system. In order to familiarize with components model read *Component model*.

class fantastico.sdk.commands.command_syncdb.**SdkCommandSyncDb**(*args*, *cmd_factory*, *settings_facade_cls=<class 'fantastico.settings.SettingsFacade'>*)
This class provides the algorithm for synchronizing **Fantastico** projects database scripts with the current configured database connection. Below you can find the order in which scripts are executed:

1. Scan and execute all activated extensions sql/module_setup.sql scripts.

2.Scan and execute all activated extensions for sql/create_data.sql scripts.

syncdb command first required database structure for all modules and the it populates them with necessary data. It is important to understand that rollback procedures are not currently in place and there is no way to guarantee data integrity. All components providers are responsible for writing module_setup in such a way that in case of error data is left in a consistent state.

For possible examples of how to structure a component read *Component model*

```
fsdk syncdb --db-command /usr/bin/mysql --comp-root samples
```

It is important to understand that this command will synchronize all module_setup / create_data sql scripts for current active settings. Read more about configuring fantastico on *Fantastico settings*.

exec(*os_lib=<module 'os' from '/mnt/jenkins_ebs/continous_integration/fantastico_doc_workspace/pip-deps/lib/python3.2/os.py'>, call_cmd=<function call at 0x1a012f8>*)
This method executes module_setup.sql and create_data.sql scripts.

> **Raises** **fantastico.sdk.sdk_exceptions.FantasticoSdkCommandError** When scripts execution fails unexpectedly.

get_arguments()
This method returns support arguments for **syncdb**:

> 1.-d –db-command path to mysql command.
>
> 2.-p –comp-root component root folder.

get_help()
This method returns the friendly help message describing the method.

Version command

This command tells you what is the current installed version of **Fantastico SDK**.

class `fantastico.sdk.commands.command_version.`**`SdkCommandVersion`**(*argv, cmd_factory, version_reader=<module 'fantastico' from '/mnt/jenkins_ebs/continous_integration/fa*
This class provides the command for finding out installed version of Fantastico SDK. The value is defined in fantastico root module code.

```
# display help information for version command in sdk context
fsdk version --help

# display the current sdk version
fsdk version
```

exec(*print_fn=<built-in function print>*)
This method prints the current fantastico framework version.

get_help()
This method returns the friendly help message describing the method.

3.7.4 Technical summary

class `fantastico.sdk.fantastico.`**`SdkCore`**(*argv*,   *cmd_factory=<class   'fantas-tico.sdk.sdk_core.SdkCommandsRegistry'>*,   *sup-ported_prefixes=None*,   *settings_facade_cls=<class 'fantastico.settings.SettingsFacade'>*)

This class provides the core functionality of Fantastico Software Development Kit. It wires all available commands together and handles requests accordingly. To better understand how sdk is designed see the following class diagram:
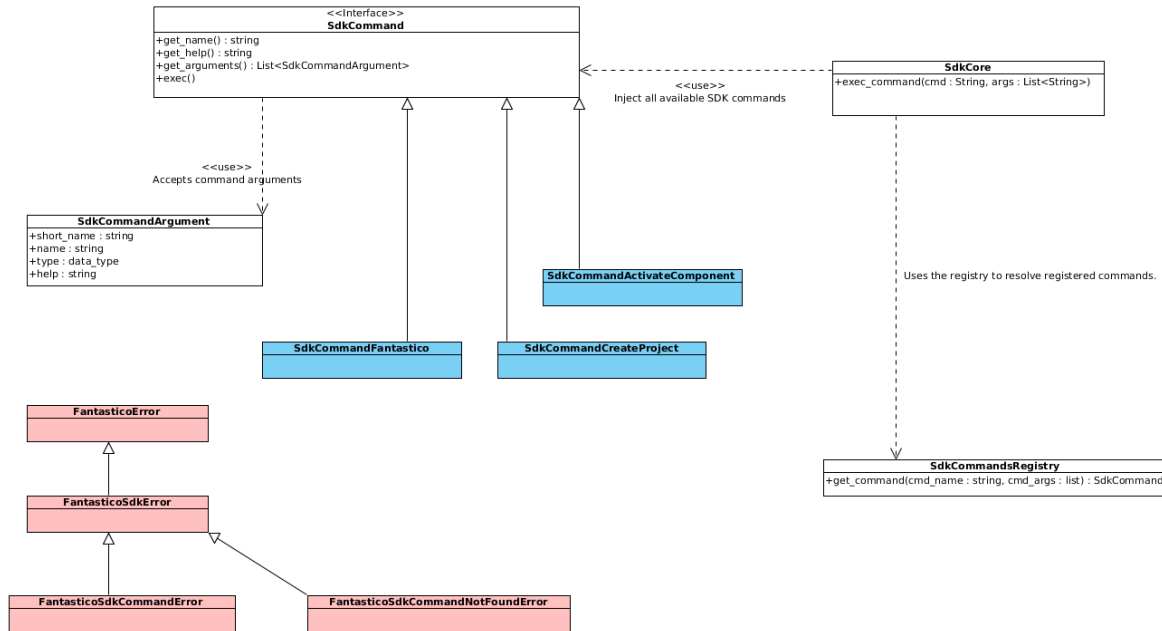
As you can see in above diagram, sdk core is just the main entry point of Fantastico Software Development Kit. It wires all available sdk commands together and it adds support for uniformly executes them and pass them arguments..

`exec`()

This method does nothing because fantastico is designed to accept only registered subcommands.

`get_arguments`()

This property retrieves support fantastico arguments.

`get_help`()

This method returns the friendly help message describing the method.

class `fantastico.sdk.sdk_core.`**`SdkCommandsRegistry`**

This class holds all registered commands available to use in the sdk. It is important to understand that commands and subcommands are registered by name and must be unique. This is because, by design, each command can easily become a subcommand for another command. It facilitates very flexible extension of sdk and reusage of existing commands.

static **`add_command`**(*cmd_name*, *cmd_cls*)

This method registers a new command using the given name.

    Parameters

- **cmd_name** (*str*) – Command name used to uniquely identify the command.

- **cmd_class** (`fantastico.sdk.sdk_core.SdkCommand`) – A subclass of sdk command.

> > **Raises** [fantastico.sdk.sdk_exceptions.FantasticoSdkError](#) If the given name is not unique or cmd class is wrong.

> static **get_command**(*cmd_name*, *cmd_args*)
> > This method retrieve a concrete sdk command by name with the give args passed.

> > **Parameters**

> > > • **cmd_name** (*str*) – The registered command name we want to instantiate.

> > > • **cmd_args** (*list*) – a list of arguments received from command line.

> > **Returns** Command instance.

> > **Return type** `fantastico.sdk.sdk_core.SdkCommand`

> > **Raises** [fantastico.sdk.sdk_exceptions.FantasticoSdkCommandNotFoundError](#) if command is not registered.

class `fantastico.sdk.sdk_core.`**SdkCommandArgument**(*arg_short_name*, *arg_name*, *arg_type*, *arg_help*)

> This class describe the attributes supported by a command argument. For a simple example of how arguments are used read `fantastico.sdk.sdk_core.SdkCommand`

> **help**
> > This read only property holds the argument help message.

> **name**
> > This read only property holds the argument name. Name property will represent the long name argument available for sdk commands. E.g: **–name**.

> **short_name**
> > This read only property holds the argument short name. Short name property will represent the short name argument available for sdk commands. E.g: **-n**.

> **type**
> > This read only property holds the argument type.

class `fantastico.sdk.sdk_core.`**SdkCommand**(*argv*, *cmd_factory*)

> This class provides the contract which must be provided by each concrete command. A command of sdk is just and extension which can provide custom actions being executed by Fantastico in a uniform manner.

> Below you can find a simple example of how to implement a concrete command:

> In the previous example, we have shown that all received arguments from command line are magically provided into **self._arguments** attribute of the command.

> When a sdk command is instantiated with a list of command line arguments the first element from the list must be the command name. This happens because all arguments passed after a command name belongs only to that command.

> **exec**()
> > This method must be overriden by each concrete command and must provide the command execution logic.

> > **Raises** [fantastico.sdk.sdk_exceptions.FantasticoSdkCommandError](#) if an exception occurs while executing the command.

> **exec_command**(**args*, ***kwargs*)
> > This method provides a template for executing the current command if subcommands are present. Internally it invokes overriden exec method.

> > **Raises**

- **fantastico.sdk.sdk_exceptions.FantasticoSdkCommandError** – if an exception occurs while executing the command.

- **fantastico.sdk.sdk_exceptions.FantasticoSdkCommandNotFoundError** – if a subcommand does not exist.

get_arguments()
> This method must be overriden by each concrete command and must return the command supported arguments.

class `fantastico.sdk.sdk_decorators.`**SdkCommand**(*name*, *help*, *target=None*, *settings_facade_cls=<class 'fantastico.settings.SettingsFacade'>*)

> This decorator describe the sdk commands metadata:

> 1. name

> 2. target (which is the main purpose of the command. E.g: fantastico - this mean command is designed to work as a subcommand for fantastico cmd).

> 3. help (which describes what this method does). It will automatically contain a link to official fantastico documentation of the command.

> It is used in conjunction with `fantastico.sdk.sdk_core.SdkCommand`. Each sdk command decorated with this decorator automatically receives **get_name** and **get_target** methods.

class `fantastico.sdk.sdk_exceptions.`**FantasticoSdkError**

> This is the base exception used to describe unexpected situations occuring into fantastico sdk. Below you can see the sdk hierarchy of concrete exceptions.
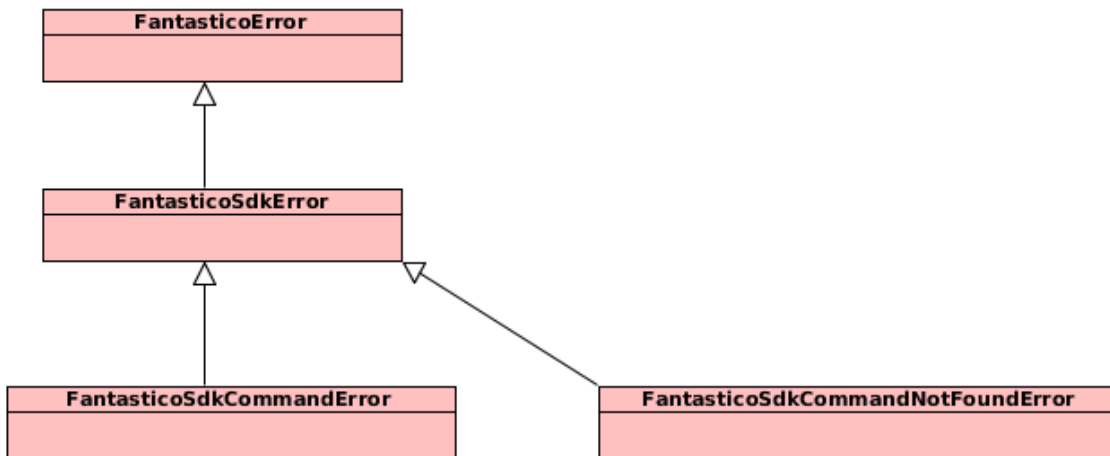
class `fantastico.sdk.sdk_exceptions.`**FantasticoSdkCommandError**

> This class describe an exception which occured into one of fantastico sdk commands.

class `fantastico.sdk.sdk_exceptions.`**FantasticoSdkCommandNotFoundError**

> This class describe an exception which occurs when we try to execute an inexistent command.

3.8 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.

3.8.1 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.

3.8.2 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.

For easily synchronization of sql scripts with a **Fantastico** project database read *Syncdb command*

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));
```

Once the component is activated (*Activate extension command*) and structure is synchronize data must be created into
the new tables. You can find such a script example below. It is up to you where you place **sql/create_data.sql**.

```
################################################################################
# Copyright 2013 Cosnita Radu Viorel
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and a
# documentation files (the "Software"), to deal in the Software without restriction, including withou
# the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
# and to permit persons to whom the Software is furnished to do so, subject to the following conditio
#
# The above copyright notice and this permission notice shall be included in all copies or substantia
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT N
# WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SH
# COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONT
# ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWAR
################################################################################

DELETE FROM mvc_friendly_messages;
INSERT INTO mvc_friendly_messages(Message)
VALUES ('Hello world!!!!'),
       ('Greetings from Australia!!!!');
```

Both scripts are autodected and run by sdk *Syncdb command*.

3.9 Component reusage

class fantastico.rendering.component.**Component**(*environment,*
                                                                                    url_invoker_cls=<class   'fantas-
                                                                                    tico.rendering.url_invoker.FantasticoUrlInternalInvoker'>)
    In fantastico, components are defined as a collection of classes and scripts grouped together as described in
Component model. Each fantastico component provides one or more public routes that can be accessed from a
browser or from other components. This class provides the mechanism for internal component referencing.

    In order to gain a better understanding about internal / in process component referencing we assume **Blog**
component provides the following public routes:

        •**/blog/articles/<article_id>** - Retrieves information about an article.

        •**/blog/ui/articles/<article_id>** - Displays an article within a html container.

    The first url is a simple json endpoint while the second url is a simple html dynamic page. When we want to
reuse a datasource or an dynamic html page in fantastico is extremely easy to achieve. Lets first see possible
responses from the above mentioned endpoints:

```
/* /blog/articles/<article_id> response */
{"id": 1,
 "title": "Simple blog article",
 "content": "This is a simple and easy to read blog article."}


<!-- /blog/ui/articles/<article_id> response-->


<div class="blog-article">
    <p class="title">Simple blog article</p>

    <p class="content">This is a simple and easy to read blog article.</p>
</div>
```

A very common scenario is to create multiple views for a given endpoint.

```
<!-- web service server side reusage -->
{% component url="/blog/articles/1", template="/show_blog_formatted.html", runtime="server" %}{%


<!-- show_blog_formatted.html -->
<p class="blog-title">{{model.title}}</p>
<p class="blog-content">{{model.content}}</p>
```

As you can see, json response is plugged into a given template name. It is mandatory that the given template exists on the component root path.

Also a very common scenario is to include an endpoint that renders partial html into a page:

```
<!-- html server side reusage -->
{% component url="/blog/ui/articles/1",runtime="server" %}{% endcomponent %}
```

Runtime attribute is used for telling Fantastico if the rendering needs to take place on server side or on client. Currently, only server side rendering is supported which actually means a page will be completed rendered on server and then the markup is sent to the browser.

In order to reduce required attributes for component tag, runtime attribute is optional with server as default value.

parse(*parser*)

This method is used to parse the component extension from template, identify named parameters and render it.

> **Parameters parser** (*Jinja 2 parser.*) – The Jinja 2 expression parser.
>
> **Returns** A callblock able to render the component.
>
> **Raises FantasticoInsufficientArgumentsError** when no / not enough arguments are provided to component.

render(*template='/raw_dump.html'*, *url=None*, *runtime='server'*, *caller=<function <lambda> at 0x2d971e8>*)

This method is used to render the specified url using the given parameters.

> **Parameters**
>
> > - **template** (*string*) – The template we want to render into the result of the url.
> >
> > - **url** (*string*) – The url we want to invoke.
> >
> > - **runtime** (*string*) – The runtime we execute the rendering into. Only **server** is supported for now.
> >
> > - **caller** (*macro*) – The caller macro that can retrieve the body of the tag when invoked.

>   **Returns** The rendered component result.
>
>   **Raises**
>
>   - **fantastico.exceptions.FantasticoTemplateNotFoundError** – Whenever we try to render
>   a template which does not exist.
>
>   - **fantastico.exceptions.FantasticoUrlInvokerError** – Whenever an exception occurs in-
>   voking a url within the container.

3.10 Built in components

Fantastico framework is really young and continuously improving. As of version **0.2.0** it is extremely easy to reuse
components provided urls in other context. This feature opens the possibility to provide common day by day used
components in new projects in order to accelerate development. In this document you can find a detailed list of built
in components as well as sample of how to use them:

3.10.1 Dynamic menu

Menus are a core part of every web site / application as well as mobile applications. More over, again and again
developers will want a quick way to define menu items without actually redefining menu data structure again and
again. This component which we generic named dynamic menu simply provides the controller and the model for easy
development of menus.

Integration

In order to use dynamic menu component within your project follow the steps below:

Component files activation deprecated

1. Create a symbolic link under your root components folder to dynamic_menu.

```
mkdir <components root>/dynamic_menu
cd <components root>/dynamic_menu
ln -s ../../pip-deps/lib/python[version]/site-packages/fantastico/contrib/dynamic_menu/sql .
ln -s ../../pip-deps/lib/python[version]/site-packages/fantastico/contrib/dynamic_menu/tests .
ln -s ../../pip-deps/lib/python[version]/site-packages/fantastico/contrib/dynamic_menu/*.py .
```

Component files activation (SDK)

```
fsdk activate-extension --name dynamic_menu --comp-root <comp root>
```

Component sample + db data

1. Create a template in one of your components in which you define the menu view:

```
<!-- *sample_menu.html* - simple snippet for creating a left / right side dockable menu. -->
{% for menu_item in model["items"] %}
    <a href="{{menu_item.url}}" title="{{menu_item.title}}" target="{{menu_item.target}}">{{menu_
{% endfor %}
```

2. In all views where you want to reuse the component you can paste the following snippet:

```
{% component template="sample_menu.html", url="/dynamic-menu/menus/1/items/" %}{% endcomponent %
```

3. Make sure you run **dynamic_menu/sql/module_setup.sql** against your configured database.

4. This script will create **menus** and **menu_items** tables into your database. Below you can find a sample script for creating a menu:

```sql
INSERT INTO menus(name) VALUES('My First Menu');
INSERT INTO menu_items(target, url, title, label)
VALUES ('_blank', '/homepage', 'Simple and friendly description', 'Home', <menu_id from previous
        ('_blank', '/page2', 'Simple and friendly description', 'Page 2', <menu_id from previous
        ('_blank', '/page3', 'Simple and friendly description', 'Page 3', <menu_id from previous
```

By default, when this component is first setup into an application, the sample menu mentioned above is created in database. You can test to see that dynamic menu works by accessing dev server url: http://localhost:12000/dynamic-menu/menus/1/items/.

Current limitations

Because **Fantastico** framework is developed using an Agile mindset, only the minimum valuable scope was delivered for **Dynamic Menu** component. This mean is not currently possible to:

• Localize your menu items.

• Display the menu items in the request language dynamically.

• Only first 100 menu items can be currently retrieved.

Technical summary

class `fantastico.contrib.dynamic_menu.menu_controller.`**`DynamicMenuController`**(*settings_facade*)
This class provides the controller for dynamic menus. The following routes are automatically made available when dynamic menu component is deployed:

> **/dynamic-menu/menus/<menu_id>/items/** – This route loads menu items from database and retrieve them in json format.

Below you can see a diagram describing relation model of the menu:
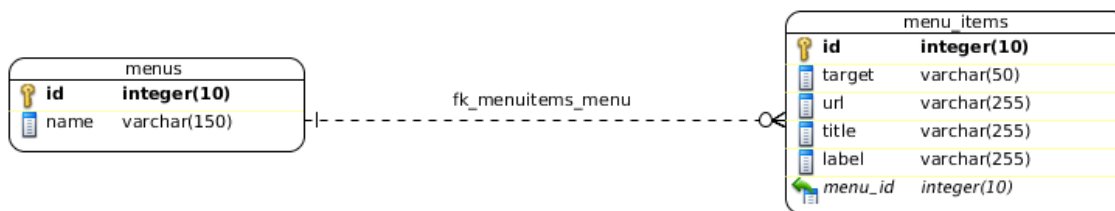
> **max_items**
> This property retrieves the maximum number of items allowed for a menu.

> **retrieve_menu_items**(**args*, ***kwargs*)
> This method is used to retrieve all items associated with a specified menu.
>
>> **Parameters**
>>
>> • **request** (*HTTP request*) – Http request being processed.
>>
>> • **menu_id** (*int*) – Menu unique identifier we want to retrieve information for.

> > > > **Returns** A JSON array containing all available menu items.
> > > >
> > > > **Raises** fantastico.contrib.dynamic_menu.menu_exceptions.FantasticoMenuNotFoundException
> > > > Whenever the requested menu does not exist.

class fantastico.contrib.dynamic_menu.menu_exceptions.**FantasticoMenuNotFoundException**
> This class defines a concrete fantastico menu not found exception raised whenever someone tries to access an inexistent menu attributes.

3.10.2 Dynamic pages

Most of the time, when a developer create a new web site or web application he follows the steps:

1. Create a set of templates (can be delegated to a web designer)

2. Create a new API or create a proxy API.

3. Create pages over the templates.

Many web sites / applications have a minimal set of master templates and all web pages follow those templates. This kind of approach keeps site consistency and decouple layouts from actual content. In **Fantastico**, it is extremely easy to work in this manner thanks to **Dynamic pages** extension.

Dynamic pages divides pages into two main parts:

1. Page meta information (title, keywords, description, language)

2. Page model / content (markup, text keys or any other kind of information).

Using dynamic pages you can easily add new web pages to your project without writing a single line of server side code.

Integration

1. Activate **Dynamic pages** extension.

   ```
   fsdk activate-extension --name dynamic_pages --comp-root <comp_root>
   ```

2. Add new dynamic pages to your project using **<comp_root>/sql/create_data.sql**.

   ```sql
   INSERT INTO pages(id, name, url, template, keywords, description, title, language)
   VALUES(1, '/en/home', '/en/home', '/frontend/views/master.html', 'keyword 1, ...', 'Home page',

   INSERT INTO page_models(page_id, name, value)
   VALUES(1, 'article_left', '<p class="hello_world">Hello world.</p>');

   INSERT INTO page_models(page_id, name, value)
   VALUES(1, 'article_right', '<p class="hello_world_right">Hello world right.</p>');
   ```

3. Update your project database

   ```
   fsdk syncdb --db-command /usr/bin/mysql --comp-root <comp_root>
   ```

4. Create **master.html** template file under **<comp_root>/frontend/views/**.

   ```html
   <!DOCTYPE html>

   <html lang="{{page.language}}">
      <head>
         <meta http-equiv="Content-type" content="text/html;charset=UTF-8" />
   ```

```html
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="keywords" content="{{page.keywords}}" />
        <meta name="description" content="{{page.description}}" />

        <title>{{page.title}}</title>
    </head>

    <body>
        <h1>{{page.article_left.value}}</h1>

        <h2>{{page.article_right.value}}</h2>
    </body>
</html>
```

After you integrated **dynamic pages** extension into your project you can access
http://localhost:12000/dynamic/test/default/page from a browser. You should see a very simple dynamic page
rendered.

Current limitations

In the first version of this component (part of **Fantastico 0.4**) there are some known limitations:

- Create / Delete / Update / Bulk listing API are not provided. You can do this through create_data.sql script.

- There is no way to rewrite dynamic pages url so that they do not contain */dynamic* prefix.

Technical summary

class `fantastico.contrib.dynamic_pages.pages_router.`**`PagesRouter`**(*settings_facade*)
  This class provides the API for managing dynamic pages. In addition, it creates the
  special route **/dynamic/<page_url>** used to access pages stored in the database. From
  dynamic pages module perspective, a web page is nothing more than a relation be-
  tween `fantastico.contrib.dynamic_pages.models.pages.DynamicPage` and
  `fantastico.contrib.dynamic_pages.models.pages.DynamicPageModel`.
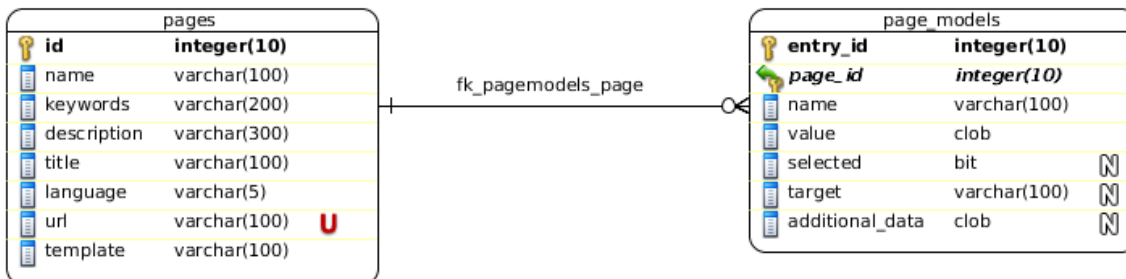
  A typical template for dynamic pages might look like:

```html
<!DOCTYPE html>

<html lang="{{page.language}}">
    <head>
        <meta http-equiv="Content-type" content="text/html;charset=UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="keywords" content="{{page.keywords}}" />
        <meta name="description" content="{{page.description}}" />
```

```
        <link href="/frontend/static/css/bootstrap-responsive.css" rel="stylesheet">
        <link href="/frontend/static/css/forhidraulic.css" rel="stylesheet">
        <title>{{page.title}}</title>
    </head>

    <body>
        <h1>{{page.article_left.value}}</h1>

        <h2>{{page.article_right.value}}</h1>
    </body>
</html>
```

serve_dynamic_page(**args*, ***kwargs*)
    This method is used to route all /dynamic/... requests to database pages. It renders the configured template
    into database binded to `fantastico.contrib.models.pages.DynamicPageModel` values.

class `fantastico.contrib.dynamic_pages.models.pages.`**DynamicPage**(*name=None,*
                                                                    url=None,   tem-
                                                                    plate=None, key-
                                                                    words=None,   de-
                                                                    scription=None,
                                                                    title=None,   lan-
                                                                    guage='en')
    This model holds meta information about dynamic pages. Below you can find all meta information for a dynamic
    page:

        1. id (unique identifier for a dynamic page)

        2. name

        3. url

        4. template

        5. keywords

        6. description

        7. language

    In a template used for rendering dynamic pages, you can easily access page meta information:

```
<p>Id: {{page.id}}</p>
<p>Name: {{page.name}}</p>
<p>Url: {{page.url}}</p>
<p>Template: {{page.template}}</p>
<p>Keywords: {{page.keywords}}</p>
<p>Description: {{page.description}}</p>
<p>Language: {{page.language}}</p>
```

    Usually it does not make sense to display dynamic page unique identifier but you can do it if necessary.

class `fantastico.contrib.dynamic_pages.models.pages.`**DynamicPageModel**(*page_id=None,*
                                                                        name=None,
                                                                        value=None)
    This class defines how page models looks like. A page model defines the actual content for en existing page.

3.10.3 Tracking codes

Every web application usually requires support for tracking visitors behavior. Most of the solutions can be easily integrated into a website by adding a small javascript snippet into every page you want to track. Below you can find some popular tracking solutions (free or commercial).

Analytic solutions

At the moment of writting this article, there are plenty of options available for web developers to track their website performance:

1. Google analytics (probably the most popular solution).

2. Reinvigorate.

3. KISSmetrics.

4. FoxMetrics.

5. Mint.

6. Open Web Analytics.

7. Clicky.

8. Mixpanel.

9. Chartbeat.

10. Adobe Web Analytics.

11. Chartbeat.

12. Inspectlet.

Of course there are many other solutions available out there. For more information about the above mentioned solution I recommend you read the excellent article posted by Aidan Huang.

Integration

Follow the steps from this section in order to enable tracking in **Fantastico** projects:

1. Activate tracking extension:

   ```
   fsdk activate-extension --name tracking_codes --comp-root <comp_root>
   ```

2. Add your tracking codes into database (easiest way is through *Syncdb command*)

3. Create a sql script similar to the one below and place it under **<comp_root>/sql/create_data.sql**:

   ```
   INSERT INTO tracking_codes(provider, script)
   VALUES ('Google Analytics', '
      <script type="text/javascript">
          var _gaq = _gaq || [];
          _gaq.push(["_setAccount", "UA-XXXXX-X"]);
          _gaq.push(["_trackPageview"]);

          (function() {
            var ga = document.createElement("script"); ga.type = "text/javascript"; ga.async = tru
            ga.src = ("https:" == document.location.protocol ? "https://ssl" : "http://www") + ".g
            var s = document.getElementsByTagName("script")[0]; s.parentNode.insertBefore(ga, s);
   ```

```
        }) ();
    </script>');
```

4. Update your project database

```
fsdk syncdb --db-command /usr/bin/mysql --comp-root <comp_root>
```

5. Use tracking codes in your pages:

```
{% component url="/tracking-codes/ui/codes/" %}{% endcomponent %}
```

Tracking component is rendering all available codes from the database. In order to check all available tracking codes configured in a **Fantastico** project visit http://localhost:12000/tracking-codes/ui/codes/. Once the page is loaded see page source.

Current limitations

In the first version of this component (part of **Fantastico 0.4**) there are some known limitations:

- No API provided for Create / Update / Delete operations.

Technical summary

class `fantastico.contrib.tracking_codes.tracking_controller.`**`TrackingController`**(*settings_facade*)

This class provides the tracking operations supported by **TrackingCodes** component.

`list_codes`(**args*, ***kwargs*)
This method provides tracking codes listing logic. It list all available tracking codes from database.

Parameters request (`webob.request.Request`) – The current http request being processed. Read *Request lifecycle* for more information.

Returns JSON list of available tracking codes. Can be empty if no tracking codes are defined.

`list_codes_ui`(**args*, ***kwargs*)
This method renders all available tracking codes.

class `fantastico.contrib.tracking_codes.models.codes.`**`TrackingCode`**(*provider*, *script*)
This class provides the model for tracking codes. It maps **tracking_codes** table to an object. In order to use this model please read *Model View Controller*.

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:

```python
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:

```python
@ControllerProvider()
class BlogsController(BaseController):
    @Controller(url="/blogs/(?P<blog_id>\\d{1,})/posts/$", method="GET",
                models={"Post": "fantastico.plugins.blog.models.posts.Post"})
    def list_blog_posts(self, request, blog_id):
        Post = request.models.Post

        blog_id = int(blog_id)

        posts = Post.get_records_paged(start_record=1, end_record=100,
```

```
                                        sort_expr=[ModelSort(Post.model_cls.created_date, ModelSort.A
                                                  ModelSort(Post.title, ModelSort.DESC)],
                                        filter_expr=[ModelFilter(Post.model_cls.blog_id, blog_id, Mod

                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 first 100 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
<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 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)
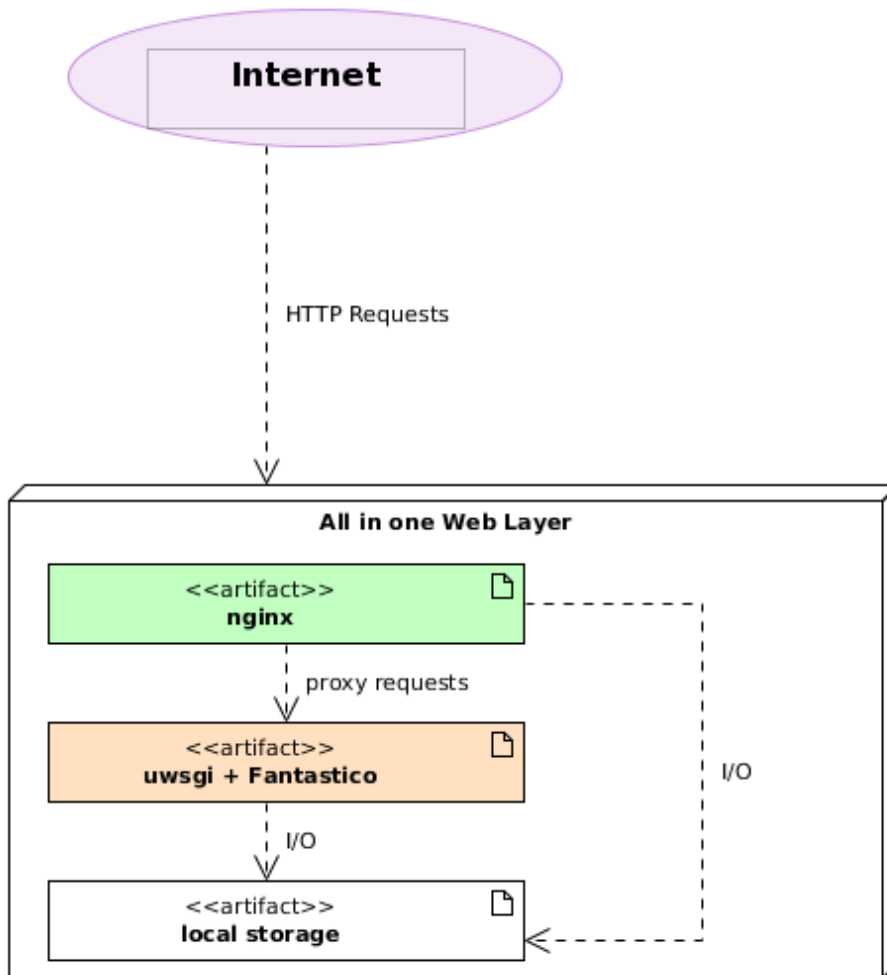
Above diagram described the simplest scenario for rolling out Fantastico to production. You can use this scenario for minimalistic web applications like:

- 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

3. export ROOT_PASSWD=<your root password>

4. fantastico_setup_low_usage_<os_distribution) –ipaddress <desired_ip> –vhost-name <desired_vhost> –uwsgi-port <uwsgi port> –root-folder <desired root folder> –modules-folder <desired modules folder> (e.g fantastico_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
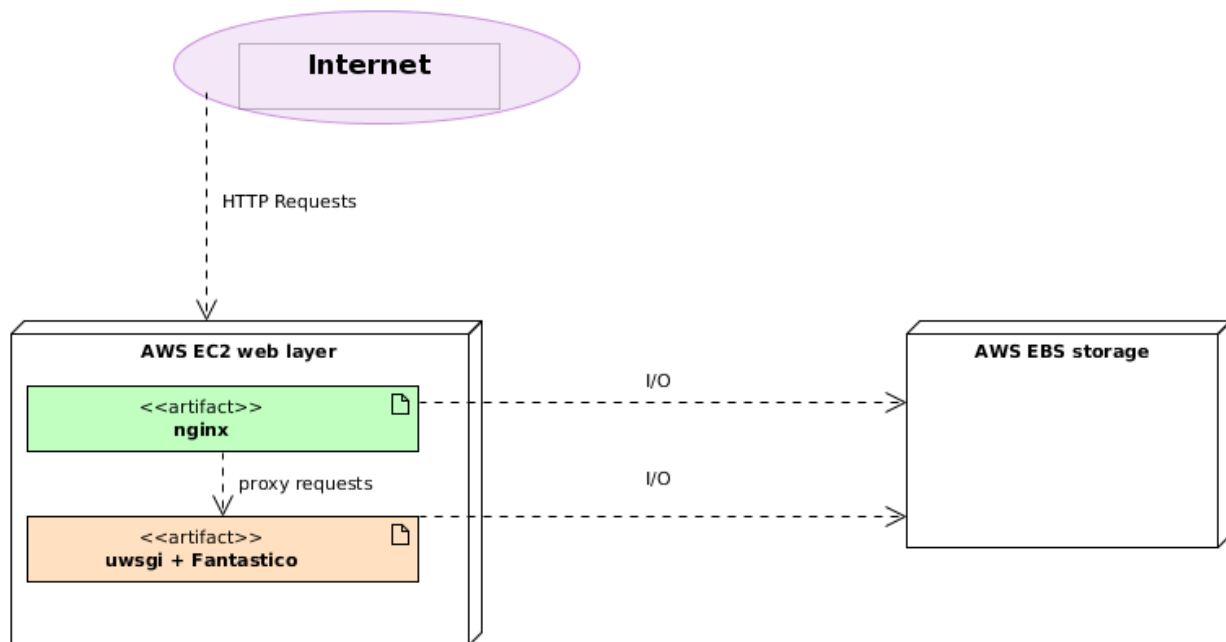
This scenario is a little bit more complex than *Low usage (simplest scenario)* but it provides some advantages:

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

6. fantastico_setup_low_usage_<os_distribution).sh (e.g fantastico_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:
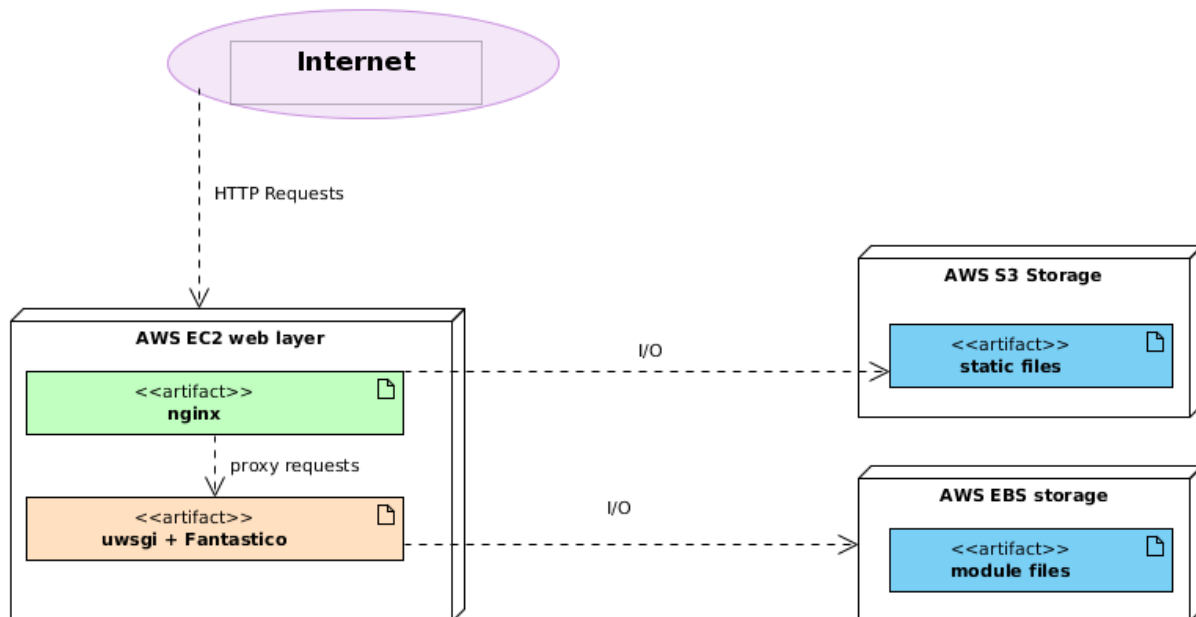
You can read more about **AWS S3** storage on http://aws.amazon.com/s3/. In this version of fantastico there is no way to sync static module files with S3 buckets. This feature is going to be implemented in upcoming **Fantastico** features. As a workaround you can easily copy **static** folder content 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. fantastico_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 fantastico_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 *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 though it 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.

Favicon

If you want your site to also have an icon which is automatically presented by browsers, in your project root folder do the following:

1. mkdir static

2. cd static

3. Copy your favicon.ico file in here.

4.3.2 Static assets on prod

There is no difference between static assets on dev and static assets on production from routes point of view. From handling requests point of view, nginx configuration for your project takes care of serving static assets and sending correct http caching headers.

4.4 Creating a new project

A new Fantastico based project can be easily setup by following this how to. In this how to we are going to create a project named **fantastico_first**.

1. cd ~/

2. mkdir fantastico_first

3. cd fantastico_first

4. virtualenv-3.2 –distribute pip-deps

5. . pip-deps/bin/activate

6. pip install fantastico

7. fantastico_setup_project.sh python3.2 my_project

The last step might take a while because it will also install all fantastico dependencies (e.g sphinx, sqlalchemy, ...). Please make sure your replace python3.2 with the correct python version. In order to test the current project do the following:

1. fantastico_run_dev_server

2. Access http://localhost:12000/fantastico/samples/mvc/static/sample.jpg

3. Access http://localhost:12000/mvc/hello-world

Your newly project is setup correctly and it runs fantastico default samples project.

4.4.1 Create first component

After the new project it's correctly setup we can create our first component.

1. . pip-deps/bin/activate

2. export FANTASTICO_ACTIVE_CONFIG=my_project.settings.BaseProfile

3. cd my_project

4. mkdir component1

5. cd component1

6. mkdir static

7. Paste an image into static folder (e.g first_photo.jpg)

8. touch __init__.py

9. touch hello_world.py

10. Paste the code listed below into hello_world.py

```python
from fantastico.mvc.base_controller import BaseController
from fantastico.mvc.controller_decorators import ControllerProvider, Controller
from webob.response import Response

@ControllerProvider()
class HelloWorldController(BaseController):
    '''This is a very simple controller provider.'''

    @Controller(url="/component1/hello")
    def say_hello(self, request):
        '''This method simply returns an html hello world text.'''

        msg = "Hello world from my project"

        return Response(content_type="text/html", text=msg)
```

11. fantastico_dev_server

12. Now you can access Hello route.

13. Now you can access First photo route.

4.4.2 Customize dev server

For understanding how to customize dev server please read *Development mode*

4.4.3 Customize uwsgi prod server

By design, each Fantastico project provides built in support for running it on uWSGI server. If you want to customize uwsgi parameters for your server you can follow these steps:

1. cd $FANTASTICO_PROJECT_FOLDER/deployment/conf/nginx

2. nano fantastico-uwsgi.ini

3. Change the options you want and save the file.

4. fantastico_run_prod_server (for testing the production server).

5. Be aware that first you need an nginx configured and your project config file deployed (Read *Deployment how to*).

CHANGES

5.1 Feedback

I really hope you enjoy using Fantastico framework as much as we love developing it. Your feedback is highly appreciated so do not hesitate to get in touch with us (for support, feature requests, suggestions, or everything else is on your mind): Provide feedback

5.2 Versions

- **v0.4.1 (Provide feedback)**

    - Fix a bug into analytics component sample data insert.

    - Fix a bug into component rendering for no json responses coming for given url.

- **v0.4.0 (Provide feedback)**

    - Fantastico SDK commands display official link to command documentation.

    - Fantastico SDK syncdb command.

    - Standard detection of database tables module setup / data insert created.

    - Multiple tracking codes extension integrated into fantastico contrib.

    - Dynamic pages extension integrated into fantastico contrib.

    - Direct feedback channel integrated into documentation (Provide feedback)

- **v0.3.0**

    - Fantastico SDK core is available.

    - Fantastico SDK activate-extension command is available.

    - Samples of how to activate extensions for an existing project are provided.

- **v0.2.2**

    - Update dynamic menu activation documentation.

    - Fix a serious bug in engine management and too many sql connections opened.

    - Fix a bug in db session close when an unexpected error occurs when opening the connection.

    - Add extensive unit tests for db session management.

- **v0.2.1**

- **Fix** packaging of pypi package. Now it is usable and contains rendering package as well as contrib package.

- **v0.2.0**

    - Framework documentation is tracked using Google Analytics

    - Component reusage is done using {% component %} tag.

    - Dynamic menu pluggable component can be used out of the box.

    - MVC documentation improvements.

    - Fix a bug in DB session management cache when configuration was changed at runtime.

- **v0.1.2**

    - Nginx config file now also maps www.<vhost_name>

    - Redirect support from controllers

    - Setup fantastico framework script does not override deployment files anymore

- **v0.1.1**

    - Favicon route handling.

    - Deployment scripts error handling and root folder execution (rather than execution only for deployment subfolder).

    - MVC how to article was changed to use get_records_paged instead of all_paged method (it used to be a bug in documentation).

    - DB Session manager was changed from one singleton connection to connection / request.

    - FantasticoIntegrationTestCase now has a property that holds os environment variable name for setting up Fantastico active config.

- **v0.1.0**

    - Built in router that can be easily extended.

    - WebOb Request / Response architecture.

    - Request context support for accessing various attributes (current language, current user and other attributes).

    - Multiple project profiles support.

    - Database simple configuration for multiple environments.

    - Model - View - Controller support.

    - Automatic model facade generator.

    - Model facade injection into Controllers.

    - Templating engine support for views (jinja2).

    - Documentation generator for pdf / html / epub formats.

    - Automatic framework packaging and deployment.

    - Helper scripts for creating projects based on Fantastico.

    - Easy rollout script for running Fantastico projects behind nginx.

    - Rollout scenarios for deploying Fantastico projects on Amazon (AWS).

    - How to sections for creating new projects and components using Fantastico.

PROVIDE FEEDBACK

Provide feedback

BUILD STATUS

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

EIGHT

LICENSE

R

S

T

U

W